



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Model-Driven Engineering of Planning and Optimisation Algorithms for Pervasive Computing Environments

Anthony Christopher Harrington

A thesis submitted to the University of Dublin, Trinity College
in fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

November 2010

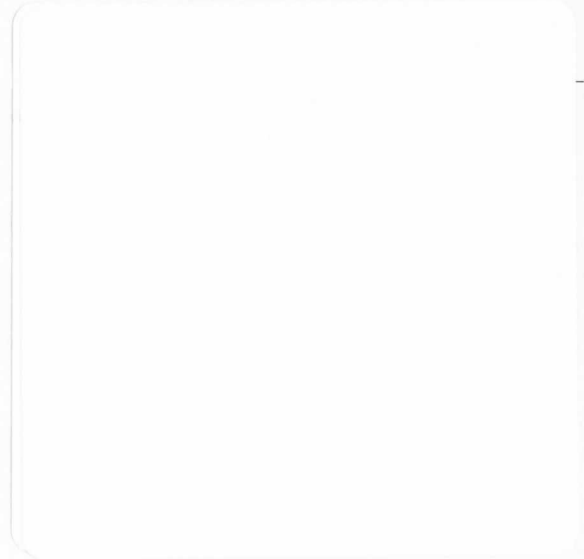


Thesis 9340

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

I agree that Trinity College Library may lend or copy this thesis upon request.



Acknowledgements

I would like to acknowledge the assistance of all those who helped me in completing my PhD. Firstly I am very grateful to my supervisor Prof. Vinny Cahill for giving me the opportunity to do both my MSc and PhD degrees and for his patience and attention to detail throughout the process. The thesis is all the better for his help. I also wish to thank Dr. Siobhan Clarke, as head of the Distributed Systems research group (DSG), for the practical assistance and teaching opportunities she has given me over the years.

I completed my undergraduate degree at University College Dublin (UCD). In 1852, the founder of UCD, John Henry Newman, wrote in a lecture entitled, "On the Idea of a University", that the goal of a university community should be to give birth to a living teaching, which in time takes the shape of a self-perpetuating tradition, that imbues and forms, more or less and one by one, all those who come under its influence.

My experience of the DSG community has given me an insight into what he meant. Throughout my postgraduate studies in Trinity, I have made good friends from over 25 different countries. I have learned much through these friendships and I am happy to have added my story to the DSG tradition. I would like to particularly acknowledge some of the friends who made my time in DSG very enjoyable: Dominik Dahlem, Eamonn Dillon, Andronikos Nedos, Raymond Cunningham, Tim Walsh, Vinny Reynolds, Cormac Driver, Aline Senart, Melanie Bouroche, Stefan Weber, Daire O'Broin, Jan Sacha, Marcin Karpinski, Mads Haahr, Elizabeth Daly, Kulpreet Singh, Peter Barron, Barbara Hughes, Elizabeth Gray, Ivana Dusparic, Gregor Gaertner, Jim Dowling and Greg Biegel.

I also wish to thank my family for their encouragement and support over the years. I am very proud of you all. Finally and most especially, I wish to thank Paulina for her patience, understanding and love during the time it has taken me to complete the thesis.

Now it is time for new challenges and new adventures!

Anthony Christopher Harrington
University of Dublin, Trinity College
November 2010

Abstract

This thesis addresses the development of that class of pervasive computing applications that can be modelled as real-world planning and optimisation problems. Canonical examples of such applications are the control of transportation infrastructures, activities such as region-wide pollution monitoring and emergency-service management. The formal foundations of planning and optimisation problems are well established, however applying these techniques to pervasive computing applications is still a challenging task.

Challenges arise from dealing with the dependence on sensor data for inferring application state and on actuators to effect changes in the environment. Both of these dependencies introduce unavoidable uncertainty into the decision/execution cycle of such applications. Other considerations such as the possibility of dynamism in the environment, which may be manifested through uncontrolled changes in application state, and application complexity due to large inter-connected state spaces, also impact on the design of planning and optimisation algorithms in pervasive computing applications.

Sophisticated planning and optimisation algorithms can be used in conjunction with state inference techniques to manage uncertainty in pervasive computing environments. However writing and deploying such algorithms is a non-trivial task and knowledge of these algorithms is not widespread among application developers. The large range of potential applications, ranging from the management of smart spaces to the control of complex real-world infrastructure such as transport systems, precludes a unified algorithmic approach to the solution of such problems and increases the expertise required of application developers.

This thesis presents a model-driven approach to developing pervasive computing applications that exploits design-time information to support the engineering of planning and optimisation algorithms that reflect the presence of uncertainty, dynamism and complexity in the application domain. Specifically, the engineering approach presented in this thesis addresses the tasks of selecting an appropriate algorithm and mapping it to the application domain.

In order to support a broad range of potential applications we provide a domain model for pervasive computing environments so that the application and its domain may be profiled and the information used to seed algorithms with appropriate sensor and actuator models and state-space information. The accuracy and spread of the sensor and actuator infrastructure is identified and made available to state-inference algorithms.

The layered domain model provides a set of object-oriented specifications for modelling physical and sensor/actuator infrastructure and application states. These specifications are implemented using XML and SQL data standards. The domain model elements are combined using spatial queries to determine the characteristics and properties of the environment relevant to the engineering of planning and optimisation algorithms. Spatial queries are supported through the tagging of domain model elements with a spatial context and the use of Geographic Information Systems modelling techniques.

Our model-driven engineering approach is implemented in a series of transformation algorithms. The initial transformation algorithm parses the domain model and generates a planning model for the application being developed that contains the specified states, actions and rewards. The second transformation algorithm parses the planning model and selects and seeds a planning or optimisation algorithm for use in the application. The domain expert can specify the algorithm to be used. If no algorithm is specified then the transformation algorithm will make an automatic selection of an appropriate algorithm. The criteria used to select an appropriate algorithm are embedded in a taxonomy that relates problem types and domain properties to algorithm types. The criteria are derived from encoding existing approaches from the literature.

As proof of concept we provide a library of planning and optimisation algorithms and the second transformation instantiates one of these algorithms with data from the planning model.

The methodology is evaluated through the development of planning and optimisation algorithms for two scenarios that are representative of the target class of applications. The expressiveness of the domain model and the functionality of the transformations is showcased when presenting the scenarios. We present an empirical evaluation of the impact of the methodology on the development effort associated with the two scenarios and provide a quantitative evaluation of the performance of the algorithms produced by the transformations.

The case-studies demonstrate that the methodology supports code reuse both at the level of the algorithm library and at the level of domain model elements across applications. Domain model elements can be reused as appropriate across different applications. The evaluation also shows how the methodology facilitates the rapid re-engineering of algorithms in response to extended application requirements.

The methodology is also shown to encode expert knowledge in the taxonomy in matching domain characteristics to algorithm types and in the mapping from the planning model to algorithm-library components. The knowledge required of the domain modeller when using the methodology is shown to be generally restricted to the application domain. Exceptions to this are clearly indicated.

The quantitative evaluation focuses on the appropriateness of the engineered algorithms through the use of simulation data to measure performance. The trade-off between the convenience of the generated algorithms and the accuracy of hand-crafted solutions is shown.

Contents

Acknowledgements	iii
List of Figures	xi
List of Tables	xiii
List of Algorithms	xiv
Nomenclature	xvi
1 Introduction	1
1.1 Motivation	2
1.2 Aims and Objectives	4
1.3 Model-Driven Engineering	5
1.4 Automated Planning and Optimisation	7
1.5 Requirements	9
1.6 Our Methodology	10
1.6.1 Development Roles in the methodology	10
1.6.2 Using the Methodology	11
1.6.3 Middleware Assumptions	13
1.7 Thesis Contributions	13
1.7.1 Limitations	14
1.8 Roadmap	14
2 Inference, Planning and Optimisation for Pervasive Computing	16
2.1 Overview of Planning and Optimisation Tasks	17
2.2 Characteristics of Pervasive Computing Environments	18
2.2.1 Modelling Real-world Environments	18
2.2.2 Uncertainty in Pervasive Computing Environments	19
2.2.3 Dynamism in Pervasive Computing Environments	19
2.2.4 Complexity in Pervasive Computing Environments	20
2.3 State Inference	21
2.3.1 Probability Theory and Bayes Rule	22
2.3.1.1 Bayes Rule and Belief Updating	26

2.3.2	Bayesian Inference and Sensor Model	28
2.3.3	Bayesian Networks	30
2.3.3.1	Inference in Bayesian Networks	32
2.3.3.2	Inference with Uncertain Evidence	34
2.3.3.3	Criticisms of the Bayesian approach	39
2.3.4	Dempster-Shafer Belief Theory	41
2.3.4.1	Criticism of Dempster-Shafer Belief Theory	46
2.3.5	Sensor Fusion	47
2.3.5.1	Sensor Fusion Configurations	47
2.3.5.2	Sensor Fusion Topologies	49
2.3.6	Summary	51
2.4	Planning	52
2.4.1	One-Shot Decision Problems	52
2.4.1.1	Bayesian Decision Networks	53
2.4.2	Sequential Decision Making	54
2.4.2.1	Approximate POMDPs for Dynamic Environments	59
2.4.3	Summary	61
2.5	Optimisation	61
2.5.1	Optimisation in Pervasive Computing Environments	63
2.5.1.1	Heuristic Optimisation Algorithms	63
2.5.1.2	Multi-objective Optimisation	65
2.6	Summary	66
3	State of the Art	68
3.1	Scope and Structure of the Review	69
3.2	Resolving Uncertainty in Pervasive Computing	70
3.2.1	Bayesian Approaches	70
3.2.1.1	Representing and reasoning with uncertain context data in SOCAM	70
3.2.1.2	Resolving Uncertainty in Context Integration and Abstraction	71
3.2.2	Dempster-Shafer Context Sensing	72
3.2.3	Hybrid Approach to Deriving Context	73
3.2.4	Analysis	74
3.3	MDE in Pervasive Computing	75
3.3.1	Olympus	75
3.3.2	Physical Active Objects	78
3.3.3	PervML	80
3.3.4	uPAIN	82
3.3.5	Review	83
3.4	Planning and Optimisation in Pervasive Computing Environments	84
3.4.1	GPT	84

3.4.2	itSimple2.0	88
3.4.3	Planning in Gaia	90
3.4.4	Smart Bay	94
3.4.5	Review	96
3.5	Conclusions	97
4	Development Process	100
4.1	Requirements Review	101
4.2	Design Decisions	101
4.3	Using the Methodology	103
4.3.1	Tool Support	105
4.4	Domain Model Architecture	106
4.5	Domain Model Topology Abstraction	108
4.6	Domain Model Layers	109
4.6.1	Infrastructure Layer	110
4.6.2	Sensor/Actuator Layer	110
4.6.2.1	Sensor Elements	111
4.6.2.2	Actuator Elements	113
4.6.3	System-States Layer	115
4.6.3.1	Inference Function	118
4.6.3.2	Factored State Space	121
4.6.4	Domain Specific Knowledge Layer	122
4.7	Policy Specification	124
4.8	Case Studies	127
4.8.1	Junction Controller Scenario	127
4.8.1.1	Scenario Domain Model	127
4.8.1.2	Policy	133
4.8.2	CCTV Camera Selection Scenario	135
4.8.2.1	Scenario Domain Model	135
4.8.2.2	Policy	144
4.9	Conclusion	146
5	Model Transformations and Code Generation	147
5.1	Planning Model Components	148
5.1.1	Modelling Partially-Observable State Spaces	148
5.1.1.1	Planning Model Sensor Fusion Support	149
5.1.1.2	Planning Model State Inference Support	153
5.1.2	Optimisation Algorithm Support	154
5.2	Domain-Model Transformation Algorithm	155
5.3	Algorithm Library and Selection Support	159
5.3.1	Extending the Algorithm Library	163

5.3.1.1	Algorithm Engineering Example	163
5.4	Planning-Model Transformation	167
5.4.1	Determining System-State Properties	168
5.5	Control-Unit Operation	169
5.5.1	Control Units for Planning Algorithms	169
5.5.2	Control Units for Optimisation Algorithms	170
5.5.3	Control Units and Middleware Services	171
5.6	Case Studies	173
5.6.1	Traffic Junction Controller	173
5.6.1.1	Domain-Model Transformation	173
5.6.1.2	Planning-Model Transformation	175
5.6.2	CCTV Camera Selection	180
5.6.2.1	Domain-Model Transformation	180
5.6.2.2	Planning Model Transformation	181
5.7	Conclusion	182
6	Evaluation	184
6.1	Requirements and Design Review	185
6.1.1	Design Decisions Review	186
6.2	Evaluation Criteria	189
6.2.1	Evaluating MDE Systems	189
6.2.2	Evaluating Automated Planning Systems	190
6.2.3	Methodology Evaluation Criteria	191
6.3	Evaluation Strategy	192
6.3.1	Development Effort of using Methodology	193
6.3.1.1	Modularity	193
6.3.1.2	Abstraction	194
6.3.1.3	Metrics for development effort	194
6.3.2	Metrics for automated planning and optimisation	197
6.3.2.1	Metrics for Control Unit Performance	197
6.4	Evaluation Platform	198
6.4.1	Evaluation Platform Services	198
6.4.2	Evaluation Platform Implementation	198
6.5	Junction Controller Scenario Evaluation	200
6.5.1	Development Effort	201
6.5.2	Automated Planning Evaluation	202
6.5.2.1	Range	203
6.5.2.2	Control Tuning	204
6.5.2.3	Problem Size	204
6.5.2.4	Algorithm Selection	204
6.6	Junction Controller Scenario Extension	206

CONTENTS

6.6.1	Development Effort for Extension	208
6.6.1.1	Degree of Automation	208
6.6.1.2	Algorithm Selection	209
6.7	CCTV Selection Scenario	209
6.7.1	Development Effort	209
6.7.2	Automated Planning Evaluation	211
6.7.2.1	Range	211
6.7.2.2	Control Tuning	212
6.7.2.3	Problem Size	216
6.7.2.4	Algorithm Selection	216
6.8	CCTV Selection Scenario Extension	219
6.8.1	Development Effort for Extension	221
6.8.1.1	Degree of Automation	221
6.8.2	Algorithm Selection	222
6.8.2.1	SA Algorithm in Extended Scenario	226
6.9	Evaluation Summary	229
6.9.1	Evaluation Findings	229
6.9.2	Requirements Coverage	231
7	Future Work and Conclusions	233
7.1	Achievements	233
7.2	Future Work	234
7.2.1	Domain Modelling	234
7.2.2	Algorithm Library	235
7.2.3	Transformations and Code Generation	235
7.3	Conclusions	237
A	Complexity	239
B	Fuzzy Logic	243
C	Bayesian Network for Junction Interest	248
D	Decision Network for Extended Junction Controller Scenario	252
E	System State XSD Schema	255
	Bibliography	258

List of Figures

1.1	(a) MDE development lifecycle, (b) Methodology development lifecycle. . .	6
1.2	Major steps in an MDE development process.	7
1.3	Conceptual model for dynamic planning.	8
1.4	Methodology lifecycle phases.	11
2.1	Planning and optimisation problems and algorithms.	18
2.2	Inputs to a Bayesian inference process.	29
2.3	Bayesian Network Sample and Connections: (a) D-Separation; (b) Earth- quake/Alarm Example	31
2.4	Bayesian Network sample and CPTs: (a) Sample Bayesian network; (b) Sam- ple conditional probability tables.	33
2.5	Jeffrey's rule and conditional independence.	35
2.6	A virtual node in a Bayesian network.	39
2.7	Properties of commonly implemented sensor fusion patterns.	48
2.8	Sensor fusion architectures.	49
2.9	Smart sensor.	51
2.10	Markov Decision Process framework: (a) MDP; (b) Partially Observable MDP	55
2.11	The optimal t-step value function is the upper surface of the value functions associated with all t-step policy trees.	59
2.12	A search tree for an approximate POMDP.	60
2.13	Easy and Hard problems for global optimisation: (a) a global algorithm will easily avoid θ_{local} and find θ^* ; (b) a global algorithm will only find θ_{local} as it is effectively impossible to find. θ^*	62
2.14	Pareto-optimal frontier.	67
3.1	Context Sensing architecture	73
3.2	Conceptual architecture for inferring context	74
3.3	PAO abstraction levels.	79
3.4	Jini-based framework for supporting PAOs.	80
3.5	PervML development process.	81
3.6	The itSimple design process and integrated languages.	88
3.7	itSimple schema for planning environment.	89

LIST OF FIGURES

3.8	The Gaia Planning Framework	91
3.9	AFME control process.	95
4.1	Methodology development process.	104
4.2	Development process tool support.	106
4.3	Layered domain model.	107
4.4	Infrastructure layer.	110
4.5	Sensor and actuator layer.	112
4.6	System-states layer.	116
4.7	Inference function inputs.	121
4.8	Junction Controller scenario layer 1 infrastructure element data.	128
4.9	Junction 1244 phase elements.	130
4.10	Bayesian network used to calculate junction-interest.	138
4.11	Tables for junction-interest network.	139
4.12	MaximalDistance calculation.	143
5.1	A Bayesian decision network generated to run a traffic junction controller.	177
6.1	LOC effort to implement the Junction Controller and CCTV Selection case studies.	202
6.2	Junction Controller scenario control unit performance.	205
6.3	Extended Junction Controller scenario control unit performance.	206
6.4	The extended Bayesian decision network for junction controllers.	207
6.5	CCTV Selection scenario SMOSA performance for 50 runs.	213
6.6	CCTV Selection scenario SMOSA performance for 100 runs.	214
6.7	CCTV Selection scenario SMOSA performance for 500 runs.	215
6.8	CCTV Selection scenario SA performance for 50 runs.	217
6.9	CCTV Selection scenario SA performance for 100 runs.	218
6.10	CCTV Selection scenario SA performance for 500 runs.	219
6.11	Extended CCTV Selection scenario SMOSA performance for 50 runs.	224
6.12	Extended CCTV Selection scenario SMOSA performance for 100 runs.	225
6.13	Extended CCTV Selection scenario SMOSA performance for 500 runs.	225
6.14	Extended CCTV Selection scenario SA performance for 50 runs.	227
6.15	Extended CCTV Selection scenario SA performance for 100 runs.	227
6.16	Extended CCTV Selection scenario SA performance for 500 runs.	228
B.1	Fuzzy Membership Time Context Function	244
B.2	Fuzzy Membership Functions: (a) Speed / Acceleration membership function; (b) Fuzzy Controller for cruise control example.	246
B.3	Fuzzy Membership Functions: (a) Acceleration membership function with value of 6/mps ; (b) Fuzzy logic controller	246

List of Tables

2.1	Widget manufacturers.	23
2.2	Full joint probability distribution for P(Manufacturer, Fault).	24
2.3	Dempster-Shafer example evidence combination.	44
2.4	Updated evidence combining data from sensor 3.	45
2.5	Input/Output Data Fusion Model	48
3.1	Summary of reviewed MDE approaches in pervasive computing.	84
3.2	Summary of reviewed planning and optimisation approaches in pervasive computing.	97
3.3	Related work and methodology requirements.	98
4.1	Reward model for the junction controller scenario.	135
5.1	Algorithm library and taxonomy.	161
5.2	Junction Controller scenario domain properties and problem types.	175
5.3	Junction Controller decision network reward model.	177
5.4	Junction Controller scenario - belief updating with virtual evidence.	179
5.5	CCTV Selection scenario domain properties and problem types.	180
6.1	Design designs and evaluation criteria.	192
6.2	Mapping from design decisions to evaluation criteria.	192
6.3	Junction Controller scenario cyclomatic complexity.	201
6.4	Automated component generation.	203
6.5	MEU algorithm and domain model planning assumptions.	203
6.6	Junction Controller scenario sensor invocation.	204
6.7	Junction Controller scenario extended decision network reward model.	207
6.8	CCTV Camera Selection scenario cyclomatic complexity.	210
6.9	SMOSA planning assumptions.	212
6.10	CCTV Selection scenario control unit performance.	214
6.11	CCTV Selection scenario sensor invocation.	222
6.12	CCTV Selection scenario normalised maximum values.	223

List of Algorithms

1	Value Iteration algorithm for calculating utilities of states.	56
2	Real-Time-Belief State Search (RTBSS) algorithm.	60
3	Local search algorithm.	64
4	Simulated Annealing algorithm.	64
5	SMOSA algorithm.	66
6	Domain-model transformation logic.	157
7	Planning-model transformation logic.	167
8	Planning problem control unit.	170
9	Optimisation problem control unit.	170
10	Control-unit evaluation.	172
11	Adding evidence to a decision network.	178

Listings

3.1	Sample Olympus program.	77
3.2	Complete GPT model of a racetrack planning problem.	86
3.3	Gaia PDDL.	93
3.4	Prolog.	93
4.1	Action XSD schema.	114
4.2	XML schema for system-states specification.	119
4.3	Inference function template.	120
4.4	Policy XSD schema.	126
4.5	Emergency-vehicle sensor meta-data excerpt.	129
4.6	Traffic-demand sensor meta-data excerpt.	129
4.7	Actuator definition for junction controller 1244.	131
4.8	Excerpt from EmergencyVehiclePresent system-state definition.	132
4.9	EmergencyVehiclePresent system-state inference function.	133
4.10	Excerpt from junction controller scenario policy definition.	134
4.11	JunctionInterest system-state definition.	137
4.12	Inference function for JunctionInterest system-state.	141
4.13	Excerpt from DegreeOfInterest system-state definition.	142
4.14	DegreeOfInterest inference function.	143
4.15	Excerpt from MaximalDistance system-state definition.	145
4.16	Inference function for MaximalDistance system-state.	145
4.17	CCTV selection scenario policy excerpt.	146
5.1	TrafficDemand inference function.	152
5.2	Python State-Generater API for optimistaion problems.	155
5.3	Algorithm taxonomy.	164
5.4	Execution cycle of SMOSA optimisation control unit.	165
5.5	Execution cycle of MEU planning control unit.	177
5.6	Mapping likelihood data to chance nodes in a decision network.	179
6.1	Junction Controller scenario extension to policy.	207
6.2	Inference function for NumberEmergencyVehicles system-state.	220
6.3	CCTV Selection scenario extension to policy	221
E.1	XML Schema for System-States specification	255

Nomenclature

Basic Notation	
$(A \setminus B)$	set A with all elements of B removed
$\emptyset$	the empty or null set
$\forall x$	for all x
$\neg A$	not A
$\prod_i f(x_i)$	the product of $f(x_i)$ indexed by i
$\sum_i f(x_i)$	the sum of $f(x_i)$ indexed by i
$A \cap B$	the union of A and B
$A \cup B$	the intersection of A and B
Optimisation	
$\hat{\theta}_k$	An estimate for θ at time k
Θ	Domain of allowable values for a vector θ
θ	Vector of parameters being estimated in an optimisation process
θ^*	The solution set for Θ that minimises $L(\theta)$
$g(\theta)$	A root finding function
$L(\theta)$	A scalar valued loss function
T	Temperature
Planning	
A	Action, set of actions
Σ	Discrete Event System
o, O	Sensor observation, set of observations

LISTINGS

R Reward
s, S State, set of states

State Inference

Ω DS Frame of Discernment
 $\Omega^\perp$ Power Set of Ω
 $\text{Bel}(X)$ Posterior distribution or Belief over X
 $\text{Bl}(i)$ DS Belief Measure for proposition i
 $m(i)$ DS Probability Mass for proposition i
 $P(X = x)$ probability that X takes the value x
 $P(X)$ (prior, marginal) probability of X
 $P'(X)$ posterior distribution over X
 $\text{Pl}(i)$ DS Plausibility Measure for proposition i
DS Dempster Schafer Belief Theory

Into this wild Abyss the wary Fiend Stood on the brink of Hell and looked a while, Pondering his voyage; for no narrow frith He had to cross.

John Milton (Paradise Lost. Book 2. 1667)

1

Introduction

This thesis presents a novel approach to the development of that class of pervasive computing applications that can be modelled as real-world planning and optimisation problems. Canonical examples of such applications are the control of transportation infrastructures, activities such as region-wide pollution monitoring, and the management of emergency services and assisted living facilities.

A model driven engineering (MDE) methodology has been designed to allow developers to create planning and optimisation applications using a domain model that shields the developer from challenges posed by the uncertainty and dynamism of pervasive computing environments.

Planning problems are modelled as decision processes constructed over the set of system goals, the actions that the system is capable of taking and the decisions that must be taken to achieve the desired outcome or state. Optimisation problems are modelled as minimising or maximising a property of the system such as vehicle waiting time or throughput at a traffic junction.

Although the formal foundations of planning and optimisation problems are well established, the practical task of applying these formal foundations to large-scale problems is challenging (Ghallab et al. 2004). Pervasive or real-world computing environments make for challenging domains and applications must often operate in the face of significant uncertainty. Applications are typically deployed over distributed sensor and actuator infrastructure. These domains exhibit partial observability due to a dependence on sensor data for state determination with the attendant possibility of error or partial coverage in the sensor readings. State transitions may appear non-deterministic to the application due to actuator error or partial observability. The environment may be dynamic resulting in

state changes that are independent of actions taken by the application.

The complexity of real-world domains, the inference of system state from noisy data, and the possible unreliability of actuator platforms used for action selection motivates the use of probabilistic planning and optimisation algorithms (Spall 2003, 30).

Knowledge of probabilistic planning and optimisation algorithms is not typical among application developers. The practical task of programming these algorithms is also complicated by the requirement to integrate with a middleware architecture to access the sensor and actuator infrastructure.

The programming model supported by the MDE methodology uses design-time information to generate instances of planning and optimisation algorithms for use in applications. These algorithms are generated with hooks to an underlying middleware for accessing sensor and actuator infrastructure in the environment.

The design-time information is captured in a domain model constructed by a domain expert. The domain model describes the location and geometry of physical infrastructure, sensor and actuator infrastructure, and system state-space data relevant to the application.

A transformation engine has been developed to process domain models, to select planning and optimisation algorithms and generate application code. A domain-model transformation algorithm parses the domain model and generates a planning model for the application that contains the sets of system states, actions and rewards. The state-space is represented as a set of state-variable objects whose values change over time. As an example, consider the development of an application to monitor the level of carbon monoxide in all zones in a large industrial complex. The domain-model transformation will generate one state-variable object per zone with each object having an associated geometry reflecting the area over which it is defined. This geometry is used to support middleware discovery and lookup services that facilitate state-inference.

A planning-model transformation algorithm parses the planning model and generates a specification of a control unit for the application. A control unit embeds a planning or optimisation algorithm and the state inference routines used in the application.

The algorithm to be used is selected from a library of planning and optimisation algorithms. The domain modeller can specify the algorithm to be used. If no algorithm is specified then the second transformation will automatically select an algorithm that matches the problem type and domain properties.

1.1 Motivation

Our characterisation of the challenges faced in developing large-scale pervasive applications is based on experience of developing such systems in the Intelligent Transportation Systems (ITS) and supply-chain management domains.

The Journey Time Estimation System (JETS) (Harrington & Cahill 2004) was developed in association with the National Roads Authority of Ireland. The system uses sensor fusion techniques to provide historical journey time information on all sections of the

primary road network in Ireland. The journey times are calculated from GPS location and speed data obtained from a large sample set of probe vehicles. Vehicle positions and speeds obtained are mapped onto the national road network and this data is processed to provide accurate estimations of journey times. The chief development challenges posed by the project were the combination of a large state space extending over a national road network, with non-uniform quality of sensor data that varied with regional traffic patterns. Producing accurate journey times required that sensor fusion techniques be used to reflect the variation in sensor quality and spread.

The iTransit project was run in association with Dublin City Council and developed a prototype ITS architecture (Meier et al. 2005) (Meier et al. 2006) to support application development and integration in the transport domain in the Dublin metropolitan area. The principal research contribution was a modelling framework for specifying sensor and actuator platforms and integrating sensor data flows with diverse quality of service metrics. A prototype Traveller Information Service application built using this framework highlighted that the fidelity of the application is dependent on the deployment and diversity of sensor infrastructure in the environment.

The UTC-NG project investigated the development of a next-generation urban traffic control systems based on the use of increased floating vehicle data and decentralised learning strategies (Dowling et al. 2005). A UTC system must try to regulate the behaviour of a large number of vehicles capable of autonomous action. Significant uncertainty is present in the system due to the potential impacts of unpredictable events such as driver behaviour, accidents and weather conditions and this requires the use of probabilistic reasoning algorithms suitable for use in real-world environments.

The DESNA project designed and prototyped a development environment to facilitate the integration of sensor/RFID data with planning and optimisation algorithms and business processes (Senart et al. 2009). The project focused on intelligent supply chain management using sensor network data and provided a business process modelling tool based on the Business Process Execution Language (BPEL) modelling standard (Alves et al. 2006), extended to include constructs appropriate to integrating planning and optimisation activities.

These diverse projects displayed some common challenges such as the uncertainty of sensor data, the requirement for large-scale distributed applications, the need to track application state in dynamic environments, and the variability in approaches to planning and optimisation.

The solutions developed for these projects showed the effectiveness of sensor fusion and the usefulness of Geographic Information Systems (GIS) techniques when relating sensor and actuator infrastructure to application state. The experience gained from these projects informed the aims and objectives of the methodology.

1.2 Aims and Objectives

The methodology described in this thesis was designed with the aim of simplifying the use of planning and optimisation algorithms in pervasive computing environments. The research was conducted with the following objectives:

- Enable non-specialists to apply planning and optimisation algorithms.
Planning and optimisation in pervasive computing environments may require the use of probabilistic planning and heuristic optimisation algorithms, knowledge of which is not widespread among application developers. High-level abstractions should be provided to ease the task of specifying the information required to use such algorithms.
- Facilitate arbitrarily complex state inference.
Inferring state from distributed sensor infrastructure leads to a requirement for multi-sensor integration and fusion involving high-level challenges such as the integration of multiple sensors at the system architecture and control level to more specific issues, possibly mathematical or statistical, involved in the actual combination of multi-sensor information (Luo & Kay 1989). The methodology should provide support for three sub tasks: sensor selection, sensor data fusion, and inference about the system state.
- Allow a wide variety of problem types composed of potentially large distributed state spaces to be addressed.
The methodology should provide generic abstractions and support multiple planning and optimisation algorithms that can be used in a wide range of applications. It is also desirable to support distributed applications with large state spaces.
- Integrate with pervasive computing middleware.
Pervasive computing environments may appear to be stochastic and dynamic to the application. Application state values can change dynamically with the environment. This requires the use of online algorithms that can sense and react to changes in the deployment environment. If an application must plan and optimise online, then it must be able to integrate with a middleware that can provide access to sensor and actuator infrastructure.

These objectives motivate the use of a high-level programming model containing a set of abstractions to represent sensor and actuator infrastructure and application states. The objective of supporting many problem types motivates a structured approach to choosing and applying planning and optimisation algorithm. These observations motivate a design that incorporates techniques from the fields of MDE and automated planning.

1.3 Model-Driven Engineering

Model-Driven Engineering (Architecture/Development) (MDE/MDA/MDD) constitutes an approach to software design and development that focuses on the use of software-platform independent models and promotes a rising of abstraction from higher-level programming languages.

The MDE approach structures system specifications as models constructed in a domain appropriate language (Anneke et al. 2003). This language typically provides modelling abstractions that map to entities existing in the domain of interest and are often constructed using Unified Modelling Language (UML) profiles¹.

The MDE life-cycle as shown in Fig. 1.1(a) (from (Anneke et al. 2003)) is similar to a traditional software life-cycle. The same phases are identified but the focus is on models as software artefacts.

The initial specification constructed using the domain-specific language is referred to as a Platform Independent Model (PIM). This model is defined with a high level of abstraction that is independent of any implementation technology. This model captures the business level functionality of the system being developed and is typically expressed in the UML language.

Thereafter depending on the technology over which it is to be implemented, e.g., the Common Object Request Broker Architecture (CORBA) or Enterprise Java Beans (EJB) platforms, a transformation algorithm is applied to produce a model that is referred to as a Platform Specific Model or PSM (Anneke et al. 2003). This model is tailored to the implementation constructs available in a specific implementation technology. The final step in the development is the transformation of each PSM to application code.

MDE is considered by some to be the first true generational shift in programming technology since the introduction of compilers (Selic 2003) and can potentially change the way applications are developed (Fernandes et al. 2008). By automating many of the complex and routine programming tasks, MDE allows developers to focus on the functionality that the system should deliver and on its general architecture, instead of worrying about every technical detail inherent in the use of a programming language (Atkinson & Kuhne 2003). MDE transformations are always executed by tools as shown in Fig. 1.2(a).

The software lifecycle supported by our methodology is based on the MDE approach. Fig. 1.1(b) shows the lifecycle activities addressed in our methodology. The artefacts produced are a domain model, a planning model and a control unit. The domain model is specified using the domain-model abstractions that are presented in detail in Chapter 4. The policy describes desired functionality and is specified over the domain model. The domain model and policy are a computational independent model of application functionality. A transformation is applied to the domain model and policy to produce a planning model artefact. The planning model is a platform or algorithm independent representation of the application. A second transformation is applied to the planning model to

¹See www.uml.org for a catalogue of existing profiles

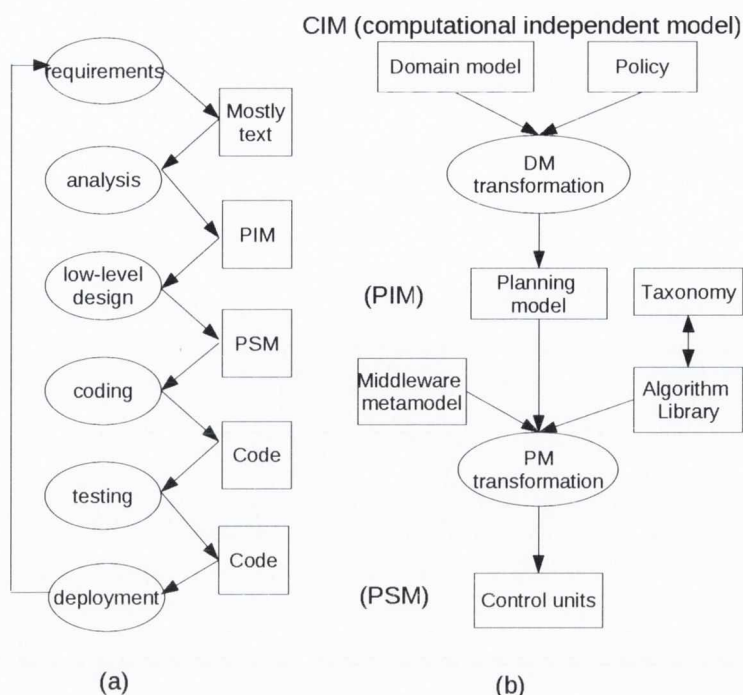


Figure 1.1: (a) MDE development lifecycle, (b) Methodology development lifecycle.

produce control units. Control units are composed with reference to a selected planning or optimisation algorithm and a model of the available pervasive computing middleware.

The transformations applied to the domain and planning model artefacts, shown in Fig 1.2(b), are automated. The transformations are described in detail in Chapter 5.

Among the benefits associated with adopting an MDA methodology (Anneke et al. 2003) (Fernandes et al. 2008) are:

- Gains in productivity.

MDE allows the developer to focus exclusively on the business logic when developing PIMs. The PSMs are generated automatically by transformation tools. Initially, someone will need to define the PIM to PSM transformation, which is a difficult and specialized task. But this transformation only needs to be done once and can be applied in the development of many systems (Anneke et al. 2003, 9). The automated transformation of code also leads to a reduction of costs and an accelerated development cycle (Mellor et al. 2003).

- Concepts close to the domain and reduction of semantic gap.

The development of software through models rather than through programming languages enables non-computing specialists with domain expertise to produce systems (Selic 2003). Also, because the model elements can map closely to constructs in the problem domain it should ease the task of modelling problems.

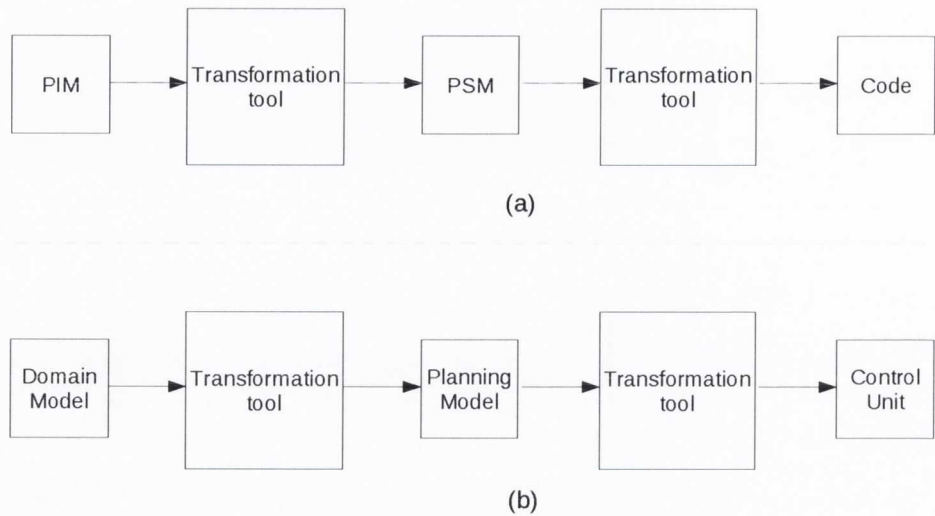


Figure 1.2: Major steps in an MDE development process.

- Capture of expert knowledge.
Expert knowledge is captured in the mapping functions that convey information to the transformation of one model to another. This enables an independent evolution of the models and leads to extended longevity of the models (Mellor et al. 2003).
- Portability and and re-use.
PIMs are platform independent and can be automatically transformed into multiple PSMs for different platforms.

Chapter 6 presents an empirical evaluation of our methodology based on applying the supported lifecycle model to the development of two case-study scenarios representative of the target class of applications. The evaluation presents metrics to gauge the impact of the methodology on development effort.

1.4 Automated Planning and Optimisation

Automated planning focuses on the design and use of information processing tools that give access to affordable and efficient planning resources. Planning involves choosing actions to change the state of a system. A conceptual model for planning in pervasive computing environments requires a general model for a dynamic system. A conceptual model for dynamic planning is shown in Fig 1.3. This model can be depicted through the interaction between the 3 components.

1. A state-transition system Σ , evolves as specified by its state-transition function, according to the events and actions it receives.

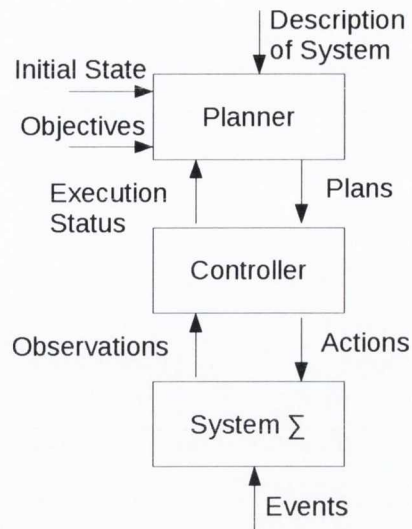


Figure 1.3: Conceptual model for dynamic planning.

2. A controller given as input the state of the system, provides as output, an action according to some plan.
3. A planner given as input a description of the system, an initial state and some objective, develops a plan for the controller in order to achieve the objective. Given access to the current state of the system a planner will provide a plan mapping states to actions.

Automated planners take as input a description of the problem to be solved and produce as output a plan to govern the actions taken by an application. Our methodology incorporates an automated planning component. The domain model abstractions contain information that is useful to selecting and applying planning algorithms.

Because we wish to support a wide variety of problem types we also provide support for optimisation algorithms. Optimisation algorithms operate by maximising or minimising properties of an application as specified by a domain modeller using the policy.

The conceptual model from Fig 1.3 is useful for understanding the integration of planning and optimisation algorithms in the control unit artefacts generated by the model transformations. The control units implement the functionality of the controller. The planning model contains the state-transition system information. The planner is an instance of a planning or optimisation algorithm selected from the library of algorithms provided.

To access the current state of the system the control units must be able to access sensor and actuator infrastructure in the deployment region. This requires that the control units must make use of a middleware for access and query operations.

Our methodology design synthesises concepts from the fields of MDE and automated planning. The MDE components address software engineering challenges associated with

developing the target class of pervasive computing applications. The automated planning component allows specialist knowledge to be encoded in the methodology tool-chain and reduces the knowledge of planning and optimisation algorithms required by developers.

1.5 Requirements

The methodology has been designed to support developers in the task of developing pervasive computing applications that use planning and optimisation algorithms. This is a multi-stage task that involves specifying application logic, sensor and actuator infrastructure, and the design and implementation of planning and optimisation algorithms.

Decomposing this multi-stage task leads to the following technical requirements:

- R1 - Allow application logic (policy) to be specified at a high-level.
A developer should be able to specify the application state space and desired behaviour without reference to planning and optimisation techniques and without reference to properties of the execution environment.
- R2 - Provide abstractions masking runtime conditions.
One of the difficulties when developing the target class of applications is that runtime conditions in pervasive computing environments are often unknown and unknowable at design time. The following information may be unknown at design time:
 - The availability and quality of sensor and actuator infrastructure. An application may be deployed over variable numbers of possibly mobile sensors with variable quality data.
 - The state of the environment. The state of the environment will often not be directly observable and may be uncertain.

In so far as possible it is desirable that both a domain expert and an AI developer should be able to use the methodology without explicit reference to runtime conditions relating to sensor/actuator infrastructure or uncertain state data.

- R3 - Simplify the development of planning and optimisation algorithms in pervasive computing environments.
A developer should be able to implement planning and optimisation algorithms without reference to properties of the deployment environment.
- R4 - Provide a development model that is extensible, that promotes reuse, and is suitable for use in large-scale applications.

The development model should allow applications to be engineered using a range of planning and optimisation algorithms. It should allow applications to be easily re-engineered and extended and should also scale to accommodate large data sets associated with large-scale applications.

It should also facilitate domain knowledge and algorithm code reuse. If a developer writes a planning or optimisation algorithm it should be available for reuse across multiple applications. Likewise domain knowledge should also be reusable across multiple applications where appropriate.

- R5 - Rapidly test algorithm efficacy.

At design time the following information may not be known:

- The most appropriate algorithm to use. It may not be possible to theoretically establish the most appropriate algorithm to use in advance. Empirical testing may be necessary to determine which algorithm is most suitable.
- The most appropriate values for algorithm parameters. Likewise it may not be possible to theoretically establish useful values of parameters for algorithms.

Developers should be able to use the methodology to rapidly evaluate the performance of a range of algorithms with a single domain model.

The parameters of these algorithms should be easily configurable as the optimal values will often be domain appropriate.

- R6 - Provide a consistent representation of uncertain data.

Uncertainty in state information is a characteristic of pervasive computing environments. The degree of uncertainty in application state data should be quantified. Handling the uncertainty in state estimates is algorithm specific. This leads to a requirement to provide an algorithm-independent, consistent format to handle uncertain data.

- R7 - Automatic selection of appropriate planning and optimisation algorithms.

Automated selection of appropriate algorithms is necessary to facilitate complete automation of the process that enables a domain modeller with no knowledge of planning and optimisation algorithms to produce a running application incorporating planning and optimisation functionality.

1.6 Our Methodology

Our methodology is model-driven in that it is centered around a domain model produced by a domain modeller. Once the model has been correctly compiled, a series of automated transformations are applied to produce a control unit for the system which embeds the planning or optimisation algorithm used in the system.

1.6.1 Development Roles in the methodology

There are two development roles enabled by our methodology:

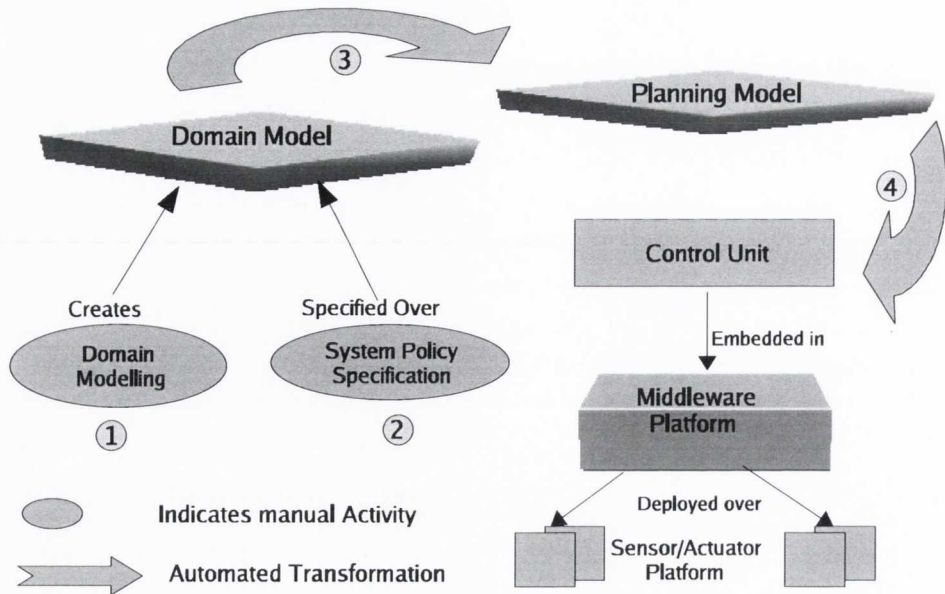


Figure 1.4: Methodology lifecycle phases.

- Domain Modeller.

The domain modeller creates the domain model for an application. Their role is to specify the physical features of the environment in which the application will run, to profile the sensor and actuator infrastructure in the environment and to define the application state space. They are also required to specify any high-level state inference routines required.

- AI Modeller.

The AI modeller should be proficient with planning and optimisation algorithms. Their role is to create an instance of an algorithm type, to define the problem type and the environment properties in which it is suitable for use and define a mapping from the domain model to the algorithm. A mapping must be written for every algorithm added to the library. This mapping is preserved in the form of automated transformations and can be reused in future applications.

The approach is intended to position domain experts at the center of the pervasive systems development process. Provided the algorithm taxonomy is constructed and a mapping defined from the problem type to an algorithm instance, it is sufficient for the domain modeller to specify the model and the transformation algorithms will generate a control unit.

1.6.2 Using the Methodology

The phases involved in using our methodology are shown in Figure 1.4.

1. Domain modelling.

The domain modeller manually constructs a domain model using abstractions that provide a high-level method of specifying the information required to use planning and optimisation algorithms. The domain model stores the available knowledge of the problem structure that comprises the specification of application state space and characteristics of the sensor and actuator infrastructure. The design of the domain model is presented in detail in Chapter 4.

Domain-model elements contain spatial attributes used to specify the physical location and geometry of entities modelled using the domain model.

2. Policy specification.

The system policy is specified by the domain modeller by associating rewards with value ranges for application states. Rewards may also be specified for state-action combinations. These values are later extracted from the policy and used to create a reward model or loss function used by planning and optimisation algorithms.

The spatial attributes of the application states over which the policy is specified are combined to create a policy scope. The policy scope defines a region of the environment in which the application operates.

3. Domain to planning model transformation.

This transformation produces a planning model that encodes the problem structure in a fashion suitable for use by a range of algorithms. A planning model (Harrington & Cahill 2007) is created with information extracted from the domain model. Domain-model abstractions are designed to map directly into the components of the planning model.

4. Planning model to control unit transformation.

This transformation algorithm parses the planning model and produces a control unit for the system that embeds the planning or optimisation algorithm and state inference routines. The transformation supports algorithm selection and seeding with planning-model data. A mapping between planning-model components and algorithm instances is defined initially by an AI modeller and encoded in a transformation procedure that can be used automatically thereafter.

Algorithm selection can be specified by the domain modeller or automated by this transformation. If algorithm selection is automated and the domain modeller is completely unaware of the algorithm properties then it may not be desirable to deploy the control unit without testing it. This motivates a simulation step to verify application behaviour. A simulation activity may be performed in the testing activity shown in Fig. 1.1.

5. Structure and use of control units.

Control units embed state inference and planning or optimisation algorithms used

to determine the behaviour of a pervasive computing application.

For applications using planning algorithms, control units execute plans to govern the actions taken by an application. For applications using optimisation algorithms, control units select from candidate solutions that maximise or minimise properties of an application as specified.

The requirement to use sensor and actuator infrastructure means that the control units must make use of a middleware for access and query operations. When the control units are generated they must make some assumptions about the middleware services available.

1.6.3 Middleware Assumptions

The control units are intended to be deployed using a middleware platform appropriate for pervasive computing systems. This thesis does not directly address the provision of middleware services. The generated control units provide information for sensor/actuator selection and identification, and assume middleware abstractions for discovery and lookup services. Such abstractions are provided by a range of pervasive computing middlewares such as (Román et al. 2002) (Nicklas et al. 2003) (Senart et al. 2006).

The control units assume the following middleware services are available:

- Sensor and actuator discovery and addressing services are available via middleware services. The domain model uses spatial operations implemented with GIS standards to provide information for discovery services. We assume that given a spatial geometry and the name of a sensor datum or actuator action, the middleware can discover and address appropriate sensor and actuator infrastructure in that region.
- Read/write operations to sensors and actuators are available via middleware services. We assume that sensor types used by the control unit can be accessed and read through a synchronous middleware call. The returned value may be correct or erroneous. We assume that each actuator referenced by the control unit is invoked through a middleware call. The actuator itself may fail or perform a different action from that selected by the control unit.

The deployment of control units is not addressed in the thesis. A light-weight middleware service providing the services listed above, has been prototyped to allow the generated control units to be tested. Control units can be deployed and tested on the local machine and use the assumed middleware services to communicate with simulated sensor and actuator middleware.

1.7 Thesis Contributions

We classify the contributions of this thesis from the perspectives of software engineering and automated planning. From a software engineering perspective we make the following

1.8. ROADMAP

contributions:

- A novel methodology for engineering pervasive computing applications that supports the rapid testing of a suite of planning and optimisation algorithms for a given problem specification.
- A domain model for pervasive computing applications that provides a high-level method for specifying information relevant to the use of planning and optimisation algorithms.
- Two transformation algorithms that support the selection and generation of planning and optimisation algorithms and application code. The generated code contains middleware hooks for accessing sensor and actuator infrastructure in the environment.
- A library of algorithms that can be re-used across multiple planning and optimisation problems.

From an automated planning perspective we make the following contributions:

- A taxonomy which encodes information about suitable algorithms for problem types. The second transformation algorithm also supports an encoding of the information required to use these algorithms. This information is captured in the mapping functions written for each algorithm in the library.
- Automated selection of algorithms. If the domain model does not specify an algorithm to use in the problem then the second transformation can automatically select and seed an appropriate algorithm for the application. We believe our approach to automating the identification of suitable algorithms for use in pervasive computing applications to be novel.

1.7.1 Limitations

We do not claim that the automatically selected algorithms are optimal for the problem type. A customised solution hand-crafted by an expert will always be superior to a generated solution. The evaluation case studies show the high impact on performance of tuning algorithm parameters. Automatically selected algorithms execute with default values for algorithm parameters. The use of default parameter values leads to a potential reduction in application performance.

1.8 Roadmap

The remainder of this thesis is structured as follows. Chapter 2 presents an overview of planning and optimisation in pervasive computing environments. Techniques for sensor fusion and state inference are also discussed. Chapter 3 presents research related to the use of MDE methodologies, and planning and optimisation algorithms in pervasive computing

environments. Chapter 4 presents the development process supported by the methodology tool chain. Chapter 4 contains a case-study section describing how the programming model is applied to two application scenarios. Chapter 5 presents the transformation algorithms used to generate planning models and control units. The operation of the transformation algorithms is described using the two case studies introduced in Chapter 4. Chapter 6 presents an empirical evaluation of the impact of the methodology on the development effort associated with developing the two case studies. The performance of the algorithms generated by the automated planning and optimisation component is also analysed. Chapter 7 concludes the thesis with a summary of the work and outlines open issues and possible future extensions to the methodology.

It makes me so happy. To be at the beginning again, knowing almost nothing. ... It's the best possible time to be alive, when almost everything you thought you knew is wrong.

Tom Stoppard (Arcadia)

2

Inference, Planning and Optimisation for Pervasive Computing

This thesis addresses the challenges associated with the development of applications for pervasive computing environments. More specifically it focuses on engineering the inference, planning and optimisation algorithms to be used in these applications.

We define a pervasive computing environment as a region of the physical environment or real-world space that is augmented with sensor and actuator devices and pervasive computing applications as those that execute in such an augmented physical space. This thesis focuses on those pervasive computing applications that perform some combination of the following tasks:

- Applications that use sensor data to infer values for states of the environment that are relevant to the application.
- Applications that use actuator infrastructure to take actions in accordance with some user-specified objectives.
- Applications that use inferred state information to optimise application states.

These tasks are referred to respectively as inference, planning and optimisation and a detailed definition is provided in the next section.

This chapter positions these tasks in the context of pervasive computing. Characteristics of pervasive computing environments that impact on these tasks are described. Subsequently a set of algorithms representative of approaches to these tasks is presented. The treatment of these algorithms is intended to show when they are suitable to be used and what information must be provided to use them.

2.1 Overview of Planning and Optimisation Tasks

We address the following tasks in the context of pervasive computing.

- State inference

This is the task of determining the true state of the system given variable-quality sensor data. State inference may be performed at a single instant or else may be modelled as an inference process over time. State inference involves the sub-tasks of sensor identification, selection and sensor fusion.

- Planning

Planning is the task of choosing an action to carry out in order to achieve a goal. Planning problems can be divided into classical or real-world problems. Classical planning problems are modelled using assumptions that typically include restricted state transition systems and full observability. Real-world planning problems are considerably more challenging due to the presence of uncertainty and dynamism in the environment and are representative of the types of problems encountered in pervasive computing environments. Planning problems may be modelled as single or sequential decision problems.

- Optimisation

Optimisation is the task of maximising or minimising a property of a system and can be expressed as finding the value(s) of a vector $\theta \in \Theta$ representing some system property that minimises a loss function. An example would be, the selection of traffic light settings in a UTC system to minimise the waiting time of vehicles at the junction.

Optimisation can be modelled as a search problem and algorithms divided into deterministic and heuristic approaches. A deterministic or exhaustive search will find the optimal solution but may not be feasible given the size of the search space or the time constraints for finding a solution. Heuristic algorithms can be applied to a wide variety of problems of varied combinatorial structure to find useful if not optimal results. Optimisation problems can be considered as a subset of planning and acting problems but result in different system architectures and hence are classified independently in this chapter.

Fig. 2.1 shows a taxonomy of algorithms that can be used for these three tasks. The algorithms are related to problem types and domain properties. The domain properties listed in Fig 2.1 are uncertainty, dynamism and complexity and are described in following sections.

This chapter also examines a selection of algorithms for performing the tasks of state inference, optimisation and planning. The selection of algorithms shown in Fig. 2.1 is not intended to prescribe the only correct approaches for such problems. There is clearly a large set of candidate algorithms that can be applied across problems. The selection shown

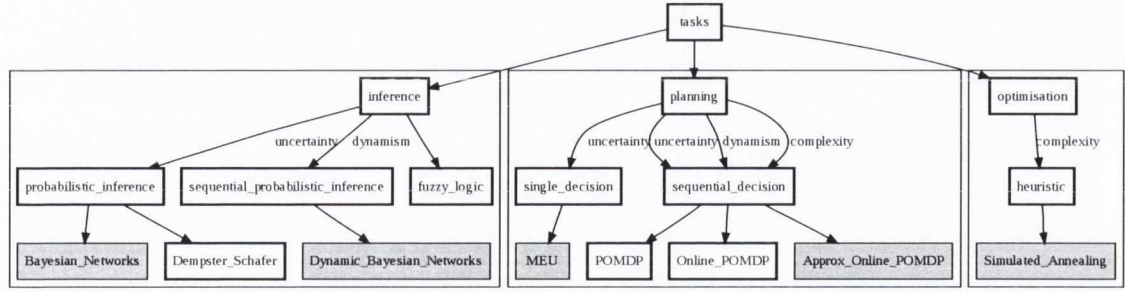


Figure 2.1: Planning and optimisation problems and algorithms.

in Fig. 2.1 is informed by reviewing the literature and by implementing and testing many of the algorithms described here. Almost all the algorithms discussed here have been implemented in the thesis evaluation scenarios.

The treatment of the algorithms in this chapter focuses on how they would be used in pervasive computing environments. An overview of the algorithms is presented, including practical issues relating to specifying parameters, and computational overhead.

2.2 Characteristics of Pervasive Computing Environments

In this section we introduce some characteristics of pervasive computing domains and applications with the intention of identifying difficulties faced by application developers. To clarify the discussion that follows we introduce a semi-formal model suitable for modelling real-world environments (Ghallab et al. 2004).

2.2.1 Modelling Real-world Environments

The following model can be used to model real-world environments, $\Sigma = (S, A, T, O, R)$ where:

- $S = \{s_1, s_2, \dots\}$ is the set of system states;
- $A = \{a_1, a_2, \dots\}$ is the set of actions;
- $T(s, a, s')$ represents a stochastic state-transition system that gives the probability $P(s'|s, a)$ of moving to state s' if the action a is performed in state s .
- $O = \{o_1, o_2, \dots\}$ is the set of observations that are produced by the sensor infrastructure in the region. An observation or sensor model function $O(s', a, o)$ gives the probability $P(o|a, s')$ of observing o if action a is performed and the resulting state is s' . This function describes how sensors are affected by the actual state of the system and is often referred to as a likelihood function.
- $R(s)$ is the reward function and $R(s, a, s')$ represents the immediate reward the agent will get if it performs action a while in state s and moves to state s' .

In the discussion that follows we will refer to the elements of this model to show how the characteristics of pervasive computing environments impact on planning and optimisation.

2.2.2 Uncertainty in Pervasive Computing Environments

Uncertainty can arise from a number of sources. The dependence on sensor data results in partial observability of the system state due to the possibility of error in sensor data readings or error in the state inference routines used to combine sensor data. The use of actuator platforms to effect change in the environment is also a source of uncertainty due to possible actuator failure.

The presence of uncertainty impacts on the design and execution of pervasive computing applications. If system state is highly uncertain then it is difficult to determine the correct action to take and the possibility of actuator failure may require the regular monitoring of system state and corrective replanning.

An environment is fully observable if the complete state of the environment at each point in time is available to the application. In the case of pervasive computing environments data on system state is typically inferred from sensor data with an attendant possibility of error in the sensor readings and hence the decision making process is unavoidably at a distance from the true state of the system.

This uncertainty between the true and inferred system state can be modelled by using sensor models that associate a probability with a mapping from sensor observations to state values. Sensor models are usually specified at design time prior to the deployment and execution of applications. If applications are deployed over a large area in pervasive computing environments there may be incomplete or non-uniform sensor coverage throughout the region. This non-uniformity may be due to variations in sensor coverage and quality due to the use of different sensor types and deployment densities. Non-uniformity in sensor data coupled with the possibility of sensor mobility motivates the need for run-time sensor profiling to correctly determine confidence levels to associate with sensor data used for state-inference.

The possibility of actuator failure is a source of uncertainty when specifying state-transition systems. An actuator model can be used to capture the reliability of the actions available to the system and can relate the execution of available actions to the probability of a change in the system state.

2.2.3 Dynamism in Pervasive Computing Environments

A dynamic environment is one in which the state of the environment may change independently of the actions taken by the application. The presence of uncontrolled events and resultant state transitions is a source of dynamism in the system. Such sources of dynamism abound in pervasive computing environments. Consider a domain such as Urban Traffic control. Road traffic accidents, wide fluctuations in traffic demand, extreme

2.2. CHARACTERISTICS OF PERVASIVE COMPUTING ENVIRONMENTS

weather conditions, etc, are all sources of uncontrollable uncertainty in an urban traffic control domain.

The presence of dynamism has implications for the timing of decision making and planning activities for applications operating in such environments. If an environment is dynamic then online planning algorithms may need to be used to respond to unplanned or unforeseen system state changes.

Decision making and planning may be online or offline in response to dynamism in the environment. If action selection and execution are interleaved at run-time then the algorithm is said to be online and can sense and respond to changes in the environment. Conversely an offline algorithm is one in which a policy governing action selection is constructed prior to the time in which the application begin executing.

Accessing sensor data will usually come at a cost and if a pervasive computing environment is highly dynamic there is no point in wasting computing effort in planning a policy governing action selection for a long-horizon.

2.2.4 Complexity in Pervasive Computing Environments

Complexity in pervasive computing applications may arise due to issues such as state space size, the degree of interconnectedness among system elements and partially-understood or stochastic state-transition systems.

For applications modelled as complexity problems, the complexity of a problem can impact on the selection of an appropriate algorithm. If the complexity of a problem is such that a deterministic or brute force solution is be infeasible given computational or time constraints, then a heuristic approach may provide useful though not optimal results.

Complexity measures may be roughly divided into those that measure the complexity of an algorithm versus those that measure the complexity of a problem.

The complexity of an algorithm can be expressed as a function of its running time or space requirement and written using big O notation. For a pervasive computing task it may be more relevant to classify the complexity of the given problem rather than that of an algorithm. Computational complexity theory analyses problems rather than algorithms and is concerned with the question for which decision problems do efficient algorithms exist.

Traditional computational complexity analyses focus on decision problems in general and on language recognition problems in particular (Ghallab et al. 2004, 551). Complexity theory is based on the standard Turing model of computation and measures the efficiency of decision problems by measuring the time and space resources of a Turing machine required to solve a decision problem of finite length. A key definition of complexity theory is that *feasible* or *tractable* means polynomial time (Rudich Steven 2004, 20).

The class of decision problems that have a worst-case running time of $O(n^k)$ for some k is called P. These problems can be solved deterministically in polynomial time and are hence considered to be the 'easiest' or most tractable problems.

2.3. STATE INFERENCE

Another important class of problems is NP, the class of non-deterministic polynomial problems. This is the class of problems for which a solution can be verified but not generated quickly, i.e., polynomially. The hardest set of problems in this class are NP Complete problems. Establishing that a problem is NP Complete is useful as heuristic approaches should then be adopted in place of any further analysis of the problem.

The use of complexity analysis in pervasive computing is complicated by the fact that such analysis is applied to stationary and finite decision problems. In order to discuss the computational complexity of planning problems, the problems must be reformulated as language-recognition problems.

This approach may not be useful when dealing with many of the problems faced by application developers in pervasive computing environments, many of which are dynamic and continuous in nature.

More fundamentally though pervasive computing applications even if formulated as planning problems will typically operate in continuous noisy environments. Complexity theory deals with finite or bounded decision problems with observable and enumerable inputs. (Spall 2003, 268) recognises this difficulty and states that any static measure of problem difficulty will always be severely limited.

Appendix A contains a more detailed discussion of complexity metrics and their applicability to pervasive computing applications.

2.3 State Inference

State inference is the process of determining the state of a system given observations produced by the sensor infrastructure. A pervasive computing system relies on sensor infrastructure to gain knowledge about the system state. Sensor data is not a perfect reflection of system state as errors can arise due to partial coverage and sensor error. Hence there is an unavoidable degree of uncertainty associated with the state inference process.

There are theoretical and practical issues to be addressed to support state inference. The theoretical issues are:

- how to represent uncertain state information.
- how to use uncertain data from sensors to update existing state information.

The practical issues to be address when supporting state inference are:

- how to model sensors and represent sensor data.
- how to associate deployed sensors with application state-space values.
- how to access sensor data.

This section presents techniques for representing and updating uncertain state information. Probability theory, Bayesian Inference and Dempster-Shafer theory are examined in detail. Section 2.3.5 presents an overview of sensor fusion, addressing practical issues relating to associating sensor data with states and accessing sensor data.

2.3.1 Probability Theory and Bayes Rule

Probability theory provides a language to make propositions about random variables reflecting states of the environment. A random variable can be boolean, discrete or continuous. An example of a discrete random variable would be one representing the weather that can take on particular values such as sunny, rainy or misty. An example of a continuous random variable would be the temperature in a room.

An application's knowledge of a pervasive computing environment can at best provide only a degree of belief. Probability provides a way of summarising this uncertainty. Probabilities between 0 and 1 correspond to different degrees of belief in the truth of some statement about an aspect of the real-world with the value 0 representing false or no belief and 1 representing truth or full belief. A probability of 0.8 in a proposition should not be interpreted as implying that the proposition is 80% true, rather it implies an 80% degree of belief in the proposition.

An application's beliefs will depend on the sensed information that the application has received to date. This sensed information constitutes evidence on which probability assertions are based. Receiving new evidence should result in the degree of belief of a proposition being updated. Before the evidence is received the system has a prior or unconditional probability in a proposition. After some evidence has been received and processed, the system has a posterior or conditional probability in a proposition.

Probability is a finite measure. Being finite means that it has a largest possible value which is 1. Being a measure means that it is a function, or a mechanism that takes inputs such as an event x , and converts it into output, probability $P(x)$. The range of probability or the set of possible outputs, is the interval $[0, 1]$. The domain of probability - its set of possible inputs, is not necessarily a set of numbers. We can also consider the probability of events. Thus the domain of probability is a set of events.

Axioms of Probability

Kolmogorov's axioms provide the foundation of probability theory:

1. All probabilities are between 0 and 1.
For any proposition a : $0 \leq P(a) \leq 1$.
2. True and false propositions have probabilities 1 and 0 respectively.
 $P(\text{True}) = 1$ $P(\text{False}) = 0$.
3. The probability of a disjunction/union is given by:
 $P(a \cup b) = P(a) + P(b) - P(a \cap b)$.

From these axioms, the following can be derived:

2.3. STATE INFERENCE

- $P(\neg a) = 1 - P(a)$ where $\neg a$ means “not” a .
- If a proposition, a , consists of a number of mutually exclusive or disjoint atomic events e_i , where $(1 \leq i \leq N)$ and $a = \bigcup_{i=1}^N e_i$, $e_i \cap e_j = \emptyset \quad \forall \quad 1 \leq i, j \leq N$ then

$$P(a) = \sum_{i=1}^N P(e_i)$$

This states that the probability of a proposition is equal to the sum of the probabilities of the atomic events in which that proposition holds.

Conditional Probability

Consider a domain containing two random variables. Initially we have a set of prior probabilities $P(a)$ and $P(b)$. If the events a and b , occur independently of each other then the joint probability of a and b can be written as: $P(a) \cap P(b) = P(a) \times P(b)$

The occurrence of an event may be influenced by the occurrence of another event. If some evidence is received on variable b then prior probabilities are no longer applicable and instead conditional or posterior probabilities become relevant.

The conditional probability of event a given event b is the probability that a occurs when event b is known to have occurred. It is written as $P(a|b)$ and defined as:

$$P(a|b) = \frac{P(a \cap b)}{P(b)} \tag{2.1}$$

and holds whenever $P(b) > 0$

Using conditional probabilities this can be written in a form known as the Product rule or the fundamental rule of probability:

$$P(a \cap b) = P(a|b)P(b) \tag{2.2}$$

This rule gives the joint probability for events that are dependent.

Consider the following simple example of two manufacturers (a & b) of widgets.

	Manufacturer a	Manufacturer b	Total
Number with a fault	200	300	500
Number with no fault	29,800	19,700	49,500
Total	30,000	20,000	50,000

Table 2.1: Widget manufacturers.

Table 2.1 represents the frequency that a widget develops a fault within 1 year of purchase. Assume that what happened in the past is an indicator of what might happen in the future. The table can be used to calculate the probability of a future event such as a fault developing within 1 year of purchase.

2.3. STATE INFERENCE

Dividing each entry in the table by 50,000 gives the joint probability distribution, which can be used to calculate probabilities for all propositions in the domain

	Manufacturer a	Manufacturer b	Total
Fault = True	0.004	0.006	0.01
Fault = False	0.596	0.394	0.99
Total	0.6	0.4	1.0

Table 2.2: Full joint probability distribution for $P(\text{Manufacturer}, \text{Fault})$.

Assume we are interested in the probability of a machine developing a fault given that the purchaser bought from a particular manufacturer. We identify the atomic events in which the proposition holds and sum their probabilities $P(a) = \sum_{i=1}^N P(e_i)$.

This probability can be written as $P(\text{Fault} = \text{true} | \text{Manufacturer} = a)$ or $P(\text{Fault} | a)$. Substituting into the Product Rule formula gives:

$$\begin{aligned}
 P(\text{Fault} = t | \text{Manufacturer} = a) &= \frac{P(\text{Fault} = t \cap \text{Manufacturer} = a)}{P(\text{Manufacturer} = a)} \\
 &= \frac{0.004}{0.6} = 0.0067 \\
 P(\text{Fault} = t | \text{Manufacturer} = b) &= \frac{P(\text{Fault} = t \cap \text{Manufacturer} = b)}{P(\text{Manufacturer} = b)} \\
 &= \frac{0.006}{0.4} = 0.015
 \end{aligned}$$

Joint and Marginal Probabilities

Summing a row or a column in the joint probability table allows the calculation of a marginal probability. $P(\text{Fault}) = P(\text{Fault} \cap a) + P(\text{Fault} \cap b)$. This allows the manufacturer effect to be “removed”. That is, a distribution over Fault can be obtained by summing out all the other variables from the joint distribution that contain Fault.

Marginalization can also be applied to sets of variables such as Y and Z such that the marginal or unconditional probability $P(Y)$ is: $\sum_z P(Y, z)$, and can also be performed using conditional probabilities instead of joint probabilities, by using the product rule: $P(Y) = \sum_z P(Y|z)P(z)$.

Consider a domain with n binary variables, $\{y_1, y_2, \dots, y_n\}$, the joint probability distribution is written $P(y_1, y_2, \dots, y_n)$. Calculations become more complicated as the number of variables increases. With two binary variables, there are four entries in the joint probability distribution table. This is because the two variables take on two possible values $2^2 = 4$. For n binary variables, there are 2^n entries in the table.

The full joint probability distribution specifies the probability of every combination of events and is therefore a complete specification of the uncertainty in a system. A particular value in the joint distribution is given as $P(x_1 = x_1 \cap x_2 = x_2 \cap \dots \cap x_n = x_n)$, which can be written as $P(x_1, x_2, \dots, x_n)$.

The size of the table grows exponentially with the number of variables. Using a table

2.3. STATE INFERENCE

to represent the joint probability distribution is not the most efficient representation. It could be considered as a brute force approach but is not a practical tool for supporting medium or large sized inference tasks.

The use of causal or model-based knowledge allows large-scale probabilistic systems to be feasible in the real world. The knowledge used to make larger scale systems feasible are independence and conditional independence among environment variables.

Independence and Conditional Independence

Two variables, a and b , are said to be independent if: $P(a \cap b) = P(a)P(b)$. As the variables a and b do not influence each other: $P(a|b) = P(a)$ or $P(b|a) = P(b)$.

If there are 4 binary variables, $\{y_1, y_2, y_3, y_4\}$ with y_4 is independent of the others, then the joint probability distribution can be written

$$P(y_1, y_2, y_3, y_4) = P(y_1, y_2, y_3)P(y_4)$$

So, instead of having a table containing $16(=2^4)$ entries, there are two tables. The first table has three variables and eight entries, while the second table has one variable and two entries. Thus, the space needed to represent the joint probability distribution is reduced to: $2^3 + 2^1 = 10$.

Independence is a symmetrical relationship and assertions are usually based on knowledge of the domain. They can dramatically reduce the amount of information necessary to specify joint distributions across sets of variables.

Two variables, a and b may be independent given evidence about another variable, c . This is a form of conditional independence and can be written: $P(a, b|c) = P(a|c)P(b|c)$.

In such a situation, although a and b are not independent, if the value of c becomes *known* for certain, then any evidence about b cannot change the belief about a . In such a case: $P(a|c) = P(a|b, c)$

Let variables a and b represent the toss of a coin and c represents whether the coin is biased towards "Heads" or not. If c is unknown and if it is observed that a is "Heads", then this increases the likelihood that b will also be "Heads". However, if c is known, then the outcome of a does not influence the belief in the outcome of b .

In this case a is *screened off* from b by c . This can be written as: $(a \perp\!\!\!\perp b|c) \equiv P(b|a, c) = P(b|c)$

Chain Rule

From the product rule for two events, a general form of the chain rule can be derived to calculate the full joint probability distribution as a product of conditional probabilities.

$$\begin{aligned} P(x_1, x_2, \dots, x_n) &= P(x_1) \times P(x_2|x_1) \times P(x_3|x_1, x_2) \dots \times P(x_n|x_1, \dots, x_{n-1}) \\ &= \prod_i P(x_i|x_1, \dots, x_{i-1}) \end{aligned}$$

Applying the chain rule to large strongly coupled domains becomes intractable as the

2.3. STATE INFERENCE

number of events or variables increases because the computation is exponential in the number of random variables to be solved for. However if the complete set of variables can be divided into independent subsets, then the full joint can be factored into separate joint distributions on those subsets. In large domains with weakly coupled subsets of dependent variables this can significantly reduce the effort required to calculate joint probability distributions.

If $P(y_1, y_2, y_3, \dots, y_n)$ is a joint probability distribution over variables $\{y_1, y_2, y_3, \dots, y_n\}$ the Chain Rule allows the joint probability distribution to be factored into a product of conditional probabilities:

$$P(y_1, y_2, y_3, \dots, y_n) = P(y_1|y_2, y_3, \dots, y_n) \times P(y_2|y_3, \dots, y_n) \times P(y_{n-1}|y_n) \times P(y_n)$$

This is useful because conditional independence relationships are more common than independence relationships. The chain rule can be stated formally as:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i|x_1, \dots, x_{i-1}) \quad (2.3)$$

Using the Chain Rule helps to reduce the number of probabilities needed to represent the joint probability distribution. The question now arises how evidence (or new sensor data) is incorporated to update the joint probability distribution.

2.3.1.1 Bayes Rule and Belief Updating

Bayes rule provides a general mechanism by which new evidence or sensor data can be combined with existing beliefs or probability distributions over the current system state.

Bayes rule can be derived from the associative property of the product rule:

$$P(a \cap b) = P(a|b)P(b) \text{ and } P(a \cap b) = P(b|a)P(a)$$

Equating the two right-hand sides and dividing by $P(a)$ gives:

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \quad (2.4)$$

Bayes rule can also be stated in terms of hypothesis testing as providing a measure of the probability of a hypothesis h conditioned upon some evidence e , that is equal to its likelihood $P(e|h)$ times its probability prior to any evidence $P(h)$, normalized by dividing by $P(e)$. This is written as:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e)}$$

Having applied Bayes rule to obtain $P(h|e)$ - this is adopted as the posterior or updated degree of belief in the hypothesis h - or $\text{Bel}(h) = P(h|e)$. Adopting the posterior probability as the new belief in a hypothesis is referred to as conditionalisation. There are two basic

2.3. STATE INFERENCE

assumptions of conditionalisation that are relevant when using Bayes theorem (Korb & Nicholson 2004). These are:

- There must exist joint priors over the hypothesis and evidence spaces. Without a joint prior Bayes theorem cannot be used consistently.
- The evidence conditioned upon is all of, and the only evidence learned. This is called the total evidence condition and cannot always be guaranteed. This is especially the case in pervasive computing environments where sensor data may be received from subsets of available nodes due to power or communication constraints.

The following state inference example illustrates how Bayes rule can be used to update the degree of belief or probability of a proposition given some evidence.

Consider an air-traffic control scenario (Bertsekas & Tsitsiklis 2002, 31). If an aircraft is present in a certain area, a radar correctly registers its presence with probability 0.99. If it is not present, the radar can falsely register an aircraft presence with probability 0.10. Assume that the prior probability of an aircraft being present is 0.05. If the radar sensor now registers a presence Bayes rule can incorporate this piece of evidence and calculate the probability that an aircraft is actually present.

Let a equal the event that an aircraft is present, and let b equal the event that a radar signals an alarm. A domain expert provides the following probabilities:

$$P(a) = 0.05, \quad P(b|a) = 0.99, \quad P(b|\neg a) = 0.10$$

Bayes rule can be used to calculate the updated belief in the event $P(a|b)$, i.e., that an aircraft is present given that the radar has registered its presence.

$$P(a|b) = \frac{P(b|a) \times P(a)}{P(b)} \tag{1}$$

$$= \frac{P(b|a) \times P(a)}{P(b|a) \times P(a) + P(b|\neg a) \times P(\neg a)} \tag{2}$$

$$= \frac{0.99 \times 0.05}{0.99 \times 0.05 + 0.1 \times 0.95} = 0.342 \tag{3}$$

Step 1, is obtained from Bayes rule. The denominator in step 2 is the marginal probability or likelihood of the evidence, i.e., the radar reading. Step 3, gives $\text{Bel}(a)$, the posterior probability of the aircraft being present by combining the prior probability of an aircraft being present and the radar sensor data.

Bayes rule can therefore be used as a sensor fusion technique and it is straightforward to use it to combine data from multiple sensors. This rule underlies all modern AI systems for probabilistic inference (Russell & Norvig 2003, 479).

2.3.2 Bayesian Inference and Sensor Model

Pervasive computing systems use sensor data to infer system state. Bayes rule allows sensor data to be combined and used as evidence to perform inference about propositions or system states. Inference or belief updating is the process by which beliefs are conditioned upon new evidence or information about the value of a variable. In this sections we present a model for representing and reasoning about sensor data.

A sensor model should provide a coherent description of the sensors ability to extract information from its surroundings. The sensor model presented here is similar to that in (Mitchell 2007, 24) and uses the following notation:

- Θ is the state in which the system is interested
- $y_i, i \in (y_1, y_2, \dots, y_N)$, denotes a vector of N raw sensor measurements or pieces of evidence which are directly observed, while the variable of interest Θ is not directly observed but instead is inferred.
- I represents any relevant background information.

Using Bayesian inference the task of inferring Θ is equivalent to estimating the posterior probability $P(\Theta = \theta|y, I)$, where θ represents the true value of the variable of interest Θ . Fig. 2.2 (Mitchell 2007, 25) shows the use of a sensor model as an input to the Bayesian inference process. This figure shows four inputs to the Bayesian inference routines. These components are:

- Data y or $f(y|\theta)$ The input data may be sensor data readings y or data generated as a function of sensor data $f(y|\theta)$.
- $P(\theta|I)$ This is a prior probability function which characterises current beliefs about θ , the variable of interest. In the absence of any further information this distribution may be modelled using historical data or typical values for θ .
- $P(y|\theta, I)$ This is a likelihood function which describes how the values generated as a function of the sensor measurements depend on the true value θ and the background information I . The likelihood function provides a method of estimating how likely the various possible values of θ are given that the sensor data has produced the evidence y .

If a sensor reading y is directly related to the variable θ of interest, then the inference task is simplified. A direct mapping from a sensor reading to a variable of interest results in a simple likelihood function. However there are times where more complex models must be built to combine sensor data. These models combine domain-specific knowledge about the structural relationship between variables in a system and their interdependencies and causal relations.

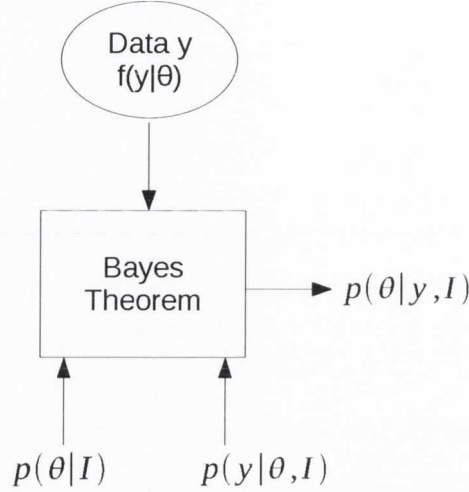


Figure 2.2: Inputs to a Bayesian inference process.

The output of the Bayesian inference routine is a posterior distribution $P(\theta|y, I)$ that summarises the updated knowledge about θ . The posterior distribution is calculated from applying Bayes rule as follows:

$$P(\theta|y, I) = \frac{\overbrace{P(y|\theta, I)}^{\text{likelihood}} \overbrace{P(\theta|I)}^{\text{prior}}}{\underbrace{m(y)}_{\text{evidence}}} \quad (2.5)$$

The denominator $m(y)$ is a normalizing constant and is the unconditional or marginal distribution of data y . It can be considered as the likelihood of the evidence or sensor data and written as:

$$\frac{P(y|\theta)P(\theta)}{\int P(y|\theta)P(\theta)d\theta}$$

When there is more than one unknown parameter, e.g. $\theta = (\theta_1, \theta_2)$, and only one component is relevant, say θ_1 , the remaining unknown quantities are removed by integration:

$$P(\theta_1|y) = \frac{\int P(y|\theta_1, \theta_2)P(\theta_1, \theta_2)}{\int \int P(y|\theta_1, \theta_2)P(\theta_1, \theta_2)d\theta_1, d\theta_2} \quad (2.6)$$

This is equivalent to the marginalisation operation for discrete variables discussed earlier.

The application of Bayes Rule gets more complicated as the number of variables increase. The most challenging aspects are identified in (Mitchell 2007, 119) as:

- the development of a model $P(y|\theta, I)P(\theta, I)$, which must effectively capture the key features of the underlying problem;
- the necessary computation required for deriving the posterior distribution. In general

2.3. STATE INFERENCE

both exact and approximate inference are NP hard problems (Korb & Nicholson 2004, 53).

A popular approach to tackling these two problems is to build graphical models for the purpose of inference. Jordan (1999) describes graphical models as:

a marriage between probability theory and graph theory. They provide a natural tool for dealing with two problems that occur throughout applied mathematics and engineering - uncertainty and complexity ... Fundamental to the idea of a graphical model is the notion of modularity - a complex system is built by combining simpler parts. Probability theory provides the glue whereby the parts are combined, ensuring that the system as a whole is consistent, and providing ways to interface models to data. The graph theoretic side of graphical models provides both an intuitively appealing interface by which humans can model highly-interacting sets of variables as well as a data structure that lends itself to the design of efficient general-purpose algorithms.

Bayesian networks are one type of graphical model that address the complexity of inference computations through exploiting independence relationships between variables.

2.3.3 Bayesian Networks

A Bayesian network is a graphical model that allows knowledge about uncertain domains to be represented and reasoned about. The nodes in a Bayesian network represent random variables from the domain. The set of edges connect pairs of nodes and represent causal dependencies between variables. The structure of the network reflects the conditional dependencies that exist between variables. A Bayesian network (BN) is a Directed Acyclic Graph (DAG), i.e. there are no cycles in the network.

If the domain contains weakly coupled subsets of variables this is reflected in a smaller number of edges connecting variables and in reduced conditional probability table sizes. The decomposition of large probabilistic domains into weakly connected subsets via conditional independence has been cited as one of the most important developments in the recent history of AI (Russell & Norvig 2003, 482).

Bayesian network node types may be: boolean nodes representing variables taking binary values true and false; discrete nodes representing ordinal valued variables, e.g., low, medium, high; integral or discrete valued variables or continuous functions.

If there is a directed link from node X to node Y , then X is said to be a parent of Y . Each node, X_i , has a conditional probability distribution, $p(X_i | \text{Parents}(X_i))$. The conditional probability distribution for each node given all the possible combinations of values of its parent nodes must be specified. The size of the conditional probability distribution is exponential in the number of parents. A node with n parents requires a conditional probability table with 2^{n+1} probabilities. The probability represents the effect of the parents on the variable.

2.3. STATE INFERENCE

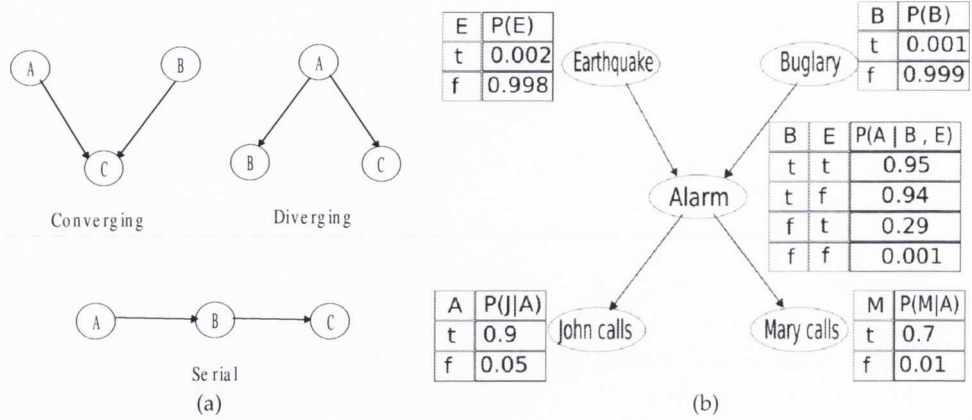


Figure 2.3: Bayesian Network Sample and Connections: (a) D-Separation; (b) Earthquake/Alarm Example

Constructing a BN to ensure that directed links go from causes to effects requires a certain level of domain expertise. Different node orders may result in different network structures, but they should all represent the same joint probability distribution. However they will vary in terms of compactness. The optimal order is to add the root nodes first, then the nodes they influence directly and continue until leaf nodes are reached.

The joint probability distribution for a BN, with variables $\{X_1, X_2, \dots, X_n\}$ is given by:

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^N P(X_i | \text{parents}(X_i))$$

This is equivalent to the Chain Rule provided that:

$$P(X_i | \text{parents}(X_i)) = P(X_i | X_1, \dots, X_{i-2}, X_{i-1})$$

In other words, this requires that each variable is conditionally independent of its predecessors given its parents.

Independence in Bayesian Networks

The topological criteria used to determine conditional independence is called d-separation. There are three types of connections between variables in a BN. The connection types are explained with reference to a sample Bayesian network relating to an earthquake/house alarm scenario shown in Fig. 2.3b (Korb & Nicholson 2004, 44).

- A **Converging Connection** represents a situation where a node C has two causes: A and B. If there is no evidence about the state of C, then A and B are independent and knowledge of A has no influence on B and vice versa. Variables A and B remain independent unless evidence about C is received. A and B are said to be *d-separated*. This can be written as : $P(A|B \cap C) \neq P(A|B) \equiv \neg(A \perp\!\!\!\perp B|C)$

In the burglary/earthquake example, the earthquake and burglary variables are in-

dependent. However, if evidence is received that the alarm has sounded, then these events become dependent. If evidence that an earthquake has occurred is received, then belief that a burglary has also occurred should be reduced.

- A **Diverging Connection** represents the situation where a node A has two effects: B and C . Influence can pass between the children of A provided the state of A is unknown, hence B and C are dependent but become independent when evidence about A is received. B and C are said to be *d-separated* given A .

This conditional dependence can be written as: $P(C|B \cap A) = P(C|A)$ and the *d-separation* of C and B given A expressed as: $B \perp\!\!\!\perp C|A$

In the burglary/earthquake example, the `MaryCalls` and `JohnCalls` variables are independent given knowledge of the alarm status. If the state of the alarm is not known and Mary has called, this should increase the belief that John will call. If however it is known that the alarm has sounded, then whether Mary called or not should not influence the belief of whether John will call or not.

- **Serial Connection**, influence can pass between A and C through B provided that the state of B is unknown. A and C are dependent but become independent when evidence about B is received. A and C are said to be *d-separated* given B .

This conditional dependence can be written as: $P(C|A \cap B) = P(C|B)$ and the *d-separation* of A and C given B , expressed as: $A \perp\!\!\!\perp C|B$

In the burglary/earthquake example, knowledge about the state of the variable `MaryCalls` should influence the belief of a burglary or an earthquake having occurred assuming that the state of the alarm is not known by some other means.

This conditional independence or *d-separation* relationship also applies to sets of nodes. A set of nodes E can *d-separate* two other sets of nodes X and Y if every path from a node in X to a node in Y is *d-separated* given E . Establishing *d-separation* properties between sets of nodes reduces the size of the conditional probability tables that must be represented. In general, if there are N binary nodes or variables in a system, the full joint would require $O(2^n)$ space to represent, but the factored form would require $O(n2^k)$ space to represent, where k is the maximum fan-in of a node.

2.3.3.1 Inference in Bayesian Networks

The basic task for a probabilistic inference system is to compute the posterior probability distribution for a set of query variables given some observed event - that is some assignment of values to a set of evidence variables. Let X denote a query variable; E denote the set of evidence variables $e_1, \dots, e_m$, and e be a particular observed event; and Y denote the non-evidence variables $y_1, \dots, y_l$. The complete set of variables is then $X \cup E \cup Y$.

2.3. STATE INFERENCE

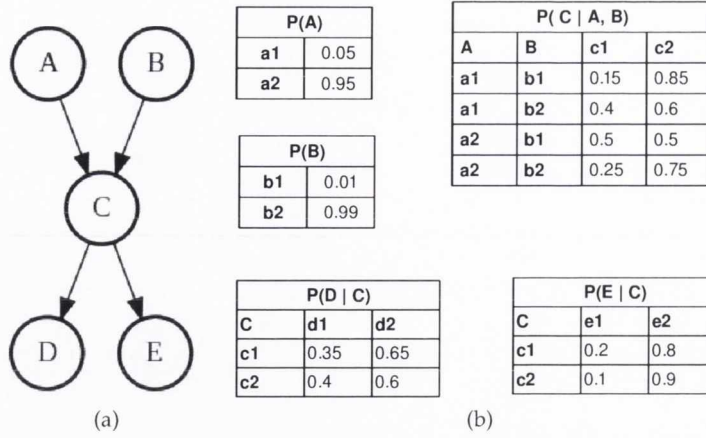


Figure 2.4: Bayesian Network sample and CPTs: (a) Sample Bayesian network; (b) Sample conditional probability tables.

A typical query asks for the posterior probability distribution of a query variable given some assignment of values to a set of evidence nodes. This can be calculated by marginalising over or summing out the non-evidence nodes:

$$P(X|e) = \alpha \sum_y P(X, e, y) \quad (2.7)$$

A Bayesian Network encodes:

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^N P(X_i | \text{parents}(X_i))$$

Therefore the terms $P(x, e, y)$ in the joint distribution can be written as products of conditional probabilities from the network and a query can be answered by computing sums of products of conditional probabilities from the network.

Consider the sample Bayesian network in Fig. 2.4. Assume that sensor data is received for the value of nodes D and E such that $D = d_2$ and $E = e_2$. Assume that we are interested in the probability that the value of node A will take the value a_2 given the observed sensor data. This can be calculated using Eqn. 2.7 :

$$P(A = a_2 | d = d_2, e = e_2) = \alpha \sum_B \sum_C P(b)P(a)P(c|a, b)P(d|c)P(e|c) \quad (1)$$

$$= \alpha P(a) \sum_B P(b) \sum_C P(c|a, b)P(d|c)P(e|c) \quad (2)$$

Step 1 shows that B and C, the non-evidence nodes are marginalised out by summation.

2.3. STATE INFERENCE

Step 2 shows a simplification of the algebra. The result is calculated using summation as:

$$\begin{aligned}
& P(a) \times P(b) \times P(c|b, a) \times P(d|c) \times P(e|c) + \\
& P(a) \times P(b) \times P(\neg c|b, a) \times P(d|\neg c) \times P(e|\neg c) + \\
& P(a) \times P(\neg b) \times P(c|\neg b, a) \times P(d|c) \times P(e|c) + \\
& P(a) \times P(\neg b) \times P(\neg c|\neg b, a) \times P(d|\neg c) \times P(e|\neg c) \\
& = 0.95 \times 0.99 \times 0.75 \times 0.6 \times 0.9 + \\
& \quad 0.95 \times 0.99 \times 0.25 \times 0.65 \times 0.8 + \\
& \quad 0.95 \times 0.01 \times 0.5 \times 0.6 \times 0.9 + \\
& \quad 0.95 \times 0.01 \times 0.5 \times 0.65 \times 0.8 = 0.5082025
\end{aligned}$$

To obtain a normalised result we must calculate α - the marginal likelihood of the evidence. Similarly to above we can calculate $P(A = a_1|d = d_2, e = e_2) = 0.0266025$. $\alpha = (0.5082025 + 0.0266025)^{-1} = 1.869840409$, which leads to a normalised posterior probability $P(A|d = d_2, e = e_2) = \{a_1, a_2\} = (0.0498, 0.9502)$

2.3.3.2 Inference with Uncertain Evidence

The discussion above assumed that any sensor evidence is a direct observation of the value of a variable and results in the belief for that node being set to 1 for that value and 0 for all other values. In the example above it was assumed that the evidence for $D = d_2$ and $E = e_2$ is true with probability 1. However this is an atypical case in a pervasive computing environment where evidence for observed nodes will usually come from sensors with possible errors in readings.

Sensor evidence with an associated uncertainty is referred to as soft or virtual evidence. The semantics of belief updating given virtual evidence are not clear (Jensen & Nielsen 2007, 40). Two techniques for representing uncertain evidence have been proposed: soft evidence represented as probability distributions and virtual evidence represented as likelihood ratios. The soft evidence approach requires uncertain evidence to be specified in terms of the effect it has on beliefs once accepted, while the virtual evidence approach requires the uncertain evidence to be specified in terms of its strength only (Chan & Darwiche 2005). Chan & Darwiche (2005) also show that while these techniques differ in how they represent uncertain evidence they both use the principle of probability kinematics to revise beliefs.

Soft Evidence and Jeffrey's Rule.

Consider the problem of revising a probability distribution P given uncertain evidence relating to a set of mutually exclusive and exhaustive events $(\gamma_1, \gamma_2, \dots, \gamma_n)$. To use the soft evidence method, the following probability distribution must be specified:

$$P'(\gamma_i) = q_i, \text{ for } i = 1, \dots, n.$$

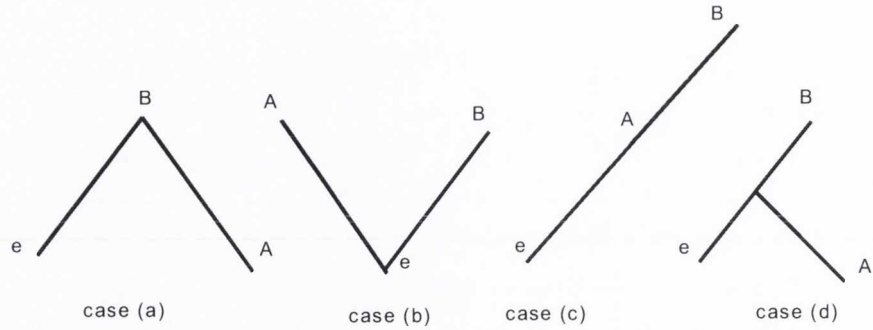


Figure 2.5: Jeffrey's rule and conditional independence.

where P' is a posterior probability distribution that results from accepting the given evidence. The principle of probability kinematics assumes that the conditional belief in any event α given any γ_i remains unchanged.

(Chan & Darwiche 2005) defines the probability kinematic principle as: Suppose that two probability distributions P and P' disagree on the probabilities they assign to a set of mutually exclusive and exhaustive events $(\gamma_1, \dots, \gamma_n)$. The distribution P' is obtained by probability kinematics on $(\gamma_1, \dots, \gamma_n)$ iff for every event α in the probability space:

$$P(\alpha|\gamma_i) = P'(\alpha|\gamma_i), \text{ for } i = \{1, \dots, n\} \quad (2.8)$$

This can be restated as two probability distributions disagree on probabilities for a set of events, but agree on how that event affects another event, e.g., $P(\alpha) \neq P'(\alpha)$ but $P(\alpha|\gamma) = P'(\alpha|\gamma)$

Jeffrey's Rule

(Chan & Darwiche 2005) defines Jeffrey's rule as follows: Given an original distribution P and some uncertain evidence on a set of mutually exclusive and exhaustive events $(y_1, \dots, y_n)$ and assuming such evidence is specified by the set of posterior probabilities $P'(y_i) = q_i$, for $i = 1, n$ the new posterior distribution P' proposed by Jeffrey's rule is :

$$P'(\alpha) = \sum_{i=1}^n q_i \frac{P(\alpha, \gamma_i)}{P(\gamma_i)} \quad (2.9)$$

This is used as in the following example, due to Jeffrey (Pearl 1986)(Chan & Darwiche 2005). Assume that we are given a piece of cloth, where its colour can be one of green, blue or violet: (c_g, c_b, c_v) . We want to know whether on the next day, the cloth will be sold (s), or not sold ($\neg s$). The original state of belief is given by the distribution P :

$$\begin{aligned} P(s, c_g) &= 0.12 & P(s, c_b) &= 0.12 & P(s, c_v) &= 0.32 \\ P(\neg s, c_g) &= 0.18 & P(\neg s, c_b) &= 0.18 & P(\neg s, c_v) &= 0.08 \end{aligned}$$

2.3. STATE INFERENCE

Therefore the original state of belief on the colour of the cloth is given by:

$$(P(c_g), P(c_b), P(c_v)) = (0.3, 0.3, 0.4)$$

Assume that we now inspect the cloth by candlelight and conclude that our new state of belief on the colour of the cloth should be:

$$(P'(c_g), P'(c_b), P'(c_v)) = (0.7, 0.25, 0.05).$$

If we revise our beliefs by applying Jeffrey's rule the new distribution P' will be:

$$P'(s, c_g) = 0.28 \quad P'(s, c_b) = 0.10 \quad P'(s, c_v) = 0.004$$

$$P'(\neg s, c_g) = 0.42 \quad P'(\neg s, c_b) = 0.15 \quad P'(\neg s, c_v) = 0.01$$

Consider the calculation of the revised belief for the proposition that the colour of the cloth is green and that it will be sold tomorrow: (s, c_g) . From Eqn. 2.9, the revised belief is calculated as: $P'(s, c_g) = \frac{P'(c_g)P(s, c_g)}{P(c_g)} = (0.7 \times 0.12)/(0.12 + 0.18) = 0.28$.

Jeffrey's Rule with Bayesian Networks

Pan et al. (2006) describes how Jeffrey's rule for belief updating can be applied to belief networks. Consider a Bayesian network N over a set of variables X modelling a particular domain, where N defines a joint distribution $P(X)$. If an observation $Q(Y)$ is received, of a probability distribution on evidence variables $Y \subseteq X$, Jeffrey's rule states that the posterior distribution of all other variables under this observation - $P'(X \setminus Y)$ should be updated to:

$$P'(X \setminus Y) = \sum_i P(X \setminus Y | y_i) Q(y_i)$$

where y_i is a state configuration of all variables in Y .

Pan et al. (2006) states that the posterior distribution or updated belief $P'(X)$ should satisfy the requirement that the conditional probability of all other variables in X given the observation Y are invariant: $P'(X \setminus Y) | Y = P(X \setminus Y) | Y$. This is a restatement of Eqn. 2.8. Thus:

$$P'(X) = P(X \setminus Y | Y) Q(Y) = P(X) \frac{Q(Y)}{P(Y)}$$

where $Q(Y)$ is the soft evidence that has been received. $Q(Y)$ can be written as $P(Y | se)$ where se is the soft evidence (possibly sensor readings) or posterior probability of Y given the evidence received.

Pearl identifies some problems with Jeffrey's rule. He states the probability kinematics rule as equating $P'(A | e)$ with $P(A | e)$ where A is a proposition that is not directly related to some received evidence e . This corresponds to an assumption that while an observation or some evidence e has changed the degree of belief in some proposition B , it did not change the conditional degree of belief in any other proposition A on the evidence B or $\neg B$. He restates Jeffrey's rule to highlight its assumption of conditional independence as: $P(A | e) = \sum_i P(A | B_i, e) P(B_i | e)$ (Pearl 1986) which agrees with Jeffrey's rule only when $(A \perp\!\!\!\perp e | B)$, i.e., A and e are conditionally independent given B_i . He illustrates this principle graphically as showing that Jeffrey's rule is only strictly correct in case (a) of Fig 2.5.

Despite the very strong conditional independence assumptions, Jeffrey's rule is very

2.3. STATE INFERENCE

convenient to use as it is not necessary to understand the process by which P' is obtained and P' can be used as a new set of priors and used in Eqn. 2.9 (Pearl 1986).

Virtual Evidence

A second method of reasoning about uncertain evidence is to treat the uncertain evidence as virtual evidence. Consider the scenario where uncertain evidence is received relating to a set of mutually exclusive and exhaustive events $(\gamma_1, \gamma_2, \dots, \gamma_n)$. This evidence can be interpreted as hard evidence on a virtual event η , where the relevance of $(\gamma_1, \gamma_2, \dots, \gamma_n)$ to the virtual event η is uncertain and is expressed as a likelihood of γ_i given this virtual evidence η , $P'(\eta|\gamma_i)$, for $i = 1, \dots, n$.

The likelihood ratio represents the observers strength of confidence toward the observed event. The likelihood ratios $\lambda_1, \dots, \lambda_n$ are defined as:

$$P(\eta|\gamma_1) : \dots : P(\eta|\gamma_n) = \lambda_1 : \dots : \lambda_n \quad (2.10)$$

λ_i can be written as $P(\text{Ob}(\gamma_i)|\gamma_i)$ and interpreted as the probability of observing γ in state γ_i if γ is indeed in state γ_i . This was referred to earlier as a likelihood function $P(o|\alpha, s')$.

The virtual evidence method assumes that for every event α in the probability space

$$P(\eta|\gamma_i, \alpha) = P(\eta|\gamma_i) \quad (2.11)$$

for $i = 1, \dots, n$. That is, the virtual event η depends only on the events $\gamma_1, \dots, \gamma_n$ and is independent of every event α given γ_i , for $i = 1, \dots, n$.

Given the constraint in Eqn 2.10 and the conditional independence assumption in 2.11, belief updating with virtual evidence is performed as:

$$P(\alpha|\eta) = \frac{\sum_{i=1}^{i=n} \lambda_i P(\alpha, \gamma_i)}{\sum_{j=1}^{j=n} \lambda_j P(\gamma_j)} \quad (2.12)$$

This theorem is provided by (Chan & Darwiche 2005) and stated to be a generalization of Pearl's method of virtual evidence. The following example originally from Pearl and referenced in (Chan & Darwiche 2005) illustrates the use of this theorem for belief updating.

Assume that we are concerned with whether a house alarm is triggered (values a and $\neg a$) and whether there is a burglary at the house (values b and $\neg b$) The original state of belief is given by the distribution P :

$$\begin{aligned} P(a, b) &= 0.000095 & P(a, \neg b) &= 0.009999 \\ P(\neg a, b) &= 0.000005 & P(\neg a, \neg b) &= 0.989901 \end{aligned}$$

This means that on any given day the prior probability of a burglary is $P(b) = 0.0001$. One day the home-owner receives a phonecall from his neighbour to say that she may have heard the alarm triggered. Since his neighbour suffers from a hearing problem, the owner concludes that there is an 80% chance that his neighbour heard the alarm ringing. This can be interpreted as follows: the probability that the neighbour will make the call given that the alarm has triggered is four times the probability that she will make the call

2.3. STATE INFERENCE

given that the alarm did not trigger. This uncertain evidence is can be considered as hard evidence on the virtual event η (the event of the neighbour calling), with likelihood ratios: $\lambda_a : \lambda_{\neg a} = P(\eta|a) : P(\eta|\neg a) = 4 : 1$.

From Eqn 2.12 the new posterior distribution P' (where $P'(\cdot) = P(\cdot|\eta)$) is:

$$P'(a, b) = 0.000369 \quad P'(a, \neg b) = 0.038820$$

$$P'(\neg a, b) = 0.000005 \quad P'(\neg a, \neg b) = 0.960806$$

For example the posterior probability $P'(a, b)$ is calculated from:

$$P(a, b|\eta) = \frac{\lambda P(a, b)}{\sum_{j=1}^n \lambda_j P(a, b)} = \frac{4 * 0.000095}{4 * 0.000095 + 4 * 0.009999 + 1 * 0.000005 + 1 * 0.989901} = 0.000369$$

Therefore the new probability that there is a burglary at the house is:

$$P'(a, b) + P'(\neg a, b) = 0.000369 + 0.000005 = 0.000374.$$

Pearl describes how virtual evidence can be incorporated into belief networks. Virtual evidence can be incorporated into a Bayesian network by adding a dummy node Z and an edge from $Y \rightarrow Z$, where one value of Z corresponds to the virtual evidence. The uncertainty of evidence is quantified by the likelihood ratios: $P(z|\gamma_1) : \dots : P(z|\gamma_n) = L(\gamma_1 : \dots : L(\gamma_n) = L(Y)$ which are specified in the conditional probability table of variable Z . The virtual evidence is incorporated by adding the observation $Z = z$ to the rest of the evidence in the Bayesian network.

Korb & Nicholson (2004) discusses the adding of virtual evidence nodes into a network. Assume in the simplest case that an uncertain observation is received for a single boolean node x with a uniform prior, i.e. $P(x, \neg x) = (0.5, 0.5)$. A virtual node v is added as a child of x and which takes boolean values. The uncertainty in the observation of x is represented by the CPT as shown in Fig. 2.6; for an 80% sure observation this gives $P(v|x) = 0.8$ and $P(v|\neg x) = 0.2$. Such a situation is shown in Fig. 2.6. (Note that the semantics of such a node are ill-defined (Korb & Nicholson 2004, 63). For example it is not clear what meaning $P(\neg v|x)$ has.)

If specific evidence is entered that v is true, Bayes' rule can be used to update the belief as follows:

$$P'(x) = \alpha P(v|x)P(x) = \alpha 0.8 \times 0.5$$

$$P'(\neg x) = \alpha P(v|\neg x)P(\neg x) = \alpha 0.2 \times 0.5$$

Since $P'(x) + P'(\neg x) = 1$, the normalisation constant $\alpha = 2$ and hence $P'(x, \neg x) = (0.8, 0.2)$. Due to the normalisation it is not the likelihoods that determine the posterior probabilities but rather the ratio: $p(v|x) : p(v|\neg x)$.

Now consider the case where the priors for x are not uniform, say $P(x, \neg x) = (0.02, 0.98)$.

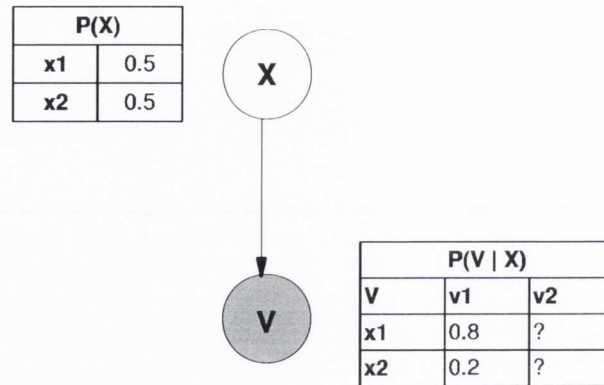


Figure 2.6: A virtual node in a Bayesian network.

If evidence is entered that v is true, inference is performed as before:

$$P'(x) = \alpha P(v|x)P(x) = \alpha 0.8 \times 0.02$$

$$P'(\neg x) = \alpha P(v|\neg x)P(\neg x) = \alpha 0.2 \times 0.98$$

which gives $\alpha = 4.72$ and $P'(x, \neg x) = (0.075, 0.925)$. The posterior belief in x is only 7.5% despite observing evidence which provides an 80% degree of belief that x is true.

Korb & Nicholson (2004) states that directly mapping the observational uncertainty into a likelihood ratio only results in the desired posterior belief for the node in question (as intended in Jeffrey rule) when the priors are uniform. If it is intended that uncertain evidence should shift the beliefs $P'(x) = P(x|v) = 0.8$ then the likelihood ratio $\frac{P(v|e)}{P(v|\neg e)}$ must be calculated appropriately given the values of the priors. $L(h)$ or $L(x|v)$ can be calculated as:

$$\lambda = \frac{P(v|x)}{P(v|\neg x)} = 0.8/0.2 \times 0.98/0.02$$

which gives $\lambda = 196$, i.e, a likelihood ratio of 196 must be used to cause a shift in the posterior belief of an earthquake given the evidence v from a prior of 0.02 to 0.8.

2.3.3.3 Criticisms of the Bayesian approach

Pearl (1990) defines Bayesianism by the following assumptions:

- a willingness to accept subjective belief as an expedient substitute for raw data. This is related to the formulation or use of prior probabilities.
- reliance on complete or coherent probabilistic models of belief.
- use of Bayes rule as the primary mechanism for updating belief in light of new

information.

Wang (2004) motivates a distinction between belief revision and updating where revision is the process by which the degree of belief in a proposition h is changed from $B(h) = p_1$ to $B(h) = p_2$, according to evidence e , and belief updating is a special case of revision in which the evidence e takes the form of " $B(h)$ should be p_2 " and is adopted, no matter what the value of p_1 is.

In the situation where there are conflicts among beliefs, i.e, $p_1 \neq p_2$, it is unusual that one piece of evidence can be completely suppressed by another piece of evidence, even though it may make sense to assume that new evidence is more relevant than old evidence. By this argument belief revision is a more general and important operation than belief updating.

Revision can thus be represented as deriving $P(h|e_0 \cap e_1)$ from $P(h|e_0)$ and $P(h|e_1)$ and can be considered as some kind of averaging of the background knowledge sets K_0 and K_1 under which e_0 and e_1 were obtained respectively.

Wang (2004) uses these assumptions to critique Bayesian belief updating. A prior probability distribution P_0 is determined by background knowledge K_0 , at time t_0 . At time t_1 a new piece of knowledge e is collected and Bayes rule is applied to change P_0 into a posterior probability distribution P_1 , where the probability of a proposition h is:

$$P_1(h) = P_0(h|e) = \frac{P_0(e|h)P_0(h)}{P_0(e)} \quad (2.13)$$

P_1 is based on K_1 , which includes both K_0 and e . By repeatedly applying Bayes rule in this conditioning process, the system learns new knowledge and adjusts its beliefs accordingly.

Under the Bayesian interpretation, a probabilistic evaluation $P(h)$ is not solely based upon an objective property of proposition h but is always conditional, in the sense that it is the relation between h and existing background knowledge K . To make this implicit conditionalisation clearer, Eqn. 2.13 can be rewritten as:

$$P_{K1}(h) = P_{K0}(h|e) = \frac{P_{K0}(e|h)P_{K0}(h)}{P_{K0}(e)} \quad (2.14)$$

where K_0 and K_1 are the background knowledge the system has at time t_0 and t_1 respectively.

Wang states that the research literature often represents the dependence of a probability to an implicit condition as an explicit conditional probability, that is, to write Eqn. 2.14 as:

$$P(h|K_1) = P(h|e \cap K_0) = \frac{P(e|h \cap K_0)P(h|K_0)}{P(e|K_0)} \quad (2.15)$$

The form of Eqn. 2.15 is intended to show that unless joint priors assumptions relating to K_0 and K_1 are met that there is an inconsistency in using Bayes rule to revise existing beliefs when new sensor data is received.

2.3. STATE INFERENCE

To apply Bayes rule to a practical domain, one of the following requirements must be satisfied:

- either the implicit condition of the initial probability distribution, that is, K_0 (the domain knowledge used to determine the prior distribution), can be assumed to be immune from future modifications; or
- all modifications of the implicit condition, (for e.g., the processing of new sensor data) can be treated as belief updating, in the sense that when new knowledge conflicts with old knowledge, the latter is completely abandoned. This prevents the inconsistencies that can arise between the assumptions used when specifying the prior beliefs and the conditions under which the evidence (sensor data) is specified and collected.

From a pervasive computing perspective, such domains are exceptions rather than general situations. In most cases it cannot be guaranteed either that all initial knowledge is unchangeable, nor that later acquired knowledge always completely surpasses earlier acquired knowledge.

Wang (2004) summarises that the Bayesian approach has no general way to handle the uncertainty within the background knowledge used in the prior probability function, i.e., to specify the confidence we have in the probability measures themselves. Thus to use a probability distribution function alone to represent uncertain state information is not enough, because it fails to show the ignorance or uncertainty about the probability function itself.

Some alternative approaches have been proposed to address this problem of ignorance versus uncertainty, including Dempster-Shafer theory which uses belief and plausibility functions to represent uncertainty and also provides an evidence combination rule to support belief revision. In the next section we present an analysis of Dempster-Shafer theory as applied to a pervasive computing environment.

2.3.4 Dempster-Shafer Belief Theory

Dempster-Shafer theory (DS) is a mathematical theory of evidence designed to distinguish between ignorance and uncertainty. Given a proposition, DS theory allows pieces of evidence to be combined to calculate the probability that the available evidence supports a proposition (either being true or false). This is different from computing the probability of a proposition (either being true or false).

DS theory differentiates between the degree of belief in a proposition given a set of evidences and the plausibility of those pieces of evidence. Interval-valued degrees of belief are used to represent the existing knowledge of the probability of a proposition.

To make this distinction clearer, consider a scenario involving the flipping of a coin from (Russell & Norvig 2003, 525): Suppose a shady character comes up to you and bets you \$10 that his coin will come up heads on the next flip. Given that the coin might or

2.3. STATE INFERENCE

might not be fair what belief should you assign to the event that the coin comes up heads?. DS theory says that because you have no evidence either way, you have to say that the belief $\text{Bel}(\text{Heads}) = 0$ and $\text{Bel}(\neq \text{Heads}) = 0$

Using probability theory, if it is known that the coin is fair, then the probability for heads and for tails is 0.5. However, if it is not known that the coin is fair, then it could be biased towards heads or tails or not biased at all. In this case, the only reasonable assumption is to say that the probability is also 0.5. These two cases are different but probability is unable to differentiate between them.

Frame of Discernment

DS theory requires, like probability theory, all the propositions to be completely specified. The set of all propositions in DS theory is called the frame of discernment and denoted by Ω . The power set of Ω is the set of all possible subsets of Ω and is denoted by $\Omega^\perp$. If $\Omega = \{a, b, c\}$, then: $\Omega^\perp = \{\emptyset, a, b, c, \{a, b\}, \{a, c\}, \{b, c\}, \Omega (= a, b, c)\}$. $\emptyset$ is the empty or null set.

Consider a pervasive computing scenario in which a smart building management system is trying to route phone calls or emails to people. The manner in which people are notified depends on their location and the social context of the location. The activity in the room can be a social function, an interview or a presentation. For example, if there are many people in a room and there is a lot of noise then the person may be notified using a loud ringtone and a vibration alert. If the activity in the room is deemed to be a presentation then the person could be notified using a silent vibration alert and, finally, if the person was engaged in an interview they should not be disturbed at all. If these are the only possible activities for a room then the frame of discernment would be $\Omega = \{s, p, i\}$ and $\Omega^\perp = \{\emptyset, s, i, p, \{s, p\}, \{s, i\}, \{p, i\}, \Omega (= \{s, i, p\})\}$. Note that $\{s, p\}$ means that the room is either being used for social function or a presentation.

Probability Mass or Measure

A probability is a mapping or function from a proposition to a real number i.e. $P(a = \text{true}) = 0.34$. In DS theory, such a mapping is called a probability measure or probability mass. The probability measure of a proposition, x , is typically denoted by $m(x)$ i.e. $m(x) = 0.457$. A probability measure, m , operates over the power set of the frame of discernment. The probability measure, m , must satisfy the following two constraints:

$$\sum_{x \in \Omega^\perp} m(x) = 1 \quad (1)$$

$$m(\emptyset) = 0 \quad (2)$$

The first constraint ensures that the total mass of every proposition in the frame of discernment sums to 1. An element that is assigned a non-zero probability is called a focal element. The second constraint ensures that the particular problem being modeled is a closed world, i.e., there is no non-zero mass assigned to an element that is not in the

frame of discernment.

Probability measures can be combined to allow different pieces of evidence to be fused. In the smart building example, a vision sensor that is detecting motion in the room and estimating the number of people present may produce a reading that assigns a probability of 0.8 that the room is either being used for a social event or a presentation.

The sensors evidence is modelled as a probability measure, $m_1(\{s, p\}) = 0.8$. The remaining 0.2 probability cannot be assigned to any element other than the frame of discernment $\Omega = \{s, p, i\}$. If the remaining 0.2 probability is assigned to the empty set $\emptyset$, this would indicate that the room was being used for another purpose and would contradict the closed world assumption and break the second restriction on m_1 .

DS Belief Theory Axioms

Two probability measures operating on the same frame of discernment can be combined using Dempster's rule of combination, to create a new probability measure or belief function. This is a generalisation of Bayes rule for belief updating. Consider two probability measures m_1 and m_2 over the same frame of discernment.

Let m_1 have focal elements in the subset A and let m_2 have focal elements in the subset B. m_1 and m_2 can be combined into a new probability measure m , with focal elements in $C = A \cap B$, by

$$m(C) = \frac{\sum_{C=A \cap B} m_1(A) \times m_2(B)}{\sum_{A \cap B \neq \emptyset} m_1(A) \times m_2(B)} \quad (2.16)$$

where the denominator is a normalising factor that ensures the probability measures in the new set of focal elements sums to 1.

Applying Eqn 2.16 creates a set C of focal elements formed by the intersection of the sets in A and B and the mass assigned to an element in C is the product of the intersecting masses. This results in an increase in the belief/support for the intersection of the focal elements of m_1 and m_2 .

In the smart building example the vision sensor assigned a probability mass of 0.8 to $\{s, p\}$, and 0.2 to the frame of discernment $\{s, p, i\}$. Assume that the system now receives evidence from an audio sensor, on the sound level in the room. This evidence leads the management system to believe that the room is being used for a social event or an interview, $\{s, i\}$ with probability mass 0.7. This might be the case if the sensor detected a number of voice signatures rather than just one, as would be the case with a presentation where the sensor might just detect the speaker's voice. Similar to m_1 , m_2 assigns a probability mass of 0.3 to the frame of discernment $\{s, p, i\}$. m_2 represents a second belief measure over the frame of discernment generated from the new sensor reading.

The focal elements of m_1 are $\{s, p\}$ and $\{s, p, i\}$. The focal elements of m_2 are $\{s, i\}$ and $\{s, p, i\}$. The intersection of these elements gives $\{s\}$, $\{s, i\}$, $\{s, p\}$, $\{s, p, i\}$. these results are shown in Table 2.3. The combined probability measure is given as m' and assigns a probability to each of these elements.

Note that the probability mass for s is 0 for m_1 and m_2 but is 0.56 for the combined

2.3. STATE INFERENCE

	$m_2(\{s, i\}) = 0.7$	$m_2(\{s, p, i\}) = 0.3$
$m_1(\{s, p\}) = 0.8$	$m'(\{s\}) = 0.8 \times 0.7 = 0.56$	$m'(\{s, p\}) = 0.8 \times 0.3 = 0.24$
$m_1(\{s, p, i\}) = 0.2$	$m'(\{s, i\}) = 0.2 \times 0.7 = 0.14$	$m'(\{s, p, i\}) = 0.2 \times 0.3 = 0.06$

Table 2.3: Dempster-Shafer example evidence combination.

function. Note also that the mass associated with $\{s, p, i\}$ for m_1 is 0.2 and for m_2 is 0.3. The combined mass for $\{s, p, i\}$ is 0.06.

Belief and Plausibility

Using a probability measure, m , two additional values can be defined that assess the belief (Bl) and plausibility (Pl) of any set of hypotheses or propositions. Firstly, the belief, $Bl(A)$, in a particular proposition A is defined as:

$$Bl(A) = \sum_{B \subseteq A} m(B) \quad (2.17)$$

The belief of a proposition is the sum of the masses of all beliefs enclosed by the proposition, i.e, the masses of all subsets. Secondly, the plausibility, $Pl(A)$, of a proposition A is defined as:

$$Pl(A) = \sum_{B \cap A \neq \emptyset} m(B) \quad (2.18)$$

The plausibility of the proposition is the sum of the masses of all beliefs that have an intersection with the proposition which is not the null set. The $Bl()$ measure represents a lower bound on the belief in a set of propositions. i.e., a lower bound on the belief in a proposition given the currently available evidence. The $Pl()$ measure represents an upper bound on the belief in a set of propositions given the currently available evidence. The difference between the belief and the plausibility is the uncertainty based on the evidence available to the system. The $Pl()$ measure can be considered as the likelihood that the system state estimate is correct.

From Table 2.3 the following belief and plausibility measures can be obtained.

- The belief in the usage context of the room being a social event is calculated as: $Bl(s) = m(s) = 0.56$.
- The plausibility of this belief given the evidence available is calculated as: $Pl(s) = m(s) + m(s, p) + m(s, i) + m(s, p, i) = 0.56 + 0.24 + 0.14 + 0.06 = 1.0$. This result of 1.0 makes intuitive sense as there is no evidence, as expressed in the probability or belief masses, that contradicts the social event usage context.
- The belief in the usage context of the room being either a social event or a presentation is calculated as: $Bl(s, p) = m(s) + m(s, p) = 0.56 + 0.24 = 0.8$. It is obvious that the belief in one of the two possible contexts will be greater than, or at least equal to the belief in either of the single contexts.

2.3. STATE INFERENCE

	$m_3(\{i\}) = 0.6$	$m_3(\{s, p, i\}) = 0.4$
$m(\{s\}) = 0.56$	$\emptyset$	$m'(\{s\}) = 0.56 \times 0.4/0.52 = 0.431$
$m(\{s, p\}) = 0.24$	$\emptyset$	$m'(\{s, p\}) = 0.24 \times 0.4/0.52 = 0.185$
$m(\{s, i\}) = 0.14$	$m'(\{i\}) = 0.14 \times 0.6/0.52 = 0.162$	$m'(\{s, i\}) = 0.14 \times 0.4/0.52 = 0.108$
$m(\{s, p, i\}) = 0.06$	$m'(\{i\}) = 0.06 \times 0.6/0.52 = 0.069$	$m'(\{s, p, i\}) = 0.06 \times 0.4/0.52 = 0.046$

Table 2.4: Updated evidence combining data from sensor 3.

- The plausibility of this belief is calculated as: $Pl(s, p) = m(s) + m(s, p) + m(s, i) + m(s, p, i) = 0.56 + 0.24 + 0.14 + 0.06 = 1.0$.

If a third sensor reading now becomes available and provides evidence that the room is being used for an interview with a probability 0.6. This new evidence conflicts with the evidence that the usage context was a social event ($m(\{s\}) = 0.56$) and the evidence that it was either a social event or a presentation ($m(\{s, p\}) = 0.24$). We can combine the existing probability measures with the third sensor's readings to get a new probability measure m' .

Note that the intersection of $m\{s\}$, $m\{s, i\}$ and $m'\{i\}$ is the empty set $\{\emptyset\}$. This results in the entries in Table 2.4 not summing to 1. Instead the entries sum to: $0.084 + 0.036 + 0.224 + 0.096 + 0.056 + 0.024 = 0.52$. The table entries are therefore normalised by dividing all entries by 0.52 so that the table entries sum to 1.

The normalisation factor is equal to $(1 - K)$ where K is the sum of all the probability mass functions or beliefs that have been assigned to conflicts. K can be calculated from Eqn. 2.19:

$$K = \sum_{A \cap B = \emptyset} m_1(A) \times m_2(B) \quad (2.19)$$

The evidence from the reading of Sensor 3 alters the belief in the usage context of the room. Prior to obtaining the reading from Sensor 3, the belief in either the social event or the presentation usage context was 0.8 and the plausibility was 1.0. After combining the evidence from sensor 3, the updated belief in the usage context of the room being used for a social event or a presentation is: $Bl(s, p) = 0.431 + 0.185 = 0.616$

The plausibility of the usage context being either a social event or a presentation is also lowered to: $Pl(s, p) = 0.431 + 0.185 + 0.108 + 0.046 = 0.769$

The uncertainty with the $\{s, p\}$ usage context is now greater and in fact the evidence contrary to this hypothesis has a confidence of $1 - Pl(\{s, p\}) = (1 - 0.769) = 0.231$. Thus, the new evidence from Sensor 3 reduces support for the proposition that the social event or the presentation is the room usage context.

2.3.4.1 Criticism of Dempster-Shafer Belief Theory

With belief and plausibility, DS theory differentiates between uncertainty and ignorance. Similar to applying Bayesian inference, new evidence can be incorporated relatively easily. In contrast to probability theory, DS theory does not consider the independence or conditional independence of particular system variables. The combination rule as presented in this chapter, effectively assumes independence in the sensor evidence.

The normalization parameter used in the evidence combination rule attributes the conflict-associated probability mass to the null set thereby ignoring the conflict completely. This can lead to extremely counter-intuitive results when the conflict between the evidence is significant. Sentz & Ferson (2006, 17) and (Wu 2003, 78) describe an example of such a case. Suppose that a patient is seen by two neurologists. The first doctor believes that the patient has either meningitis with a probability of 0.99 or a brain tumor, with a probability of 0.01. The second physician believes the patient actually suffers from a concussion with a probability of 0.99 but admits the possibility of a brain tumor with a probability of 0.01.

We can write this as

$$\begin{aligned} m_1(m) &= 0.99 & m_1(t) &= 0.01 \\ m_2(c) &= 0.99 & m_2(t) &= 0.01. \end{aligned}$$

Combining these values using Eqn. 2.16 leads to $m_3(t) = 0.0001$, i.e., the revised probability mass associated with the patient having a tumour is negligible. However using Eqn. 2.19 to calculate the normalisation parameter we find:

$$\begin{aligned} K &= m_1(m) \times m_2(t) + m_1(m) \times m_2(c) + m_1(t) \times m_2(c) \\ &= 0.99 \times 0.01 + 0.99 \times 0.99 + 0.01 \times 0.99 \\ &= 0.999 \end{aligned}$$

Therefore the normalisation parameter $(1 - K)$ results in the updated probability mass associated with having a tumour to be:

$$m_3(t) = \frac{0.0001}{1 - 0.999} = 1$$

This rule of combination yields a result that implies complete support for a diagnosis that both physicians considered to be very unlikely. A number of combination operations and methods have been developed to address the counterintuitive results of the normalisation factor in Dempster's rule when faced with strongly conflicting evidence. Research has shown that in addition to the level or degree of conflict in the available evidence, the relevance of conflict also plays a critical role (Sentz & Ferson 2006, 17). Wu (2003, 80) concludes that some degree of human intuition is always needed when combining deeply conflicting evidence.

A further problem with DS theory is that the notion of utility is not well understood.

2.3. STATE INFERENCE

Rather than computing the probability of a proposition, it computes the probability that the evidence supports the proposition. This contrasts with probability theory which makes the same ontological commitment as logic, that is, whether events are true or false in the world. The main problem with Dempster Shafer theory is that its notion of utility is weak and allows no definite decision unlike probabilistic inference (Russell & Norvig 2003, 526). This makes it a poor choice for use in planning problems.

2.3.5 Sensor Fusion

This section examines some practical issues relating to state inference. In traditional parameter measurement, every sensor has its own measurement accuracy and every measurement can have an error estimation. Classical sensor fusion in traditional parameter measurement uses statistical weighting of redundant and complementary measurements. In more complicated cases, more sophisticated sensor fusion methods are required to combine multiple measurements to produce more accurate state inference.

Sensors measure physical parameters of their environment, whereas pervasive computing applications often reason about higher-level state values. Application state will rarely correspond to a physical or atomic property of the environment.

The general challenge then is to support a mapping from sensor data to application state values. The simplest form of information mapping is one-to-one with a monotonic mapping function. For example, the room temperature measured by a thermometer sensor can be mapped into one of the three states as of cold, warm, and hot. However many applications will use multiple sensors contributing diverse data readings. When more than one sensor contributes to determining state, the mapping operation from sensor data to application state is a sensor fusion function.

2.3.5.1 Sensor Fusion Configurations

Durrant-Whyte (1988) classifies three basic sensor fusion patterns:

- Competitive sensor fusion occurs when each sensor delivers an independent measure of the same property. The aim of competitive fusion is to reduce uncertainty and error through the use of weighted average algorithms that reduce the effects of uncertain and erroneous measurements. This is often regarded as traditional or classical sensor fusion (Wu 2003, 51).
- Complementary sensor fusion combines sensor data that do not directly depend on each other to give a more complete picture of the system state. Sensor data combined according to a predefined model may enable the sensor outputs collectively to estimate high-level application state (Wu 2003, 51). Complementary sensor fusion is also referred to as feature-level fusion.
- Cooperative sensor fusion combines sensor data from a number of independent sources to derive information that would not be available from single sensors. For ex-

2.3. STATE INFERENCE

		Sensor fusion implementation model		
		Sensor level	Feature level	Decision level
Sensor fusion type	competitive	enhance reliability		flexible system configuration
	complementary & cooperative		allows complex models	

Figure 2.7: Properties of commonly implemented sensor fusion patterns.

ample, in vision applications pixels-pairs may be used to estimate object locations or distances travelled. Cooperative sensor fusion is also be referred to as decision-level fusion.

Roughly speaking competitive sensor fusion enhances measurement reliability or confidence whereas complementary and cooperative sensor fusion are used to obtain higher-level application state. Fig. 2.7, taken from (Wu 2003, 54), classifies the properties of commonly implemented sensor fusion patterns. Complementary and cooperative fusion are usually highly domain and task specific and require domain-specific knowledge-based AI techniques to be applied. If the sensors are measuring the same physical parameter, the low-level sensory data can be directly combined to enhance the reliability of state inference. Otherwise the information can be fused only at feature or decision level.

Sensor fusion schemes can be also classified using their data input-output characteristics. Table 2.5 shows a classification of inputs/outputs types (Dasarathy 1994).

Name	Description and Example
Data Input/ Data output	Input data is smoothed and filtered.
Data Input/ Feature Output	Features are generated from input data.
Feature Input / Feature Output	Input features are reduced in number or new features are generated by fusing input features.
Feature Input/Decision Output	Input features are fused together to to give an output decision.
Decision Input/Decision Output	Multiple input decision are fused together to give a final output.

Table 2.5: Input/Output Data Fusion Model

Mitchell (2007, 7) also divides the decision-input / decision-output model into two sub-models: a soft decision-input model in which each input decision is accompanied by a degree-of-support and a hard decision-input model in which the input decisions are not accompanied by any degree-of-support values.

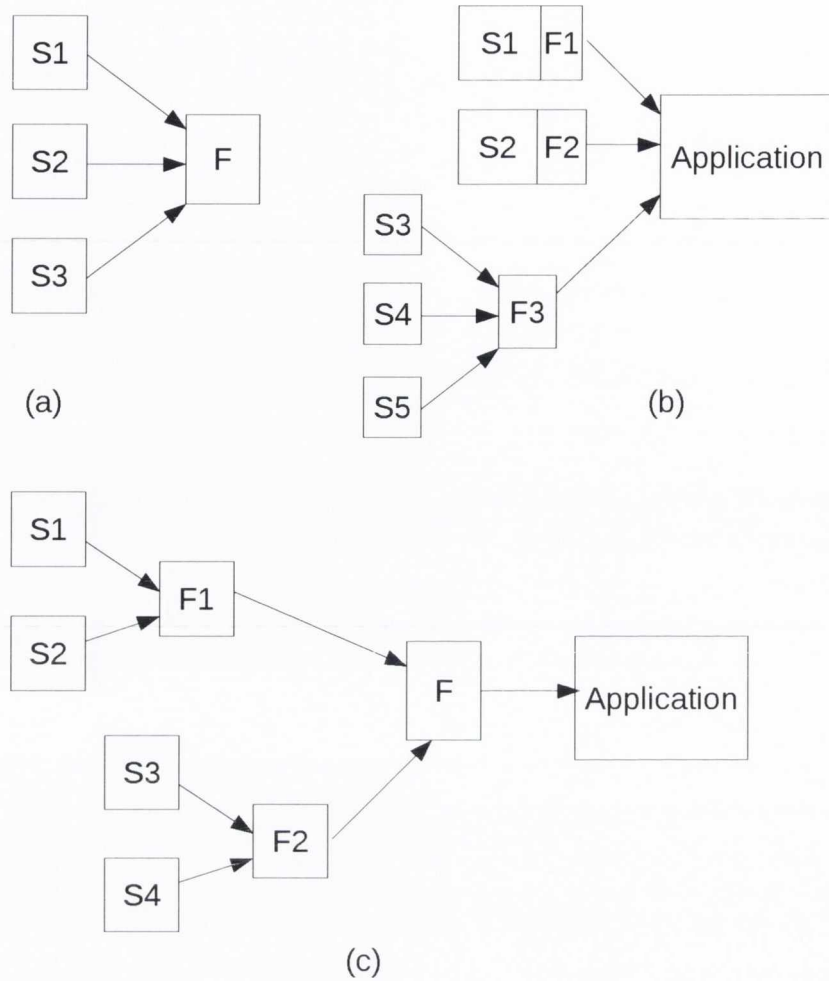


Figure 2.8: Sensor fusion architectures.

2.3.5.2 Sensor Fusion Topologies

Sensor fusion nodes may be arranged in centralized, decentralized, and hierarchical architectures as shown in Fig. 2.8. In a centralized architecture all the data from different sensors is collected and fused at a central fusion node. This is shown in Fig. 2.8 (a). Such an architecture has the usual drawbacks associated with a centralized design such as communication bottleneck, non-modularity and vulnerability (Mitchell 2007, 39).

Fig. 2.8 (b) shows an example of a decentralised architecture in which sensor measurements are fused using a set of local fusion nodes rather than at a single central node. While this provides for a more scalable design there is a possibility that redundant information may be fused thus corrupting the state inference process.

A single piece of data from a single sensor may arrive at a decentralized fusion node via multiple paths and be treated as a series of independent data readings and fused. Pieces of information from multiple sources cannot be combined unless they are independent or

have a known correlation or cross-covariance.

Mitchell (2007, 40) presents a scenario in a decentralized fusion network in which a given node F receives the same piece of information from m different sources $S_m, m \in 1, 2, \dots, M$. Suppose the piece of information sent by S_m is a scalar measurement y_m and an uncertainty σ_m^2 . If the measurements and uncertainties are regarded as independent, then the corresponding fused maximum likelihood measurement value and uncertainty are given by:

$$y_{ML} = \sum_{m=1}^M \frac{y_m}{\sigma_m^2} / \sum_{m=1}^M \frac{1}{\sigma_m^2}$$

$$\sigma_{ML}^2 = 1 / \sum_{m=1}^M \frac{1}{\sigma_m^2}$$

If all sources supply the same piece of information then: $y_1 = y = y_2 = \dots = y_M$, and $\sigma_1 = \sigma = \sigma_2 \dots = \sigma_M$. and therefore: $y_{ML} = y$, $\sigma_{ML}^2 = \sigma^2/M$.

The $1/M$ reduction in uncertainty is incorrect and occurs because the fusion node F assumes that each $(y_m, \sigma_m^2), m \in \{1, 2, \dots, M\}$, represents an independent piece of information.

One technique for reducing the effect of redundant sensor data involves maintaining covariance information to show correlations in sensor readings. However, maintaining consistent cross covariances in arbitrary decentralized networks is not possible (Julier & Uhlmann 2001). Covariance intersection (CI) algorithms have been developed to handle redundant information where correlations between data are unknown (Julier & Uhlmann 2001)(Uhlmann 2003).

CI methods use a weighted maximum likelihood estimate to fuse the measurements: $y_m, M \in \{1, 2, \dots, M\}$. The weights Ω_m are chosen such that $\Omega_m \geq 0$ and $\sum_{m=1}^M \Omega_m = 1$.

The corresponding fused estimate and uncertainty (Mitchell 2007, 41) are:

$$y_{ML} = \sum_{m=1}^M \Omega_m \frac{y_m}{\sigma_m^2} / \sum_{m=1}^M \frac{\Omega_m}{\sigma_m^2}$$

$$\sigma_{ML}^2 = 1 / \sum_{m=1}^M \frac{\Omega_m}{\sigma_m^2}$$

Assuming the weights are chosen so that all $\Omega_m = 1/M$ then the covariance intersection estimates are $y_{CI} = y$, $\sigma_{CI}^2 = \sigma^2$ and an erroneous reduction in uncertainty is avoided. Useful criteria for choosing weights are discussed in (Julier & Uhlmann 2007).

In the general case the set of M measurements will be multi-dimensional and σ_m will be expressed as covariance matrices Σ_m . Such a covariance matrix has a geometric interpretation as a covariance ellipse (Julier & Uhlmann 2001). Franken & Hupper (2005) describes a fast algorithm for calculating weights that uses sensor geometry information relating to the major and minor semi-axes of the covariance ellipses and the rotation angle

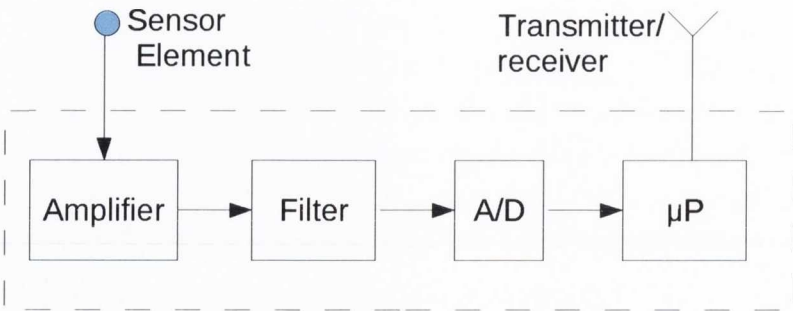


Figure 2.9: Smart sensor.

between the major semi-axes to calculate useful weights.

Finally fusion nodes may also be arranged in a hierarchical architecture as shown in Fig. 2.8 (c). In a hierarchical architecture, a series of local fusion nodes may receive data from one or more sensors. The local fusion nodes send this fused data to a single top level fusion node from where it may be made available to an application.

Fig. 2.9 from (Mitchell 2007, 16), shows the measurement of a physical property by a smart-sensor. The sensor measures a physical property and outputs an analog signal which is amplified, filtered and then converted to a digital signal by the A/D unit. The digital signal is processed by the micro-processor μP before being transmitted. The sensor may also receive queries via the transmitter/receiver unit.

Mitchell (2007, 16) and (Henderson & Shilcrat 1984) define a logical sensor as any device which functions as a source of information for a multi-sensor data fusion node. A logical sensor encompasses both smart sensors and any output of a local fusion node. For the remainder of this thesis any reference to a sensor will refer to a logical sensor.

2.3.6 Summary

Section 2.3 has described the inference problem in terms of representing uncertain information and the techniques available for updating beliefs given sensor evidence. Probability theory and Dempster-Shafer (DS) belief theory have been presented as techniques for representing and reasoning with uncertain information Appendix B evaluates the use of fuzzy logic in pervasive computing environments.

Probability theory is the most attractive technique for performing state inference in conjunction with the the tasks of planning and optimisation. The easy integration of probability theory with utility theory is very useful for decision making. Bayesian inference is predicated on probability theory and provides a mature framework for supporting belief updating in pervasive computing applications.

The use of Bayesian inference requires some application specific information, namely prior probabilities and likelihood functions which may be known by domain experts. This leads to a programming model requirement to accommodate the inclusion of such infor-

mation.

Belief updating is a critical component of planning and optimisation algorithms and the discussion of planning algorithms in the following section will show that the Bayesian framework is incorporated into a selection of planning algorithms.

2.4 Planning

This section describes a selection of planning algorithms suitable for use in pervasive computing environments. The algorithms are presented to demonstrate the relationship between algorithm design and characteristics of the problem domain.

The general class of problems addressed are those that can be modelled as follows: at any time step t , the environment is in state s , the agent takes an action a and receives a deterministic reward $R(s, a)$ from the environment as a result of this action, while the environment switches to a new state according to a transition model $T(s', a, s)$.

In this chapter pervasive computing environments have been characterised as exhibiting uncertainty, dynamism and complexity. The planning algorithms presented here are accordingly those that are typically applied in such environments.

The algorithms in this section will be described using the following components that are standard for modelling planning problems in real-world environments (Ghallab et al. 2004) and is a five-tuple $\Sigma = (S, A, T, O, R)$ where:

- $S = \{s_1, s_2, \dots\}$ is a finite set of states in which the system can be in;
- $A = \{a_1, a_2, \dots\}$ is a finite set of actions provided by the actuator infrastructure;
- $T(s, a, s')$ represents a stochastic state-transition function which gives the probability $P(s'|s, a)$ of moving to state s' if the action a is performed in state s .
- $O = \{o_1, o_2, \dots\}$ is a finite set of observations that are produced by the sensor infrastructure in the region. An observation model function $O(s', a, o)$ gives the probability $P(o|a, s')$ of observing o if action a is performed and the resulting state is s' .
- $R(s)$ is the reward function and $R(s, a, s')$ represents the immediate reward available to the system if an action a is performed while in state s and transitions to state s' .

The description of the decision and planning algorithms will focus on these components and how they are realised in a pervasive computing system. The transformation algorithms presented in Chapter 5 will show how these components are generated from the domain model design presented in Chapter 4.

2.4.1 One-Shot Decision Problems

A one-shot or single decision process performs an action selection operation that maximises the reward at that time-step. Action selection is performed using the principle of maximum

expected utility (MEU), which states that rational action selection maximises the expected utility or reward. Applying this principle to pervasive computing environments combines state inference and utility theory.

For a stochastic transition function and some evidence E that summarises the available information on system state, the expected utility of an action is calculated from (Russell & Norvig 2003):

$$EU(a | E) = \sum_{s'} P(s, a, s' | e) U(s') \quad (2.20)$$

The utility function $U(s')$ returns the expected utility value $U(\text{Result}_i(a))$ using the evidence available to it at time t_i . $P(\text{Result}_i(a) | \text{Do}(a), e)$ is obtained from the state transition system component.

2.4.1.1 Bayesian Decision Networks

Single decision processes can be compactly modeled using influence diagrams or decision networks (Jensen & Nielsen 2007, 302). Decision networks are an extension of Bayesian networks to incorporate actions and utilities. An example of a decision network is shown in Fig 5.1 (page 177). Decision networks contain three node types:

- Chance nodes (ovals) represent random variables or states of the system, exactly as in Bayesian networks. Each has an associated conditional probability table giving the probability of the variable having a particular value given a combination of values of its parents. Their parents can be decision nodes as well as other chance nodes.
- Decision nodes (rectangles) represent the decision being made at a particular point in time. The values of a decision node are the actions that the decision maker must choose between. If a decision node has a chance node as a parent this indicates that evidence about the parent node will be available at the time of decision. The actions available to the system will depend on the actuator infrastructure deployed in the region. Non-intervening actions are those that do not have a direct effect on the chance variables modelled whereas intervening actions do have direct effects on chance nodes and hence change system state.
- Utility nodes (diamonds) represent the utility function or reward model for the system and have as parents all nodes that affect the action outcome and reward. Each utility node has an associated utility table one entry for each possible instantiation of its parents. Where there are multiple utility nodes, the overall utility is the sum of the individual utilities. There is an underlying assumption that the decision will affect the utility either directly or indirectly.

Evaluating decision networks

Actions are selected by evaluating the decision network for each possible setting of the decision node. Decision networks can be evaluated using the following method:

1. Add any available evidence by setting the evidence variables for the current state.
2. For each possible value of the decision node;
 - (a) Set the decision node to that value.
 - (b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm.
 - (c) Calculate the resulting utility for the action.
3. Return the action with the highest expected utility.

The returned action is then taken to transition the system to a new state. The decision network is executed repeatedly until the system is halted.

2.4.2 Sequential Decision Making

Sequential decision making algorithms attempts to maximise the expected long term reward. (Note the term agent is used in this section as a synonym for the planning component of a system or application). If the rewards after time step t are $r_{t+1}, r_{t+2}, r_{t+3}, \dots$. Then the expected long term reward, $E\{R_t\}$, is

$$E\{R_t\} = r_{t+1} + r_{t+2} + r_{t+3} + \dots$$

Depending on the particular application, an agent's interaction with the environment may be episodic or continuous. In episodic tasks, interaction between the agent and the environment can be broken into a series of episodes and a terminal state is reached after a finite number of steps, e.g. checkmate in chess, win/lose/draw in tic tac toe, etc. If the time step to reach the terminal state is T , then the long term expected reward becomes:

$$E\{R_t\} = \sum_{i=0}^T r_{t+i}$$

In a continuous task, the interaction between the agent and the environment does not break into natural episodes and there are potentially infinitely many rewards (in the future). The typical approach is to discount future rewards using a discount rate γ . The long term expected reward then becomes $E\{R_t\} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$

This can also be written as:

$$E\{R_t\} = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (2.21)$$

where $0 \leq \gamma \leq 1$. As γ approaches 0, the discounted future reward places greater emphasis on immediate rewards from the environment. As γ approaches 1, the discounted reward places greater emphasis on rewards to be received in the future. If $\gamma = 1$, then the formula for a continuous task corresponds to the formula for an episodic task.

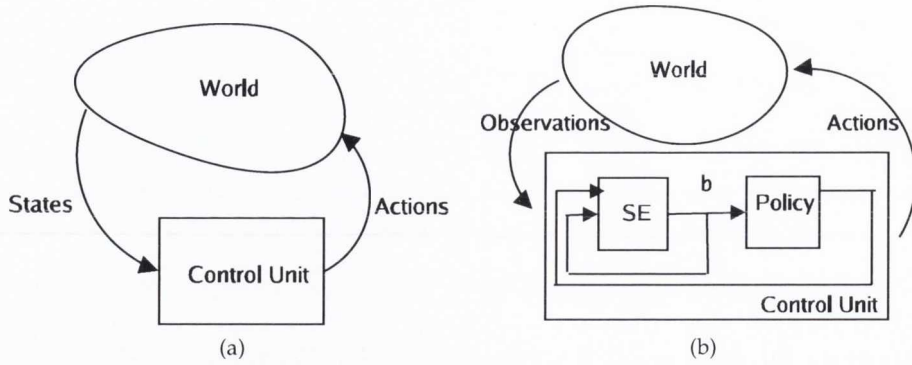


Figure 2.10: Markov Decision Process framework: (a) MDP; (b) Partially Observable MDP

A Markov Decision Process (MDP) model is a stochastic finite state machine consisting of finite sets of states, actions, and a stochastic state transition system and a reward model (Ghallab et al. 2004, 379). MDPs model sequential decision making problems and assign a value to state values or state-action combinations and are solved by maximising the reward over a time frame.

Using this model a changing world is modelled using a random variable for each aspect of the world state at each point in time. If a process exhibits the Markov property then the conditional probability of the process depends only on the most recent value of the process, not on the full collection of all past values.

As an agent transitions from state to state (by choosing actions), one assumption that is often made is that the current view summarises all the information available to the agent. This property is known as the Markov property.

Mathematically, this property can be written as:

$$\begin{aligned} P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \dots, s_0, a_0, r_0) \\ = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t) \end{aligned}$$

In other words, all the information needed to predict the next state and next reward is given by the current state and current action. Markov processes play a central role in the theory and applications of stochastic processes.

Determining the dependency among state and evidence variables is difficult. The assumption that real world processes exhibit the Markov property provides a useful method of reducing the number of conditional probability tables to be represent and is a powerful method of reducing the complexity associated with modelling real-world processes.

The behaviour of an agent is defined by its plan or policy. A policy π instructs the agent which action it should take at every time step. The policy can be considered then as the solution to the MDP. More formally, at time step t , a policy, π_t , provides a mapping from each state to action probabilities. $\pi_t(s, a)$ is the probability of choosing action a when in state s at time t . The optimal policy π^* is the one that yields the highest expected utility

or reward.

A policy can be characterised by a value function which for every state estimates the amount of discounted reward the agent can gather when it starts in s and acts according to π .

$$V^\pi(s) = E\{R_t | s_t = s\} = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s$$

where the function V^π is called the value function for policy π .

Using an additive discounted reward model the utility of a state sequence can be computed from the recursive Bellman equation:

$$V(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') V(s') \quad (2.22)$$

The value function $V(s)$ allows the agent to select an action with the maximum expected utility.

$$\pi^*(s) = \operatorname{argmax}_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')]]$$

which instructs the agent to take the action which maximises the sum of the immediate reward and the expected future discounted reward.

The Bellman equation is the basis for solving MDPs. If there are n possible states then there are n Bellman equations. Each of the n equations contains n unknowns - the utilities of the states. However, due to the non-linear max operator solving the system for each state simultaneously is problematic (Russell & Norvig 2003, 620).

Input: An MDP, discount - γ , ϵ - the maximum error allowed.

```

1 repeat
2    $V \leftarrow V'; \delta \leftarrow 0$ ;
3   foreach state  $s \in S$  do
4      $V' \leftarrow R(s, a) + \gamma \max_a \sum_{s'} T(s, a, s') V(s')$ ;
5     if  $V'(s) - V(s) > \delta$  then
6        $\delta \leftarrow V'(s) - V(s)$ 
7     end
8   end
9 until  $\delta < \epsilon(1 - \gamma)/\gamma$ ;
10 return  $V$ ;
```

Algorithm 1: Value Iteration algorithm for calculating utilities of states.

Bellman introduced the Value Iteration algorithm shown in Alg. 1. It provides a successive approximation technique based on a dynamic programming approach for solving MDPs. It uses the Bellman equation of line 4 as the basis for solving MDPs. This algorithm uses an iterative approach to calculate value functions for states. The value functions can be considered as utilities for the states. Starting with arbitrary initial values for the value of

a state sequence, in line 4 the algorithm calculates the right-hand side of the equation and plugs it into the left-hand side, thereby updating the utility of each state from the utilities of its neighbours. The update operation of line 4 is repeated until the algorithm converges and the value function stabilizes, i.e., when the largest update δ in an iteration is below a certain threshold ϵ . More details on convergence and threshold values are available in (Cassandra et al. 1994).

MDPs assume that the system states are visible to the planning algorithm. However in pervasive computing applications the system state is not directly available but is reflected through noisy sensor data or observations. The system states are therefore hidden from the agent and the MDP framework must be extended to reflect this partial observability.

Partially Observable MDP Framework

A Partially Observable Markov Decision Process (POMDP) is a refinement of an MDP to account for the uncertainty associated state inference and determination. POMDPs model the information available to the agent by specifying a function from the hidden system states to the observations produced by the sensor infrastructure. They are solved by finding a mapping from observations to actions. Unfortunately observations are not Markov because two different states might look the same due to perceptual aliasing (Cassandra et al. 1994). The system can however maintain a complete history of its past actions and observations: $h_t = a_0, o_0, \dots, a_{t-1}, o_{t-1}, o_t$. A belief state is a probability distribution over the state space that gives the probability of being in each state given the complete history of actions and observations: $b_t(s) = P(s_t = s | h_t)$. The belief state has been shown to be a sufficient or Markov statistic (Sondik 1971) that can be used to formulate the a POMDP problem as a continuous-space belief MDP.

A model of a POMDP is shown in Figure 2.10b. A belief state estimator $SE(b, a, o)$ is used to update the degree of belief in a state from $P(s' | a, o, b)$ when new observations are received. The belief update function is given by:

$$b'(s') = \frac{\sum_{s \in S} T(s, a, s') b(s) O(s', a, o)}{P(o | a, b)} \quad (2.23)$$

This equation is derived from the Bayes Rule equation given earlier:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e)}$$

where h is a hypothesised state or s' and e is the evidence received from the sensor infrastructure.

Eqn 2.23 uses Bayesian belief updating to determine the new degree of belief in s' after receiving an observation from a sensor model $O(s', a, o)$. The three terms of the numerator in Eqn 2.23 are the summation of the transition probabilities from s to s' multiplied by the probability of receiving observation o given that the system is in state s' multiplied by the prior probability that the previous state was s . This corresponds to $P(e|h)P(h)$, where the

2.4. PLANNING

probability of the sensor observations given that we have transitioned to a target state is the sum over all the ways of transitioning to the target state multiplied by the probability of receiving the sensor observations given the probability that the target state has been reached. The denominator of Eqn 2.23 is a normalising factor and is the prior probability of the evidence or observations.

The value function for a belief state for a horizon of length t is given by:

$$V_t(b) = R(b) + \gamma \max_a \sum_{o \in \Theta} P(o|a, b) V_{t-1}(SE(b, a, o)) \quad (2.24)$$

$R(b)$ is a reward function over belief states, constructed from the original reward function on system states:

$$R(b) = \sum_{s \in S} b(s) R(s) \quad (2.25)$$

The second part of Equation 2.24 is the expected discounted future rewards. $P(o|a, b)$ is the probability of observing o if action a is performed in belief state b and is calculated from:

$$P(o|a, b) = \sum_{s'} O(s', a, o) \sum_s T(s, a, s') b(s)$$

Finally, as in the case of discrete MDPs the optimal value function can be used to determine the optimal policy.

$$\pi_t(b) = \operatorname{argmax}_a [R(b) + \gamma \sum_{o \in \Theta} P(o|a, b) V_{t-1}(SE(b, a, o))] \quad (2.26)$$

POMDPs provide a rich planning framework suitable for use in environments with partial observability, however optimal results are based on an exhaustive search of a continuous belief space that is exponential to the size of the state space (Cassandra et al. 1994).

The value function encoded in Eqn. 2.24 can be represented using policy trees, similar to the search tree shown in Fig. 2.12. The agent can take an action, make an observation, then take another action, perhaps depending on the previous observation. This search tree can be considered as a policy tree. To construct an optimal t -step policy it will generally be necessary to execute different policy trees from different belief states. The optimal t -step value of starting in belief state b is the value of executing best policy tree in that belief state. The policy tree representation leads to a geometric interpretation of the value functions over a continuous belief space as being piecewise, linear and convex (Cassandra et al. 1994). This property is shown in Fig. 2.12.

Consider a world with only two states. In such a world, a belief state consists of a vector of two non-negative numbers, $b(s_1); b(s_2)$, that sum to 1. Because of this constraint, a single number is sufficient to describe the belief state. The value function associated with a policy tree p_1 , V_{p_1} , is a linear function of $b(s_1)$ and is shown in Fig. 2.11 as a line. The value functions of other policy trees are similarly represented. Finally, V_t is the maximum of all the V_{p_i} at each point in the belief space, giving us the upper surface, which is drawn

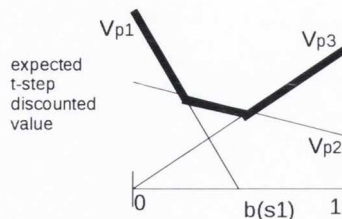


Figure 2.11: The optimal t-step value function is the upper surface of the value functions associated with all t-step policy trees.

in Fig. 2.11 with a bold line.

The convexity of the optimal value function makes intuitive sense given the values of belief states. States that are in the middle of the belief space have high entropy, the agent is very uncertain about the real underlying state of the world. In such belief states, the agent cannot select actions very appropriately and so tends to gain less long-term reward. In low-entropy belief states, which are near the corners of the simplex, the agent can take actions more likely to be appropriate for the current state of the world and, so, gain more reward (Cassandra et al. 1994).

Despite the structure of the belief space, complexity theory results show that solving infinite horizon POMDPs is undecidable (Madani et al. 1999). For large-scale pervasive computing systems that are modelled as sequential planning problems operating in possibly dynamic environments this motivates the use of approximate rather than exact POMDP algorithms.

2.4.2.1 Approximate POMDPs for Dynamic Environments

POMDPs are an active research topic with substantial research literature on approaches to solving them using exact or approximate solutions. (Ross et al. 2008) provide a recent survey on the family of online POMDP algorithms. Following the classification of (Murphy 2005) POMDP algorithms may be classified in terms of computing exact or approximate value functions over exact or approximate belief state update functions, i.e. computing approximate solutions to Equations 2.23 and 2.24 respectively. POMDP algorithms may be further subdivided as using online or offline policy computation. Offline approaches compute a policy to define which action to execute in every possible belief state. This computation is completed prior to system execution. This is only practical for small problems in a stationary environment.

Online algorithms alternate policy calculation and execution cycles while the system is running and restrict policy calculation to those belief states that are reachable over a limited forward horizon (Paquet 2006). Online approaches are therefore tractable even for systems with large state-spaces and are also appropriate for dynamic environments as the system can react to state changes that are outside of its control (Paquet 2006).

The decision cycle of an online approximate POMDP is initialised with a current belief

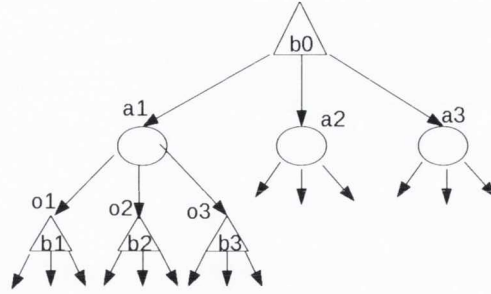


Figure 2.12: A search tree for an approximate POMDP.

state. An action specified by the policy is executed, after which an observation may be received and the system's belief state is updated. The policy may then be recalculated by summing over the sets of actions and possible observations up to a specified depth.

Real Time Belief State Search (RTBSS)

An example of an approximate, online POMDP is the RTBSS algorithm developed by (Paquet 2006) and which implements the POMDP framework as a search tree similar to that shown in Fig. 2.12.

Input: b : The current belief state. d : The current depth.

```

1 if  $d = 0$  then
2   | return  $U(b)$  ;
3 end
4 actionList  $\leftarrow$  SORT( $b, A$ ) ;
5 max  $\leftarrow 0$  ;
6 foreach  $a \in$  actionList do
7   | curReward  $\leftarrow R_B(b, a)$  ;
8   | uBound  $\leftarrow$  curReward + Heuristic( $b, a, d$ ) ;
9   | if uBound > max then
10    | | foreach  $o \in O$  do
11    | |   | curReward  $\leftarrow$  curReward +  $\gamma P(o|a, b) RTBSS(SE(b, a, o), d - 1)$ 
12    | |   end
13    | | if curReward > max then
14    | |   | max  $\leftarrow$  curReward ;
15    | |   | if ( $d = D$ ) then
16    | |   |   | action  $\leftarrow a$ 
17    | |   |   end
18    | |   end
19   | end
20 end
21 return max

```

Algorithm 2: Real-Time-Belief State Search (RTBSS) algorithm.

2.5. OPTIMISATION

The algorithm uses a branch and bound strategy to limit the search space to be explored. The bound gives an upper and lower limit on the maximal expected value associated with a policy tree descending from a belief state. Defining the upper and lower limit is domain specific. If no bounds are defined then the algorithm falls back to an exhaustive search through the belief space.

The algorithm takes two arguments, b the current belief state and d the current depth. D is a static maximal search depth. Lines 1-2 are executed when the algorithm reaches the maximal depth allowed for the search. At this point, the algorithm returns the utility value of the current belief state $U(b)$. If the current node is not a leaf of the tree, i.e., at depth D , the next step of the algorithm is to sort the actions according to the current belief state at line 4. This causes the most promising actions to be tried first with higher upper bounds and may generate more pruning early in the search tree. At line 7, for each action a the immediate reward is calculated for executing a in belief state b . At line 8 the upper bound is calculated as the sum of the immediate reward and a domain specific heuristic for estimating the bounded reward. At line 9, if pruning is turned on, the bound determines whether or not the branch will be explored. If the branch is explored then the reward of the current action a is calculated by summing the expected reward for each observation. To achieve this, line 11 calls the RTBSS algorithm recursively.

Lines 13-16 are used to record the best value for the current node among all the actions tried. It also specifies the policy by recording the action that returned the best value at the root of the tree. This algorithm has shown promising performance in partially observable dynamic environments (Paquet et al. 2005).

2.4.3 Summary

This section has presented algorithms for single and sequential decision making. Decision making is modelled as a planning problem and the impact of characteristics such as uncertainty, complexity and dynamism are reflected by refinements to the planning algorithms.

The presence of uncertainty in state determination can be mitigated by the use of probabilistic inference to determine likely system state values. The presence of complexity and dynamism in the environment can likewise be mitigated by the use of approximate and online algorithms for action selection and monitoring.

2.5 Optimisation

In Section 2.1 the optimisation problem was formulated as the problem of minimising a loss function. If Θ is the domain of allowable values for a vector θ , then optimisation is the challenge of finding the values of a vector $\theta \in \Theta$ that minimises a scalar-valued loss function $L(\theta)$.

Optimisation can also be formulated as a root finding problem, i.e., to find the values

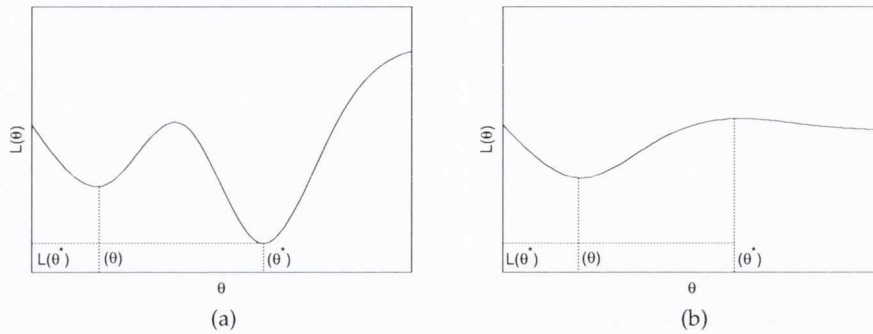


Figure 2.13: Easy and Hard problems for global optimisation: (a) a global algorithm will easily avoid θ_{local} and find θ^* ; (b) a global algorithm will only find θ_{local} as it is effectively impossible to find. θ^* .

of $\theta \in \Theta$ that solve the equation $g(\theta) = 0$ for some vector-valued function $g(\theta)$ (Spall 2003, 2). Algorithms for root-finding functions usually calculate the gradient of the loss function i.e., $g(\theta) = \partial L(\theta)/\partial \theta$, thereby requiring some understanding of the physical principles of the problem being modelled.

A formal statement of the optimisation problem as used in this thesis is given as finding the set:

$$\Theta^* \equiv \arg \min_{\theta \in \Theta} L(\theta) = \{\theta^* \in \Theta : L(\theta^*) \leq L(\theta) \forall \theta \in \Theta\}$$

where θ is a p -dimensional vector of parameters that are being adjusted and $\Theta \subseteq \mathbb{R}^p$ is the domain for θ representing constraints on allowable values for θ .

Optimisation algorithms can be divided into local and global optimisation techniques. Fig. 2.13 (Spall 2003, 6) presents two optimisation landscapes to show that the ease of finding a globally optimal solution θ^* , varies according to each problem.

The algorithms presented in this section are stochastic in the sense that there is random noise in the measurements of $L(\theta)$ or $g(\theta)$ or there is a random choice made in the search direction as the algorithm iterates toward a solution.

An important result from optimisation theory are the *No Free Lunch* (NFL) theorems (Wolpert & Macready 1997) that state that there is no one single optimisation strategy or algorithm that is optimum over the set of all possible problems. This is applicable to both noisy and noise-free problems, i.e., where the state is partially observable and fully observable respectively. Spall (2003, 18) summarises the NFL theorems as follows: If there are N_θ possible values for θ and N_L possible output values for the loss function, then there are $(N_L)^{N_\theta}$ possible mappings of θ to possible loss function values. The NFL theorems indicate that: When averaging over all $(N_L)^{N_\theta}$ possible mappings from Θ to the output space of loss function values, all algorithms work the same (i.e., none can work better than a blind random search). The NFL algorithms motivate tailoring algorithm selection to problem structure.

2.5.1 Optimisation in Pervasive Computing Environments

In Section 2.2 we characterised pervasive computing environments as being partially observable, dynamic and complex. The dependence on noisy sensor data to infer loss function values, imposes fundamental limitations on the quality of the solution due to the statistical error of the input data. The dependence on noisy measurements can be written as: $y(\theta) \equiv L(\theta) + \epsilon(\theta)$ where y is a sensor observation and ϵ represents the noise term.

The complexity of real world problems also imposes limits on the quality of solutions found. For multivariate optimisation problems, the search region grows geometrically with the dimensions of the problem. Bellman (1961, 97) describes this phenomenon as:

... the curse of dimensionality, a malediction that has plagued the scientist from the earliest days.

In pervasive computing environments, the loss function, L may be sufficiently complex so that it is not possible to obtain a closed-form analytical solution (Spall 2003, 4). This motivates the use of stochastic optimisation algorithms to find useful if not optimal solutions in pervasive computing environments.

The dynamic nature of the environments means that solutions to the loss function are time varying, i.e., the landscape of the loss function $L(\theta)$ will vary over time. In such cases the optimisation process will have to be run periodically. The cost associated with evaluating θ , i.e., the sensor invocations and processing overhead, also motivates the use of stochastic algorithms rather than exhaustive search algorithms. In general it is not possible to guarantee that an optimal solution will be found for the target class of problems we address.

2.5.1.1 Heuristic Optimisation Algorithms

In this section we consider heuristic algorithms for optimisation. A general property of heuristics is that they are very robust in the sense that they can be applied to a large class of optimization problems. This makes heuristics a useful tool across problems which have very different structures. Hromkovič (2001, 432) defines a heuristic as:

a robust technique for the design of randomized algorithms for optimization problems, and it provides randomized algorithms for which one is not able to guarantee at once the efficiency and the quality of the computed feasible solutions, even not with any bounded constant probability $p > 0$.

Simulated Annealing (SA) is a widely used heuristic algorithm based on an analogy with the statistical mechanics of annealing in solids (Suppaitnarm et al. 2000). It is a local search algorithm enriched by a randomized decision that enables the algorithm to leave local optima in order to find better solutions.

Hromkovič (2001) provides the following informal definition of solving an optimisation problem. A problem instance x of an optimisation problem specifies a set of constraints.

2.5. OPTIMISATION

These constraints determine the set $M(x)$ of feasible candidate solutions for the problem instance x . M may be empty or infinite. The objective determined by the specification is to find a solution from $M(x)$ that is the best one among all solutions in $M(x)$.

Input: A problem instance x

- 1 Start with a (randomly chosen) candidate solution α from $M(x)$;
- 2 Replace α with a neighbour of α whose loss function value is an improvement in the comparison with the loss function value of α ;
- 3 Repeat Step 2 until a solution α is reached such that no neighbour of α can be viewed as a better solution than α ;
- 4 Output α ;

Algorithm 3: Local search algorithm.

Alg. 3 shows a local search algorithm. $M(x)$ can be viewed as a graph $G(M(x))$ whose vertices are candidate solutions and two neighbours α and β are connected by the edge α, β if and only if α and β are neighbours. The local search is then a search in $G(M(x))$ which moves via an edge α, β from α to β only if $L(\beta) > L(\alpha)$ for maximisation problems and $L(\beta) < L(\alpha)$ for minimisation problems. Local search provides iterative improvement and halts with a candidate solution that cannot be improved by the movement to any of its neighbours. In this sense, local search produces local optima of $M(x)$.

SA is an advanced local search algorithm that intersperses solution improvement steps with randomised solution acceptance. The randomised step allows the possibility of transitioning to a solution with a poorer loss function value but also allows the algorithm to escape from local maxima and minima into which it may become trapped.

Input: An input instance x

- 1 Compute or select (randomly) an initial candidate solution $\alpha \in M(x)$;
- 2 Select an initial temperature (control parameter) T ;
- 3 Select a temperature reduction function f as a function of parameters T and time $I = 0$;
- 4 **while** $T > 0$ **do**
 - 5 | randomly select $\beta \in \text{Neigh}_x(\alpha)$;
 - 6 | **if** $L(\beta) \leq L(\alpha)$ **then** $\alpha = \beta$;
 - 7 | **else** generate a random number r uniformly in the range $(0, 1)$;
 - 8 | **if** $r < e^{\frac{L(\beta) - L(\alpha)}{T}}$ **then**
 - 9 | | $\alpha = \beta$
 - 10 | **end**
 - 11 | $I = I + 1$;
 - 12 | $T = f(T, I)$;
- 13 **end**
- 14 output(α);

Algorithm 4: Simulated Annealing algorithm.

The logic of the SA algorithm is shown in Alg. 4. In line 1, an initial candidate solution is randomly selected. In lines 2-3, a temperature annealing schedule is computed. The higher the temperature parameter, the more likely it is for randomised acceptance of poorer

2.5. OPTIMISATION

solutions in an effort to escape local maxima and minima. Lines 4-12 shows the execution cycle of the algorithm in which candidate solutions are repeatedly generated and evaluated. In line 12, the temperature is annealed or reduced after each evaluation cycle. As the temperature reduces the probability of accepting a poorer candidate solution decreases and the quality of candidate solutions converges.

The robustness and wide applicability of SA have made it a widely used optimization algorithm. The successful application of SA to a particular optimisation problem depends on the choice of the algorithm parameters - the state space and cooling schedule. Hromkovič (2001) provides guidelines on specifying the cooling schedule and concludes that SA can provide feasible solutions of high quality at polynomial time but that it is not possible to guarantee high quality solutions after a reasonably restricted number of iterative steps.

The randomised candidate acceptance shown in line 8 of Alg. 4, uses the Metropolis algorithm to calculate acceptance probabilities. The Metropolis algorithm accept a deterioration in candidate solution quality with the probability:

$$p(\alpha \rightarrow \beta) = e^{-\frac{L(\beta) - L(\alpha)}{T}}$$

The probability $p(\alpha \rightarrow \beta)$ follows the laws of thermodynamics in that the probability of the movement from candidate solutions α to β decreases with $L(q) - L(s)$, i.e., large deteriorations are less probable than small deterioration and the probability $p(\alpha \rightarrow \beta)$ increases with T , i.e., large deteriorations are more probable at the beginning when T is large rather than later when T becomes smaller and smaller.

2.5.1.2 Multi-objective Optimisation

Applications may require more than one objective to be optimised simultaneously. Multi-objective optimisation problems may be defined with several incommensurable and conflicting objectives. There may be no optimal solution across all objectives but there may be a set of alternative solutions that are useful in the sense that there are no other solutions in the search space that are superior to each other when all objectives are considered. Such a set of solutions are known as Pareto-optimal solutions (Suman & Kumar 2006). More formally Pareto optimal solutions are defined as follows: Consider a minimization problem with two decision vectors a and b . a is said to dominate b if

$$\begin{aligned} \forall i \in \{1, 2, \dots, N\} : f_i(a) \leq f_i(b) \text{ and} \\ \exists j \in \{1, 2, \dots, N\} : f_j(a) < f_j(b) \end{aligned}$$

When the objectives associated with any pair of non-dominated solutions are compared, it is found that each solution is superior with respect to at least one objective. The set of non-dominated solutions to a multiobjective optimization problem is known as the Pareto

2.6. SUMMARY

optimal set (Suman & Kumar 2006).

Step 1: Start with a randomly selected candidate solution α , evaluate the multi-dimensional loss function and store the candidate solution in the Pareto set of solutions.

Step 2: Using a random perturbation select a new candidate solution β , in the neighbourhood of current candidate solution α and re-evaluate the loss function.

Step 3: Compare the β candidate with all the solutions in the Pareto set. If β is useful, i.e., not dominated, then it is archived or stored.

Step 4: If β is archived, make it the current solution by making $\alpha = \beta$ and go to Step 7.

Step 5: If β is not archived, accept it with probability: $P = \min(1, \prod_{i=1}^N \exp(-\frac{\Delta s_i}{T_i}))$ where $\Delta s_i = (l_i(\beta) - l_i(\alpha))$

If β is accepted after the Metropolis step, make it the current solution and go to Step 7.

Step 6: If β is not accepted, retain α as the current solution and go to Step 7.

Step 7: Periodically restart with a randomly selected candidate solution from the archived Pareto set.

Step 8: Reduce the temperature periodically using a problem dependent annealing schedule.

Step 9: Repeat Steps 2-8, until a predefined number of iterations is carried out.

Algorithm 5: SMOSA algorithm.

There are numerous multi-objective simulated annealing algorithms as surveyed in (Suman & Kumar 2006). One algorithm with good performance across a range of problems is the SMOSA algorithm (Suppaitnarm et al. 2000).

The SMOSA algorithm logic is shown in Alg. 5. At every evaluation step the newly generated candidate solution is evaluated against the already archived candidates. If it is added to the archive then the archive is pruned to remove any newly dominated solutions. As a result of successive pruning, the probability of accepting new solutions decreases as the execution cycle progresses. Due to the metropolis step it is possible for a dominated solution to be temporarily accepted. When this happens the dominated solution is used to seed the generation of the next candidate solution and then discarded.

Figure 2.14 shows a pareto-optimal solution archive. The points marked in X represent candidate solutions generated by the SMOSA algorithm that are located on the Pareto frontier. The squares below the frontier represents generated candidate solutions that are dominated in both dimensions by stronger solutions and will be pruned.

2.6 Summary

This chapter has profiled pervasive computing environments as being characterised by uncertainty, dynamism and complexity. The tasks of state inference, planning, and optimisation have been introduced and algorithms typically used for these tasks have been presented. These algorithms are chosen to be representative rather than prescriptive of the approaches used to tackle such problems.

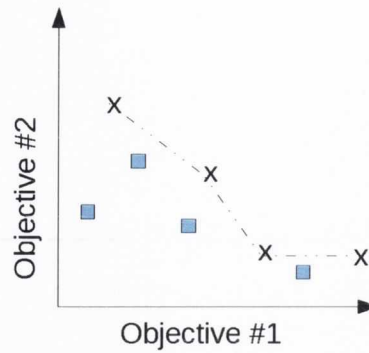


Figure 2.14: Pareto-optimal frontier.

The impact of uncertainty, dynamism, and complexity on the tasks of inference, planning and optimisation has been discussed. The review of state inference techniques concluded that a Bayesian inference framework was most appropriate for handling state uncertainty in planning and optimisation tasks. The review of planning and optimisation algorithms indicates that approximate or heuristic approaches are often used for noisy and dynamic environments as deterministic and optimal solutions may be infeasible. If the complexity of the problem precludes a deterministic and exhaustive approach, heuristic techniques are useful. This is especially true in dynamic environments where solutions may need to be recalculated periodically in response to changing application state.

We look on past ages with condescension, as a mere preparation for us
... but what if we're only an after-glow of them?
J.G. Farrell, (The Siege of Krishnapur)

3

State of the Art

This thesis proposes a MDE methodology for developing planning and optimisation applications for pervasive computing environments. This chapter examines the support currently available for the development of these applications. This chapter contains three sections. The first section examines how developers approach inference activities in pervasive computing applications, the second section examines how model-driven engineering approaches are applied to developing pervasive computing applications, and the third section examines how planning and optimisation algorithms are engineered for use in pervasive computing applications.

In the previous chapter a pervasive computing environment was defined as a region of the physical environment or real world augmented with sensor and actuator devices. A pervasive computing application executes in such an augmented physical space and may use sensor data to infer application state space values.

Planning and optimisation algorithms are used by those pervasive computing applications that perform some combination of the following tasks:

- Applications may use actuator infrastructure to plan and effect changes in the environment in accordance with some user specified objectives.
- Applications may use this inferred state information to alter or optimise their behaviour in response to changes in the environment.

The terms pervasive computing and context-awareness are widely used in the related research literature. Definitions for these terms are presented here, and the meaning of these terms, as used in this thesis is clarified.

3.1. SCOPE AND STRUCTURE OF THE REVIEW

Banavar et al. (2000) define pervasive computing with relation to: the way people view and use mobile computing devices to perform tasks; the way applications are created and deployed in support of those tasks; and how the environment is enhanced by the emergence and ubiquity of new information and functionality. They view computing devices as providing a portal to an information-enhanced application domain and promote a vision of a pervasive computing as task-oriented applications executing in an information enhanced physical space.

The research literature provides many definitions of context. Dey (2001) defines context as any information that can be used to characterise the situation of an entity, where an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves. Ye et al. (2008) defines context as being any information that is used to characterise the situation of users, their environment or their tasks. They define a context-aware computing system as one that exhibits appropriate and customised behaviours that adapt to the change of users' context. Gellersen et al. (2002) defines context as information that can be acquired from the real world through sensor capture and fusion.

This thesis is only concerned with context-aware computing as an enabling technology for pervasive computing and we consider context information as any high-level application-specific information derived from sensor data. In this thesis, the term pervasive computing refers to task-oriented applications that use information derived from sensor data. The focus of the thesis specifically relates to the engineering of planning and optimisation algorithms to accomplish desired tasks. The domain of such applications is considered to be the physical or real-world domain in which the application executes.

3.1 Scope and Structure of the Review

The research reviewed is presented in three sections. The first section addresses how uncertain application state information is represented and used in context-aware computing systems. The review aims to answer the question of how uncertain information is represented and how it is used to determine application state.

The second section reviews the application of MDE efforts to developing pervasive computing applications, to identify the services provided by MDE frameworks in this domain. Related work is reviewed to determine the modelling abstractions provided for pervasive computing environments and the level of tool support provided for generating application functionality from models. Finally the executable model and middleware requirements of each approach is reviewed.

The third section reviews the systematic engineering of planning and optimisation algorithms for pervasive computing environments. The related work is reviewed with the following questions in mind: Related work is reviewed to determine the range of planning and optimisation algorithms supported and how algorithms are selected and seeded with problem data. Finally the executable model and middleware requirements of

each approach is reviewed.

3.2 Resolving Uncertainty in Pervasive Computing

The uncertainty of sensor data and the difficulty of reasoning under uncertainty are challenges for the developers of pervasive computing applications. In this section we survey approaches to representing and reasoning with uncertain data in the pervasive computing community. Chapter 2 presented theoretical approaches for modelling and using uncertain data for state inference. This section samples how developers are representing and using uncertain context or application state information in practice.

3.2.1 Bayesian Approaches

This section reviews two Bayesian approaches to representing and reasoning with uncertain sensor data to determine context information.

3.2.1.1 Representing and reasoning with uncertain context data in SOCAM

Gu et al. (2004) propose a Bayesian approach for representing and reasoning about uncertain contexts. They use the Web Ontology Language (OWL) ontology markup language (Dean & Schreiber 2004) to model context data using first-order predicate calculus. The basic model has the form of $\text{Predicate}(\text{subject}, \text{value})$ in which:

- $\text{subject} \in \{S : \text{a set of subject names}\}$, e.g., a person, a location or an object.
- $\text{predicate} \in \{V : \text{a set of predicate names}\}$, e.g., "is located in", "has status", etc.
- $\text{value} \in \{O : \text{a set of all values of subjects in } S\}$, e.g., "the living room", "open", "close", "empty", etc.

A sample usage is $\text{Location}(\text{John}, \text{bathroom})$, which captures the information that if the predicate *in* is applied to a subject *John*, the resulting value is *bathroom*, i.e., "John is in the bathroom". The authors propose to extend this model to the form: $\text{Prob}(\text{Predicate}(\text{subject}, \text{value}))$, which includes a probability value in the range $[0, 1]$. To support this extension the authors define two OWL classes: *PriorProb* and *CondProb* which can be associated with context predicates.

The authors use Bayesian networks to perform inference using the probability annotated context predicates. Predicates are modelled as nodes in a network and arcs between nodes should reflect causal relationships in the domain. They introduce an OWL class *dependsOn* to model the causal relationships between elements. If the OWL data is augmented with the probability and dependency information then a Bayesian network can be generated from the context model.

The authors use the service-oriented context-aware middleware (SOCAM) middleware to support inference using context data (Xiao et al. 2004). SOCAM provides a context model

3.2. RESOLVING UNCERTAINTY IN PERVASIVE COMPUTING

built over the basic concepts of person, location, activity and computational entity (such as a device, network, application, service, etc.). SOCAM provides support for mapping sensor data to Bayesian nodes generated from context data. No evaluation criteria are provided for the approach and a sample usage scenario is presented by way of an evaluation.

3.2.1.2 Resolving Uncertainty in Context Integration and Abstraction

Ye et al. (2008) also present a Bayesian approach to evaluating and resolving uncertainty in context information derived from sensor data and used in pervasive computing applications. The authors consider context as any data acquired from various kinds of sensors that are distributed in a pervasive computing environment. They propose a model of context uncertainty and a technique for resolving or minimising uncertainty by distilling the most accurate context from a large number of trivial and noisy contexts.

They categorise sensors according to the type of data they provide, environmental sensors that provide data from the physical environment, such as temperature and noise, positioning sensors that locate and track movement, device sensors that report the state of hardware and equipment, and virtual sensors that extract information from other software or applications. Each sensor type is profiled to establish its frequency, accuracy and precision. Frequency is the sampling rate for the sensor, precision defines the sensor data range, and accuracy is the reliability of the sensor.

They represent context using an Resource Description Framework (RDF) tuple $c = ((s, p, o), t, \text{conf})$ where:

- (s, p, o) is a RDF triple, that indicates that a subject s is associated with an object o in a relationship p , where s and o are context types.
- t is a life span of a triple, which can be a time instant or a time interval with a starting time t_s and an ending time t_e
- $\text{conf}(\in [0, 1])$ is a context confidence degree to which the subject and object are correct in this predicate. The context confidence is the system's belief in the truth of a context and it is used in a similar fashion to a probability measure.

An example context is:

$((\{\text{erica}\}, \text{hasLocation}, \{[12.22, 5.26, 0.09]\}), "2007 - 10 - 26" "09 : 36 : 02", "0.7")$

which indicates that a person named Erica was located at a coordinate $[12.22, 5.26, 0.09]$ at a time instant "2007-10-26 09:36:02" with a confidence "0.7".

The confidence of a piece of context data is determined by the sensor type from which it is derived. If the context is derived from an environmental or positioning sensor then its confidence is taken from the accuracy metric associated with the sensor type. Confidence values for contexts derived from a virtual sensor such as an online calendar must be determined in a domain and application specific manner.

Context Integration and Abstraction

The authors present a context integration framework designed to combine multiple context data with a shared predicate. The context integration framework is used to resolve conflicting context data and increase the accuracy of fused context data

They define a procedure to determine the timeliness of sensor data relating to contexts and a Bayesian value factor calculation to determine the conditional probability of a context given the pertinent sensor data readings available.

Context integration requests are expressed in the form: $((s, p, ?), t, ?)$ i.e., to decide on the object value, given a subject, predicate and time. The procedure is carried out by computing a time and value factor. The time factor ranks data in order of the freshest or most recently received data. The value factor measures the level of agreement between disparate contexts. Bayes rule is used to compute the conditional confidences for each context value, i.e. to determine how probable the context value is given all the other available context values. The time and value factors are combined to select the most likely predicate values from a set of disparate contexts.

The authors also define a process called context abstraction whereby a number of contexts are aggregated to identify a higher-level situation. A situation is specified as a series of constraints of context values. An example is given of a "meeting" situation, specified using a constraint on a calendar - a meeting scheduled; a constraint on a person's location - in a scheduled meeting area; and a constraint on the current time - in the scheduled meeting time. A confidence value, called a "match degree" is associated with each situation and calculated using fuzzy set membership functions on constituent context predicate and confidence values.

To evaluate their approach the authors provide a proof of concept case study of a smart office scenario. They analyse the complexity of their context integration procedure as $O(2^n)$ where n is the number of input context values. They state that this exponential complexity is unlikely to be problematic in practice because the typical number of shared predicate contexts will be limited.

3.2.2 Dempster-Shafer Context Sensing

Wu (2003) applies DS techniques for sensor data fusion for context-aware computing. The authors define an operation called "context sensing" whose objective is to bridge the gap between data provided by sensors and the information required by applications.

The context sensing approach is based on context-widgets, where a widget is defined as a software agent that transfers low-level sensor data into a predefined higher-level context. Examples of context-widgets include sound processing widgets to convert microphone readings into speaker recognition context information, and map registration widgets to transform GPS readings into location context.

The authors provide a context sensing architecture shown in Fig 3.1. This architecture is designed to provide sensor fusion support to the Georgia Tech Context Toolkit system

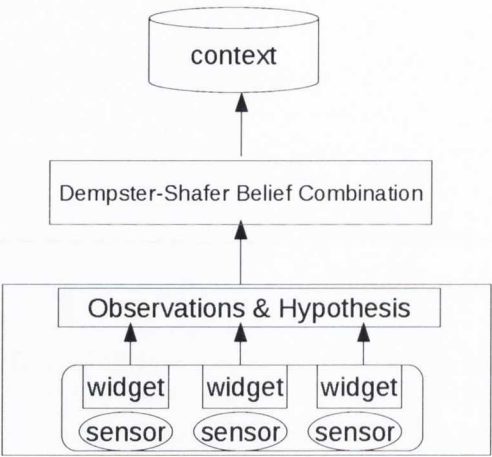


Figure 3.1: Context Sensing architecture

(Dey et al. 2001). Fig. 3.1 shows three layers. The lowest layer consists of widgets collecting sensor data relating to hypotheses that are of interest to an application. The second layer uses DS belief theory to fuse sensor data by combining compatible evidence and resolving conflicting data. The highest layer uses the fused hypotheses to determine context data and makes context information available to applications.

The authors acknowledge the difficulties caused by evidence conflicts and propose a weighted DS evidence combination rule to adjust different sensors’ voting effectiveness according to their specifications and historical performances. In the event of evidence conflicts, evidence from sensors with larger weights are trusted over evidence from lower weighted sensors. Wu et al. (2002) show that the weighted DS combination rule provides more accuracy in identifying user activities than a classical DS approach.

This approach is shown to alleviate but not eliminate difficulties caused by conflicting sensor evidence. The authors state that the intended applications are for human-computer interaction and human intervention is available to resolve significant conflicts where necessary (Wu 2003, 80).

3.2.3 Hybrid Approach to Deriving Context

Dargie (2007) defines context as a higher-level abstraction of a dynamic real world-situation and proposes a conceptual architecture for combining probabilistic schemes for deriving context data from multiple physical sensors. The architecture proposes a hybrid approach to deriving context and motivates the use of fuzzy logic, Hidden Markov Models (HMMs), Bayesian networks (BNs), and Dempster-Shafer (DS) belief theory at different levels.

Fuzzy logic is described as appropriate for membership problems where disparate pieces of data can be combined to determine the context to which they belong. HMMs are proposed for predicting behaviour or sequences of events. BNs are appropriate for computing a single higher-level context as an abstraction over numerous primitive contexts

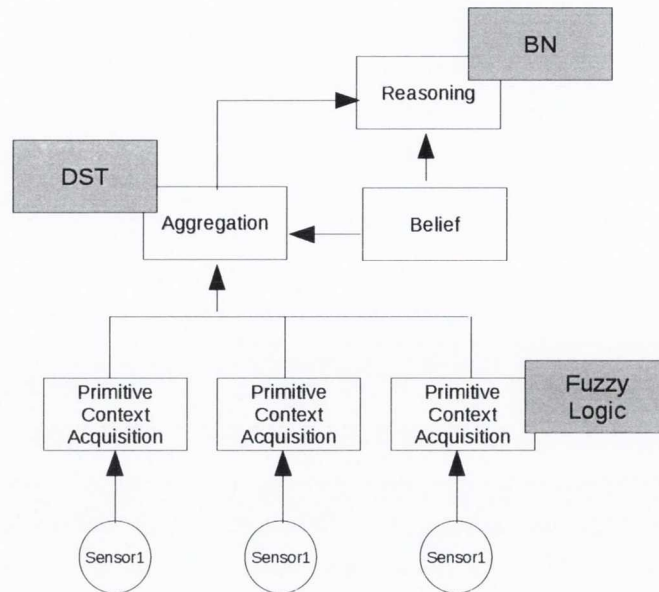


Figure 3.2: Conceptual architecture for inferring context

or sensor reading. Dempster-Shafer theory (DST) is appropriate for combining evidence gathered from two or more independent sensors to minimise the effect of imprecision.

The conceptual architecture for sensor fusion is shown in Fig. 3.2, and computes context in three steps. The first step captures primitive context data by directly employing sensors. Fuzzy logic is employed at step one to transform sensor data into application meaningful values. For example temperature readings might be transformed into values such as warm or cold.

The second step aggregates multiple primitive contexts describing a single state. DS theory is used to combine different primitive contexts. A DST processor takes as its input the fuzzy sets and combines them using the DST combination rule.

The third step combines the aggregated contexts to compute high-level application contexts. HMMs are used if multiple time-dependent higher-level contexts are to be computed. BNs are used if a single high-level application context is to be computed from several primitive contexts.

3.2.4 Analysis

The following commonalities are noted in the approaches reviewed. All the approaches view context as being conceptually different from sensor data and provide both low-level sensor fusion to combine sensor data and some high-level support to infer context from fused sensor data.

Both Gu et al. (2004) and Ye et al. (2008) use Bayes rule to combine conditional probabilities of multiple context data with a shared predicate, with the former proposing BNs

for higher-level inference, while the latter use fuzzy set membership for combining context to derive a higher-level "situation". Both approaches also use annotated predicate logic to represent a context datum and its associated uncertainty, thereby encoding the uncertainty of the context data into its representation.

Dargie (2007) and (Wu 2003) recognise that no single technique is universally useful when deriving high-level context from fused sensor data. Chapter 2 provided an example of the weakness of DS theory when resolving conflicting evidence and Wu (2003) concludes that DS theory is primarily suitable in human-centric tasks where human intervention is available to resolve seriously conflicting evidence.

Given the target domain addressed in this thesis, the methodology we propose must use robust low-level sensor fusion techniques to handle mobile sensor and actuator infrastructure of varying quality. The planning algorithms reviewed in Chapter 2 require uncertainty estimates for all state data. The representation of context or application state provided by the methodology must also encode an uncertainty estimate. Finally the related work indicates a need for the methodology to allow flexible higher-level state inference to be specified on a per state basis.

3.3 MDE in Pervasive Computing

This section reviews the application of MDE techniques, as described in Chapter 1, to engineering pervasive computing applications. In particular the focus is on the nature of the models used, the transformations employed, the degree of automation for code generation, and the challenges tackled by the approach.

When reviewing each research project in this section the following questions are kept in mind:

- How are models structured?
- How are models transformed into an executable?
- How are executables run?

3.3.1 Olympus

Olympus is a high level programming model for pervasive computing environments. The Olympus programming model is built over the Gaia middleware (Román et al. 2002) and supports an active space modelling abstraction (Ranganathan & Campbell 2005a) (Ranganathan & Campbell 2005b). An active space is a physical space that has been enhanced with a large number of digital devices such as sensors, computing devices and actuators.

The variety of digital devices and the dynamism associated with them make it difficult for developers to program active spaces. Active spaces may host a large number of different types of services and applications, and there will potentially be different ways of

performing a task using different combinations of the hardware and services that are in the space. Some of these ways of performing tasks will be better than others depending on the current context, resources available and user preferences.

The Olympus model provides developers with high-level abstractions when programming such spaces so that developers can specify desired functionality without being aware of the specific resources available, context, policies and user preferences while developing their applications.

Abstractions are provided in the form of eight virtual entity classes that include: services, applications, devices, physical objects, locations and users. An entity is a physical instance of a service, application or device and a virtual entity is an abstract representation of one of these objects. The Olympus model uses a discovery service provided by the GAIA framework to associate or resolve these virtual entities into actual entities in the environment. If the discovery service finds multiple potential entities in a space to select, then the framework uses a utility model to select the most suitable entity.

The Olympus programming model provides:

- An ontology for specifying entity hierarchies and properties. This ontology is specified in OWL.
- A semantic matching algorithm and utility function used to select appropriate physical entities for performing a certain task.
- Policies and rules for specifying the utility function used in the semantic matching algorithm. The utility function can be expressed using the following properties:
 1. Location of the entity, i.e., closer entities are preferred.
 2. The semantic similarity of the tasks supported by the entity.
 3. The state of the entity, e.g., load levels and battery resources.
 4. The context of the space, e.g., a user in a meeting room may prefer to receive silent notifications of events.
- A set of high-level operators for performing certain common tasks in active spaces. The high-level operators are: `stop`; `start`; `saveState`; `suspend` and `resume` operators. These operators are used to manage the lifecycle of active-space entities and to query or change the state of active-space entities.

Mapping abstractions to physical entities

Virtual entities are mapped at run-time to appropriate physical entities in the active space. This mapping is facilitated by an ontology providing properties of physical entities. The ontology is written in OWL. A developer can also browse the ontology to see what kinds of entities are available and what constraints are specified on their use. The ontology for an active space is initially created by an administrator and extended by application developers to include descriptions of new entities added to the active space.

```
ActiveSpace as1;      //refers to a virtual active space entity
as1.hasProp("containsPerson","Sal");

as1.instantiate();    /*as1 now refers to the Active Space where
                      the user Sal is located */

Application app1;     //refers to a virtual application entity
app1.hasProp("class","Slideshow");
app1.hasProp("file","Olympus.ppt");
app1.start(as1);      /*app1 is started in Active Space as1 in
                      the "best" possible configuration */
```

Listing 3.1: Sample Olympus program.

An application developer can specify constraints that entity instances selected at run-time must conform to. These constraints take the form: <entity><property><value>. An example is given as: `device1.hasProp("resolution","800*600")`. These constraints are used by the semantic matching algorithm referred to above. The administrator of a space can also specify space-level policies for an active space. These policies are written in the form of rules using Prolog. These policies act as constraints on an active space. An example is given as:

```
disallow(presentation, notify, user):-
subclass(presentation, audiopresentation),
activity(user, meeting)
```

This policy specifies that no audio presentation application component should be used to notify a user in case he is in a meeting.

Model Transformations

No model transformations are used in the Olympus model. A semantic matching algorithm automates the selection of the entity that will provide the required service, given the high-level policy supplied by the user. This functionality is supported by the Gaia service discovery framework which returns all entities that match the requirements. The returned entities are then evaluated using a utility function that combines the policies, constraints and contexts specified by the developer.

Execution Model

The Olympus model is implemented in C++ and developers using the Olympus programming model must specify virtual entities and active space operators using C++. Any instantiation of an active space entity in the code engages Gaia middleware services to support execution.

In this code excerpt the developer is trying to start a power-point slideshow application in a room containing a user called Sal. At execution time the programming model uses the matching algorithm and the middleware discovery service to locate a device that can display the power-point slides.

Olympus programs consist of two main segments. The first segment declares virtual entities, desired properties and constraints. When the virtual entity constructor is called the framework calls the discovery service to find suitable physical entity instances. In the second segment of an Olympus program, developers can use the high-level operators listed earlier, on the selected entities. An implementation of these high-level operators must be provided for each active space.

No detailed evaluation of the programming model is provided. The authors show some code samples and state that they have noticed that development time was reduced when using the Olympus programming model to program a number of active spaces.

3.3.2 Physical Active Objects

Baresi & et al (2006) propose a model-driven approach to developing pervasive computing applications. Their approach provides a physical active object (PAO) abstraction for modelling real and virtual sensors. The authors provide a metamodel with container and wrapper classes to associate sensors with application data, and a framework for automatic code generation based on the Jini middleware (Newmarch 2000).

The authors cite the need for a systematic approach to support the integration of sensors with applications designed to control and manage the physical environment. Accordingly, they provide a reference model architecture shown in Fig. 3.3, for this class of applications. The reference model is shown as it would be applied to designing an application to track a container of goods in the logistics domain. PAOs are situated at the lowest layer of the architecture and represent GPS antennas and RFID tags.

The second layer, the device access level, manages access to the physical objects. An Operation Manager abstraction is provided to integrate the PAO model with the middleware used to access sensors. Operation managers generate events upon receipt of relevant sensor data.

The third layer, the logical level, contains logical objects mapped to sensors. A change in the real world as detected by a sensor, is reflected in a state change in associated logical objects. For example if an RFID tag is seen by a reader then an event is raised by an operation manager and the state of the associated logical object is updated.

The highest layer, the application level, contains application objects and contexts. Multiple logical objects may be associated with an application object. Application objects combine data from logical objects to provide high-level data in an application specific manner. For example in the container tracking example an application object provides the location of a container derived from fusing logical object data obtained from RFID and GPS sensor data.

Multiple application objects may be combined into contexts.

Modelling Support

The authors provide an object-oriented metamodel for modelling applications using the proposed reference architecture. The metamodel contains a base abstract class called a

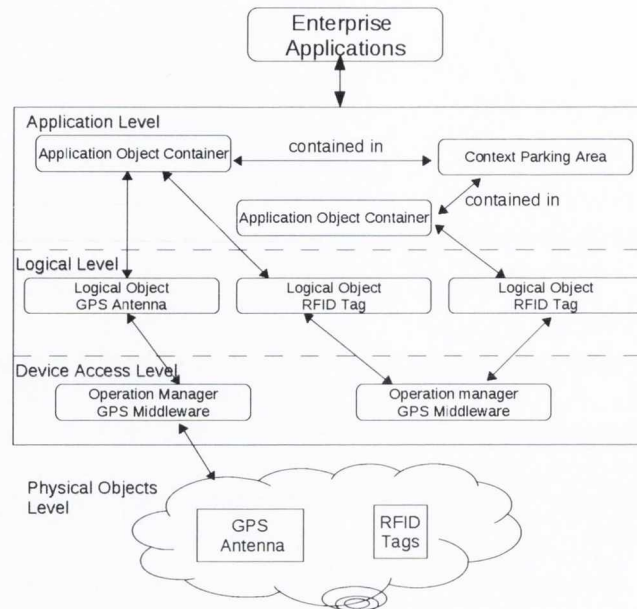


Figure 3.3: PAO abstraction levels.

FunctionalElement extended by logical and application objects. Application objects have methods to attach logical objects, while context objects provide methods to attach application objects. UML statecharts can be used to specify the behaviour of logical and application objects.

Model Transformations

The PAO approach supports two model transformations. The first transformation parses application models and uses the Eclipse Java Emitter Template Framework¹ to create code skeletons to invoke Jini services.

The second transformation uses UML statechart information to generate stub code for logical and application objects. The authors state that the stub code will usually have to be integrated with hand written code for any non-trivial application.

Execution Model

The authors provide a framework shown in Fig. 3.4, that incorporates Jini middleware services, to support code generation and to execute PAO based systems.

A Factory component based on the abstract factory pattern, is responsible for instantiating services that belong to the application. The framework also contains an observer component that listens to the physical environment through operation managers that provide access to the underlying sensor infrastructure.

The authors have prototyped some of the components of the programming model, but their evaluation is limited to describing a usage scenario based on an intelligent container system.

¹<http://www.eclipse.org/emf>

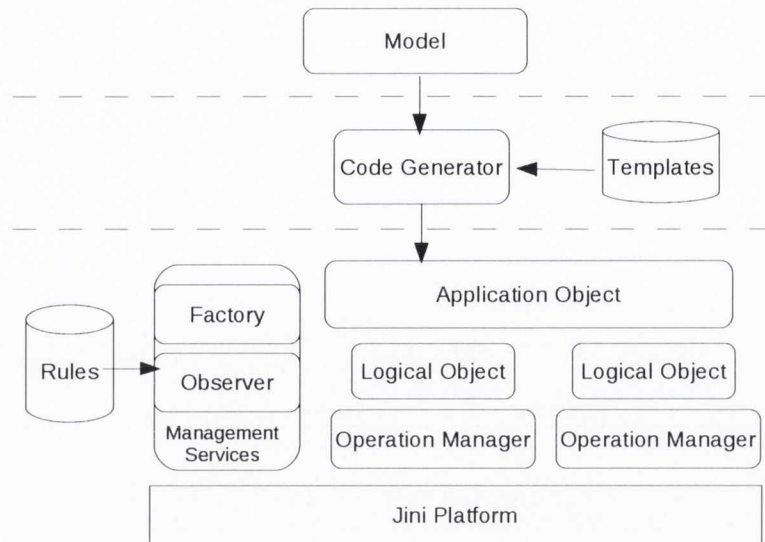


Figure 3.4: Jini-based framework for supporting PAOs.

3.3.3 PervML

Serral et al. (2008) present a model-driven method for developing context-aware pervasive systems. They argue that the complexity of context-aware pervasive computing systems precludes ad-hoc solutions and that a systematic approach to developing such applications is required. Their approach provides a set of models for representing high-level application context, a transformation engine to transform these models into code and a strategy for adapting run-time application behaviour in response to changes in deployment environment.

The approach is centred around the PervML language for modelling pervasive computing applications (Muñoz & Pelechano 2006). The PervML development process shown in Fig 3.5, includes the following steps:

1. Modelling the system using the PervML models.
2. Generating Java code from the PervML models for implementing system services and generating OWL ontologies from user and location models for representing system context data.
3. Implementing or reusing existing OSGi drivers for managing access to devices and services in the area.
4. Deploying the compiled Java code and OSGi drivers into an OSGi server.

The code generated by the transformations is combined with a PervML implementation framework that is integrated with the OSGi middleware² to access entities in the deployment environment.

²<http://www.osgi.org>

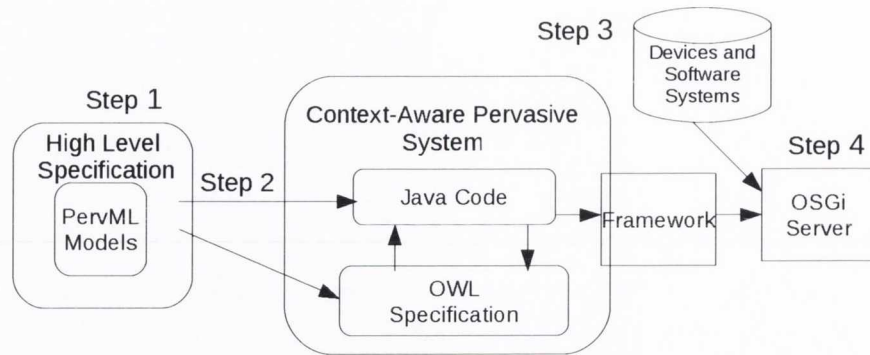


Figure 3.5: PervML development process.

Modelling Support

Pervasive computing environments are modelled using a combination of PervML and UML diagrams. PervML extends the SOUPA ontology for pervasive computing (Chen et al. 2003) to support modelling information about services available in the environment, users and system policies. PervML is constructed as a series of UML component diagrams.

PervML templates are provided for specifying application functionality that abstracts away the heterogeneity of devices and services that exist in pervasive environments.

The PervML development process supports the role of system analyst and system architecture. System analysts specify application functionality using the Services, Structural, and Interaction templates. The Services template describes the services to be provided by the application using pre and post conditions expressed in OCL, a state machine indicating the operations that can be performed at each time-step and triggers that specify proactive behaviour for the application in response to changes in the deployment environment. The Structural template specifies how service instances can be combined to provide an application and specifies the service interface or exposed behaviour of each service. Finally the Interaction template uses UML sequence diagrams to specify communication patterns between services in an application in reaction to events that can occur in a deployment environment.

System architects specify the devices and software systems that support the services specified by system analysts. System architects provide models that bind the application being developed with its physical or logical environment. Architects specify a Binding Providers model for describing sensors and other relevant devices in the environment, a Component Structure model used to assign devices and software systems to application components and a Functional model that specifies the actions executed when the operation of a service is invoked. Actions are specified using the Action Semantics Language (ASL) in UML.

PervML distinguishes between design-time and runtime context information. Design-time context includes information such as user preferences, user-space information, system

services, security and privacy services. Run-time context includes information such as temporal information, user mobility, service state, available services and actions that users perform. Run-time context is used to adapt application behaviour.

Model Transformations

Two model transformation algorithms are provided. The initial transformation parses the PervML model and generates a set of Java classes for implementing the functionality of the specified services. Similar to the PAO approach, models are translated into code skeletons using the Eclipse Java Emitter Template framework³.

The second transformation generates the OWL specification from the extensions to the PervML models. The OWL specification describes the context-aware information relevant to the system functionality. This specification is intended to be used to support the inference of context knowledge at runtime.

Execution Model

The authors provide a PervML implementation framework for executing the generated code. This framework is built over the OSGi middleware. The framework supports common pervasive application functionality and thereby reduces the quantity of code that must be generated by the model transformations. The implementation framework supports the presentation of context-aware information and services to a user.

The three-layered framework is designed using the Layers Architectural pattern (Gamma et al. 1995). A Driver layer manages access to physical devices and external software. Device drivers must be manually written for this layer. A Logical layer contains two sublayers: a Communications sub-layer to provide a high level representation of a device driver, and a Services sub-layer that exposes the functionality provided by software services in the deployment environment. Finally an Interface Layer manages access to applications by clients using the Model-View-Controller (MVC) pattern.

The implementation framework uses the OWL ontology generated from the PervML models to identify relevant run-time context data. Run-time context information is intended to be combined with reasoning and machine learning algorithms to support run-time application behaviour adaption. However this support is at an early stage of implementation and not much detail is currently available.

The authors describe the application of their approach to a smart home scenario but no detailed evaluation is provided.

3.3.4 uPAIN

Machado et al. (2007) describe a MDE approach to requirements engineering for pervasive computing applications. The approach is based on the construction of executable interactive prototypes to support the validation of workflow requirements and is centred around capturing requirements using UML models and transforming these requirements into Coloured Petri-Nets (CPNS) (Jensen 1992). The use of CPNs allows the dynamic

³<http://www.eclipse.org/emf>

3.3. MDE IN PERVASIVE COMPUTING

properties in the system and their interaction in the environment to be examined. They demonstrate the approach using a pervasive healthcare case study in which a health information system (uPAIN) is built to explicitly support the interaction between people in a pervasive workflow.

The case study is concerned with the process of pain control in patients in a hospital who are subject to relatively long periods of pain during post-surgery recovery. The goal of the uPAIN system is to give each patient a PDA which interfaces with a patient controlled analgesiac dispenser and stores electronic patient healthcare records indicating values of pre-determined physiological indicators.

The authors generate interactive animation-based prototypes reflecting system behaviour that allow stakeholders to visualise executable versions of UML Use Case and Sequence diagrams. The UML diagrams are derived from previously elicited requirements statements.

Modelling Support

UML Use Case and Sequence diagrams are used to model system requirements.

Model Transformations

Model transformations are provided to translate Use case and Sequence diagrams into CPN diagrams that can be used to analyse the dynamic properties of the system during its interaction with the environment. The authors present a transformation technique to automate the CPN generation and they use XML as a common data format throughout their toolchain.

Execution Model

The generated CPNs are executed to examine the dynamic properties of the system and validate the workflow requirements. The generated CPNs allow for greater quality control over requirements engineering and do not provide actual functionality for the system.

3.3.5 Review

The work reviewed in this section shows that MDE techniques are gaining in popularity among the pervasive computing community. Baresi & et al (2006) and Serral et al. (2008) recognise the need for a systematic approach to pervasive computing development.

The reviewed work cites a range of motivations for adopting an MDE approach such as supporting code reuse, masking the complexity of designing services and applications for dynamic runtime environments, and providing uniform abstractions over heterogeneous sensing and computing entities.

From a software engineering perspective the approaches are broadly similar and follow a model→transform→generate cycle. The majority of approaches motivate the use of a domain-specific modelling language containing constructs appropriate to describing typical entities found in pervasive computing environments. The degree of automation seems directly proportional to the maturity of the work.

Table 3.1 summarises some properties of the reviewed work. All the systems reviewed

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

offer some limited form of tool support to help application developers. All but one of the systems provide domain-specific modelling (DSM) languages. The MW (middleware) column shows the middleware platforms used by the systems to support device discovery and access at run-time in pervasive computing environments.

The Olympus and PervML approaches motivate the use of optimisation techniques in pervasive computing. However their focus is on optimising the management of physical devices and on adaptive and responsive user tailored services rather than providing planning and optimisation functionality for task-oriented applications.

Model Driven Engineering in Pervasive Computing Environments				
System	Tool Support	DSM	Middleware	Evaluation
Olympus	Partial	✓	Gaia	Qualitative
PAO	Partial	✓	Jini	Qualitative
PervML	✓	✓	OSGi	Qualitative
uPAIN	Partial	X	X	Qualitative

Table 3.1: Summary of reviewed MDE approaches in pervasive computing.

The evaluation of these approaches is problematic. The work is generally immature and evaluations are qualitative. The qualitative evaluations are brief and provide no evidence on the usefulness of the approaches.

3.4 Planning and Optimisation in Pervasive Computing Environments

This section reviews the application of planning and optimisation algorithms to pervasive computing applications. A definition of planning and optimisation is provided in Chapter 2. The focus of this review is on the selection and use of planning and optimisation algorithms in pervasive computing environments. For each research project reviewed in this section, the following questions are considered.

- How are problems modelled?
- How is an appropriate algorithm selected?
- How is the application executed?

3.4.1 GPT

The General Planning Tool (GPT) is an automated planning tool for modelling, analyzing and solving planning problems characterised by uncertainty and partial information (Bonet & Geffner 2001). The range of planning problems that the GPT tool can solve is a function of transition and observation model properties. GPT can model deterministic or probabilistic state transition systems, and can represent sensor models that are complete (corresponding

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

to full observability), partial or null (corresponding to uncertain sensor data or to a blind controller with no sensor data).

GPT provides a predicate logic language for expressing actions, sensors and goals, and a bundle of algorithms based on heuristic search, for solving them. The problem description is solved by an algorithm chosen from the bundle. GPT produces plans that specify state-action combinations. GPT uses two classes of algorithms for solving problems. The first is an optimal heuristic A* search algorithm used for conformant planning and the second is a dynamic programming algorithm called RTDP that is used for non-deterministic, probabilistic planning.

Conformant planning techniques are used for planning when it is assumed that the agent will have no run-time observability. A conformant plan is a sequence of actions guaranteed to achieve a goal state regardless of what events occur in the environment. A conformant plan does not have to be optimal. GPT uses A* for conformant planning with a domain-independent heuristic derived from a relaxed or simplified problem version where full observability is assumed. An optimal policy is calculated for the simplified problem and used as a heuristic input into the A* algorithm.

For problems with partial observability, GPT uses RTDP (a real-time heuristic search algorithm) to calculate a partial solution or policy for a set of states that may be visited with positive probability by the optimal policy. A stopping rule is used to guarantee convergence to an optimal policy with a defined error ϵ . The RTDP algorithm is used when the agent has partial observability of the system state.

Modelling Language

GPT uses a typed logical language similar in syntax to the Planning Domain Definition Language (PDDL) (Ghallab et al. 1998), and which extends the STRIPS language (Weld et al. 1998). The GPT language allow system states to be modelled using predicate values in first-order logic. Probabilities, costs and an observation function can be associated with these fluents.

A planning problem is defined in two sections, the first is a description of the domain and the second is a description of an instance of a problem in this domain. Listing 3.2 shows a complete definition for a planning problem to drive a car from a range of possible starting positions to a goal location.

A domain definition begins with the `models` keyword classifying the state-transition dynamics and the observability of the problem. In line 2 of Listing 3.2, the racetrack domain is defined as having probabilistic dynamics, i.e., a probabilistic state-transition system and complete feedback, i.e., a fully observable state space. Lines 4-6 define the racetrack as a grid of 35×12 positions. The state space consists of 4-tuples (x, y, dx, dy) , where x and y are the coordinates of the car and dx and dy are the velocity of the car in each dimension. Lines 17-28 define the possible actions as pairs (ax, ay) of instantaneous accelerations where $ax, ay \in \{-1, 0, 1\}$. The actions have a 0.9 probability of success and 0.1 probability of failure. The transition function is specified by projecting state-action combinations to new states as: $set(s, a) = (x + dx + ax, y + dy + ay, dx + ax, dy + ay)$ Lines

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

```
1 (define (domain racetrack)
2   (:model (:dynamics :probabilistic) (:feedback :complete))
3   (:types STATE)
4   (:functions (x STATE :integer[0,34])
5               (y STATE :integer[0,11])
6               (dx STATE :integer[-5,5])
7               (dy STATE :integer[-5,5]))
8   (:external valid)
9   (:objects state lastState - STATE)
10
11   (:axiom constraints
12     :effect (:when (=valid state) false)
13             (:set state lastState)
14             (:set (dx state) 0)
15             (:set (dy state) 0)))
16
17   (:action control
18     :parameters ?ax ?ay - :integer[-1,1]
19     :effect
20       (:probabilistic
21         (0.9 (:set lastState state
22                 (:set (x state) (+ (x state) (+ (dx state) ?ax)))
23                 (:set (y state) (+ (y state) (+ (dy state) ?ay)))
24                 (:set (dx state) (+ (dx state) ?ax))
25                 (:set (dy state) (+ (dy state) ?ay))))
26         (0.1 (:set lastState state
27                 (:set (x state) (+ (x state) (dx state)))
28                 (:set (y state) (+ (y state) (dy state)))))))
29
30 (define (problem race)
31   (:domain racetrack)
32   (:init (:set (dx state) 0)
33         (:set (dy state) 0)
34         (:set (x state) 0)
35         (:set (y state) :in { 5 6 7 8 })))
36   (:goal (:and (= (y state) 0)
37              (:in (x state) { 32 33 34 }))))
```

Listing 3.2: Complete GPT model of a racetrack planning problem.

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

36-37 specify the goal state as a location the car should stop in. The location is specified as having valid x values [32,33,34] and a y value [0].

Algorithm Selection and Usage

Algorithm selection is dependent on the information in the model. If an observation model is specified then the agent will choose the RTDP algorithm which allows it to use the sensor feedback to guide action selection. If the feedback semantics are null, corresponding to a blind controller then the agent will choose the A* algorithm.

Execution Model

The use of GPT consists of a design→compile→solve→analyze cycle. The first step refers to the modelling of the problem in the GPT language. This description is then compiled into C++ data structures and object code. In the solve phase, GPT calls the appropriate solver, A* or RTDP, depending on the input problem characteristics. The output is a policy or plan specified in a graph representation, governing action selection.

The output can then be analyzed qualitatively and quantitatively. The qualitative evaluation can be done by inspecting the resulting policy and a quantitative evaluation can be done by analysing log data to determine solution costs and value function convergence.

The authors have also created an online version of GPT called mGPT, that was used in the probabilistic planning track of the 4th International Planning Competition (Bonet & Geffner 2005). mGPT is an online planner that is designed with a client/server architecture. The mGPT planner acts as a client and connects to the server that maintains an instance of the problem domain. The mGPT planner interacts with the server by exchanging messages. Each interaction cycle consists of actions decided by the planner, the effects of which are transmitted back from the server to the planner.

The GPT tool passes the problem description to an instance of the algorithm and a solution or a plan is returned to the user. This plan is usually structured as a graph and contains state-action mappings that can be used by a developer or planner. GPT is most suitable for specifying well-understood static planning problems.

GPT provides a consistent and unified framework for specifying planning tasks. The user is not required to write or understand planning algorithms beyond the knowledge of specifying the problem type. Tool support is provided in the form of a compiler and analyzer to determine the efficiency of plans.

GPT and mGPT are exemplars of the automated-planning community approach to planning as distinct from the pervasive computing or systems community approach. The GPT model as shown in Listing 3.2 contains no high-level abstractions for modelling sensors, actuators, or application state or context. There is no consideration of how to use the GPT tool to connect elements from the problem description to middleware services. The automated-planning community is primarily concerned with the theoretical solutions to problems and not with the engineering and deployment of planning and optimisation algorithms. This focus is reflected in the fact that the GPT tool is not designed to produce code implementing application functionality, but rather produces policies mapping state to action selection.

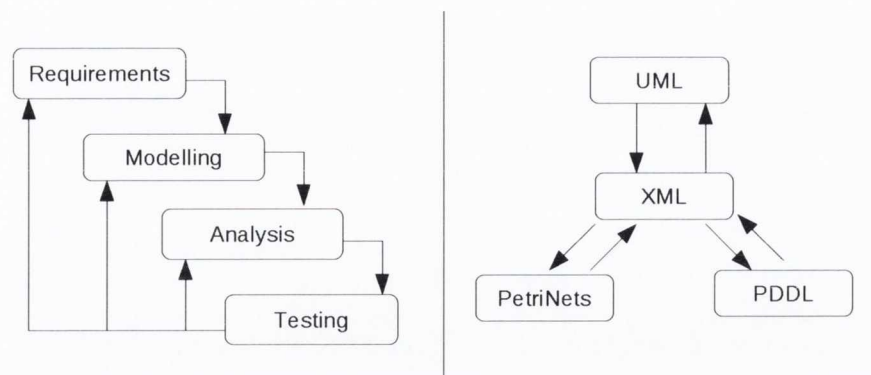


Figure 3.6: The itSimple design process and integrated languages.

3.4.2 itSimple2.0

itSimple (Integrated Tools Software Interface for Modelling Planning Environment) is a development environment toolchain for constructing planning applications. itSimple is designed to integrate a traditional software engineering approach with modelling experiences acquired in the field of automated planning.

itSimple is specifically focused on supporting the early stages of a standard software design life cycle (Vaquero et al. 2007). The supported lifecycle stages shown in Fig. 3.6 include gathering, modelling, analysis, and testing of the planning requirements of an application.

itSimple provides modelling languages for specifying planning requirements and defines transformations for the modelling languages so that they can be fed into standard automated planning tools. itSimple also provides a graphical interface for analysing plans produced by standard automated planners in which users can observe the behaviour of the planning model during plan execution.

Modelling Language

As shown in Fig 3.6, itSimple uses UML, Petri-Nets and PDDL to model planning problems at different software lifecycle stages. UML is used for requirements gathering, specification and modelling. Petri-Nets is a graph-based formalism that allows dynamic features of a planning domain such as deadlocks and concurrency issues to be tested. PDDL is a standard for describing a planning problems, and PDDL problem descriptions can be fed into a range of automated planners.

The Petri Nets Markup Language (PNML) and the eXtensible Planning Domain Domain Language (XPDDL) have a direct representation in XML. XML is therefore used to transform UML to Petri-Nets and from UML to PDDL.

In previous work, the authors have developed UML.P (Vaquero et al. 2006), an extended version of UML with added support for specifying planning domains. UML.P structures planning problems as shown in Fig. 3.7. Domains have a name and description, planning problems are associated with a domain and have constraints, initial conditions and goal

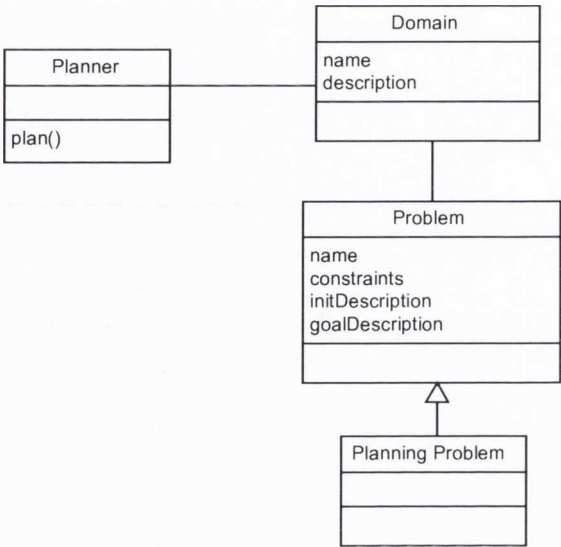


Figure 3.7: itSimple schema for planning environment.

descriptions. Planners are associated with problems and domain descriptions.

Use case diagrams model planning requirements and contain requirement descriptions, pre and post conditions, and constraints. The Object Constraint Language (OCL) is used to express constraints. itSimple automatically generates documentation from use case diagrams which can be used to validate the requirements elicitation process.

UML class and state chart diagrams are also used to specifying planning problems. The static features of a problem are defined using classes, attributes, associations and structural constraints expressed in OCL. State chart diagrams are used to represent dynamic features of the problem. A state chart diagram is defined for each class in the system that exhibits dynamic behaviour. The set of state charts in a UML.P domain model represents a set of regions in the state space. The user can specify pre and post conditions of actions using OCL.

UML diagrams provide a static view of a planning problem. In order to provide a means to analyse the dynamic properties and consistency of the planning problem, itSimple defines a transformation from state chart diagrams to Petri Nets. The use of Petri Nets allows users to formally analyze the dynamic features of a planning domain such as deadlocks, invariants, parallelism and concurrency. This transformation extracts types, predicates and functions from the class diagram and actions as well as pre and post conditions are extracted from the state charts.

itSimple also defines a transformation from UML.P to PDDL to allow plans to be created for planning problems. itSimple uses Object diagrams (D'Souza & Wills 1999) to represent the initial and goal states of the planning problem.. An object diagram is a depiction of the set of objects and the values of some or all of their attributes at a particular point in time. The UML.P and object diagrams are transformed into a PDDL description of the

problem can be fed into a number of existing planners.

Algorithm Selection and Usage

The issue of algorithm selection is not directly addressed in itSimple. The problem description is transformed into a PDDL planning problem specification and can be processed by any planner that supports the PDDL language. The supported development process is independent of the planner used to generate a plan. The class of problems that can be solved are limited to those that are supported by the PDDL language. These are static or offline problem descriptions that assume full observability over the system state space.

Execution Model

The use of itSimple consists of a model→analyze→test→cycle. itSimple provides a transformation process to translate domain and planning problem requirements data into formats appropriate to the development stage. The generated plans can be analysed to identify constraint violation throughout the execution of the plan.

The authors present a case study of using the itSimple tool to optimise a crude oil supply chain in Brazil (Vaquero et al. 2008). The plan must guarantee that at all moments the amount of oil in the refineries stays above a minimum level while minimising the cost of distribution. No reliable and robust algorithm exists for this problem in the research literature (Li et al. 2006). Three real world planners were used for this case study: Metric-FF, SGPlan and MIPS-XXL. These planners were unable to provide successful solutions to this problem. The authors summarise the case study by concluding that while the itSimple domain models can represent complex domains with many numeric variables, the current state of the art in automated planners does not appear to be sufficient to address many real world planning problems.

The authors state that a motivating challenge for them is to provide a platform that can integrate the expertise of the planning community with the design skills of the software engineering community. The use of UML statechart and activity diagrams to represent system states provides an expressive means to represent state information. However the authors do not address the modelling of sensors and actuators and do not consider the problem of mapping domain model elements to physical infrastructure in the planning environment.

Similarly to the GPT tool, the output from the itSimple tool is a policy or plan that specifies a sequence of actions. No application code is generated by the itSimple tool and no consideration is given to deploying and executing planning algorithms in real world environments.

3.4.3 Planning in Gaia

Ranganathan (2004) propose a planning approach for supporting autonomic computing in pervasive computing environments. The Gaia planning framework is designed to allow users to specify their goals in an abstract and high-level manner and uses planning techniques to decide how best to achieve these goals.

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

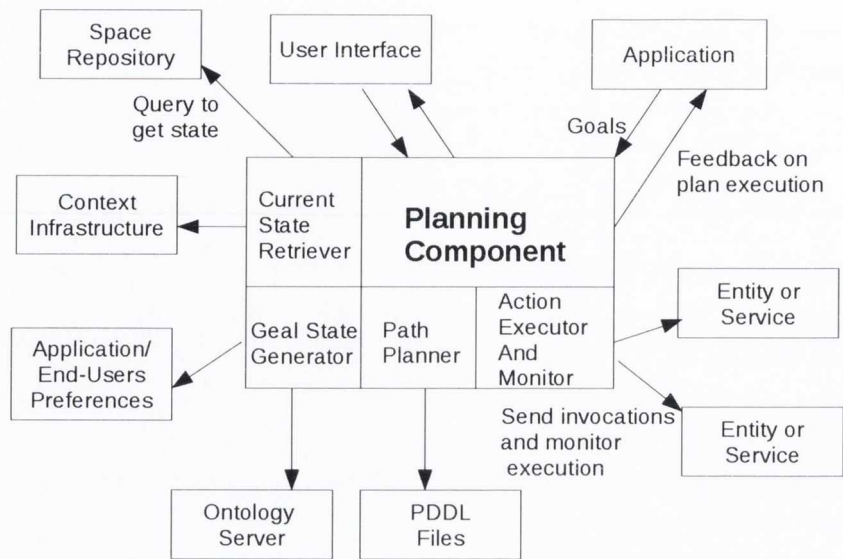


Figure 3.8: The Gaia Planning Framework

The planning framework is based on a predicate model of pervasive computing where the state of an environment is represented as a set of predicates and the actions that can be performed in the environment correspond to invocations of a service, device or application. A utility function is used to determine actions to be selected in accordance with specified goals.

The authors claim their framework provides the following features:

- Improved usability of devices in pervasive computing environments by allowing users to specify their goals in an abstract manner, e.g., display presentation.
- Improved ease of programming pervasive computing applications by allowing developers to describe the tasks to be performed in terms of abstract goals.
- Improved fault-tolerance through monitoring and replanning in the case of device failure.
- Automatic configuration of pervasive computing environments to user preferences and context.
- Adaptability to different configurations and environments. The framework uses a service discovery component to discover devices and services in the environment. This coupled with the high-level expression of user goals allows the framework to devise a plan in a variety of environments.

The architecture of the planning framework are shown in Fig. 3.8. The planning component consists of four modules which are implemented using services provided by the Gaia middleware (Román et al. 2002).

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

1. A `Current State Retriever` obtains the system state using Gaia middleware services to discover and query state information from entities in the deployment environment. Entity discovery and query support are provided by the Gaia Context Infrastructure service (Ranganathan & Campbell 2003).
2. A `Goal State Generator` generates possible states of the system that correspond to predicate of the system and chooses an optimal state using preference information, access control policies and context information.
3. The `Path Planner` searches offline for a sequence of actions to bring the environment to the chosen goal state using action information from PDDL specifications and a STRIPS blackbox planner (Kautz & Selman 1999).
4. The `Action Executor and Monitor` executes actions in the environment by invoking methods on objects and application services.

The planning functionality is provided by a STRIPS planner. The user specified goal state and current state are fed into the planner, which then searches for and returns a path between the two states using the actions available to it. If no path is found then the user is informed that the goal is not satisfiable.

Modelling Language

The Gaia planning framework uses a number of languages State or context information is represented as a set of predicates that constitutes the possible states of a device, service or application. This information is modelled using an ontology written in DAML+OIL (McGuinness et al. 2002). A sample piece of location context information is provided: `location (chris, inside, room 3231)`. This excerpt contains a subject, predicate, and value combination identifying that a user called chris, is inside room number 3231.

The actions available to the planning system are modelled in PDDL. Each entity in Gaia has a PDDL specification describing the actions it can perform. Preconditions and effects are specified for each action. An action can also be marked as retryable, in which case, the planning system can try to execute the action again if it failed the first time.

Listing 3.3 shows the PDDL description for a `setFile` action for the Microsoft PowerPoint application. In line 2, parameters are specified for id of the application and the name of the file to be displayed. Lines 5-8 contain a precondition specifying that an instance of the Powerpoint application should be currently running on a machine and that the input file should not already be displayed on another machine. Lines 10-13 specify that the effect of the action is that a PowerPoint application now displays the input file and is on slide 1. Finally in line 15 the `setFile` action is marked as non-retryable in the event of failure.

Each state of the environment is associated with a utility that is user or application specific. The utility of the entire environment is expressed in predicate form and multiple utilities are calculated as a linear combination of the individual utilities of the states of all services, applications and devices. Utilities are modelled as integers in the range [-10,10].

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

```
1 (:action setFile-ppt
2   :parameters (?id - string ? filename - pptfile)
3   :vars (?mac - machine ? slidenumber - number)
4
5   :precondition
6     (and (pptModel >id ?mac started
7           ?filename1 ?slidenumber)
8           (not (= ?filename ?filename1)))
9
10  :effect
11    (pptModel ?id ?mac started
12      ?filename 1)
13  :check
14    (= (getFile-ppt ?id) ?filename)
15    (failure non-retryable))
```

Listing 3.3: Gaia PDDL.

```
goal(display-ppt, File) :- coordinator (
appID, pptApp, M1, initialized),
pptModel(appID, M2, started, File, 1),
presentation(appID, pres1, PresentationType, M3),
controller (appID, cont1, ControllerType, M4)
```

Listing 3.4: Prolog.

System goals are represented in predicate form. Users specify goals using through a user interface tool. that generates templates describing the goal state. These templates are represented as predicates with variable valued arguments. Users must specify rules for attaining goal states. Listing 3.4 shows a sample rule written in Prolog that states that in order to display a PowerPoint file the PowerPoint application must be initialized in a machine M1 and a PowerPoint application started in machine M2 showing slide number 1 of the file. Template states are represented as predicates with variable arguments. In this sample M1 and M2 are variable names that can be resolved by the planning algorithm to entities in the deployment environment.

Algorithm Selection and Usage

The framework does not support algorithm selection or engineering. The planning component passes the current environment state, the template goal state and the PDDL action descriptions to a STRIPS Blackbox planner (Kautz & Selman 1999) which uses graph-based planning to generate a plan.

The plan is specified as a series of actions to take to transition from the initial state to a state matching the goal template state.

Execution Model.

The planning framework is heavily dependent on support from the Gaia middleware. The framework uses a service registry provided by the Gaia middleware to obtain the current state of the system. All services found by the registry are queried for their current state. The framework then invokes the STRIPS planner to generate a plan to transition from

the current state to a template state that matches the user-specified goal. Once the plan is generated the framework supports action execution and monitoring. The framework executes actions by calling methods on various objects and monitoring them for failure. In the event of an action failure the planning framework can choose to retry the action or to reinvoke the planner to generate a new plan.

The authors have implemented a prototype but the evaluation is limited (Ranganathan 2004). They state that they have used the planning framework to simplify the bootstrapping of their Gaia middleware and to manage some applications in a smart room. No detailed information is provided on this. The example cited in (Ranganathan 2004) involves selecting the most appropriate audio-visual device to display a presentation. The authors provide a measure of 30% overhead in running time for using the planning framework to determine which service to invoke over using a static script to invoke the service.

The predicate based modelling language used in the planning framework, does not contain any constructs appropriate to modelling sensors or actuators and does not address the issue of uncertain information. However by monitoring plan execution and replanning in the event of failure it does provide some support for planning in dynamic environments.

3.4.4 Smart Bay

O'Hare et al. (2007) present an approach to embed intelligent decision making within complex and dynamic environments by combining stream based computing with collaborative multi-agent system techniques. They evaluate their approach through an demonstrator project entitled SmartBay. This work focuses the deployment of intelligent applications on devices of limited computational capacity such as mobile phones, sensors and embedded computational artifacts.

The Smart Bay project uses System S, a stream-based middleware platform used in distributed systems that experience high load and dynamic input. System S is used to integrate physical world of measurements and data on actions (sensors/transducers and actuators of various sorts) into computing systems. System S provides distributed analytical functions ranging from filtering to AI based analytics and complex modelling (Jain et al. 2006), that allow intelligent management of information flows in a distributed real world application.

SmartBay uses a multi-agent-system (MAS) to facilitate collaborative decision making. The MAS is based on the Belief, Desire, Interaction (BDI) model of agents (Bratman 1987). Within the BDI model, beliefs represent possibly inaccurate information that an agent has of the world at a given instance in time. Desires or goals represent a state of the world that an agent wishes to bring about. The agent must fix upon a subset of desires and commit resources to achieving them. Traditionally the BDI model was considered too computationally intensive for resource constrained mobile devices, however the authors point to a new generation of sensor nodes with greater processing and memory capabilities such as Sun SPOT devices that support restricted Java Virtual Machines. The agent-based com-

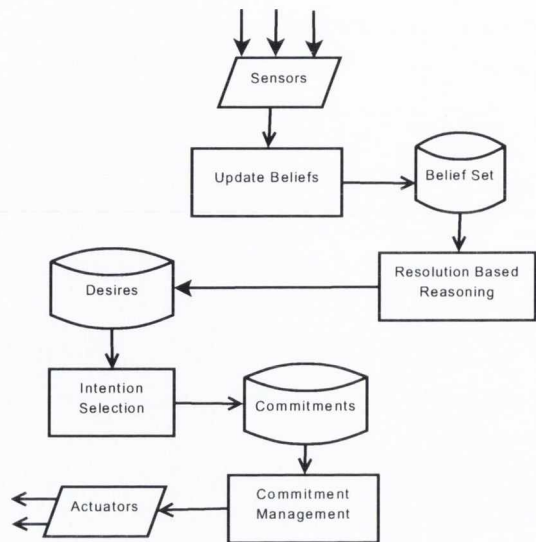


Figure 3.9: AFME control process.

ponent of Smart Bay has been developed using the Agent Factory Micro Edition (AFME), which supports the BDI framework and which runs on Sun SPOT nodes (Ma et al. 2006). AFME uses the Constrained Limited Device Configuration (CLDC) Java standard for mobile phones and sensor devices.

AFME is an agent platform that has been designed for use in resource-constrained ubiquitous computing devices. Fig. 3.9 shows a standard AFME process containing four functions. First, sensors monitoring environment state are invoked and the agent’s belief set is updated. Second, the agent’s desired states are identified. Third, a subset of commitments or actions are chosen and finally a set of actuators associated with the desired states are invoked.

Modelling Language

There does not appear to be a significant data or domain modelling component to this research work. The System S middleware uses XML to represent the configuration of the stream connections between processing elements. The AFME logic is encoded in a rule-based logic developed by the authors. Rules define the conditions under which BDI commitments will be desired and/or adopted and retracted. These rules govern and encode agent behaviour and are expressed in the form:

```
a(?variable1), b(?variable2) > doSomething(?variable1, ?variable2);
```

If the agent adopts beliefs that match the a and b predicates, a commitment to doSomething is adopted.

Algorithm Selection and Usage

Automation of algorithm selection or generation is not addressed in this work. The BDI model of agents used in the work is rule-based and does not incorporate planning or

3.4. PLANNING AND OPTIMISATION IN PERVASIVE COMPUTING ENVIRONMENTS

optimisation algorithms. Decision-making in SmartBay ranges from the local or node-based decisions to collaborative decision-making across a set of nodes. The nodes cooperate to optimise system performance and rules are provided that govern the following metrics:

- Longevity. Decisions are taken to conserve the limited power resources of the nodes.
- Sample quality. Decisions are taken which affect sensor accuracy, for example whether or not to flush a sensor prior to taking a reading.
- Sampling and transmission frequency.
- Analytics. Decisions are taken about what analysis to apply to the data but also about where the analysis should be done, i.e., locally on the node or remotely using higher-level System S processing. Such decisions are governed using a rule base specified offline.

Executable Model

The authors have used their approach in a real-time oceanographic monitoring system run by the Marine Institute of Ireland. The monitoring system comprises a network of buoys linked by both fixed and wireless communication technologies. Each sensor incorporates a range of sensors for measuring phosphate levels. The sensor data is used to measure the effects of climate change.

The decision making component of the Smart Bay system aims to improve longevity, maintain sample quality through ensuring the correct functioning of sensors, determine the most energy efficient sampling and transmission frequency. The system also decides what data analysis is to be done and where the analysis should be carried out. These activities are executed in accordance with system policy encoded in a rule base. The rule-based decision making is aimed at managing low-level system performance rather than supporting planning and optimisation of application state.

3.4.5 Review

The work reviewed in this section shows that systematic support for integrating planning and optimisation algorithms into pervasive computing environments is at an early stage.

Table 3.2 summarises some properties of the reviewed work. Column two indicates whether some form of algorithm selection is supported. GPT is the only system that interrogates the problem description to choose one of two possible algorithm types. The other systems apply one algorithm to all problem descriptions. Column three shows that only GPT supports multiple planning and optimisation algorithms. All but one of the systems reviewed provides a modelling language for specifying problems. GPT and Gaia provide predicate logic languages for specifying problems. However these languages do not contain any high-level abstractions that map to entities in pervasive computing domains. The itSimple project provides high-level modelling support using a modified UML language.

3.5. CONCLUSIONS

GPT and itSimple do not address the issue of middleware use or integration. This reflects their origin in the automated planning community, whose focus is more on theoretical planning rather than the engineering and deployment of planning and optimisation algorithms. GPT and itSimple provide a mix of quantitative and qualitative evaluations. Gaia and SmartBay only offer a limited qualitative evaluation. These evaluations are limited in detail and provide no guidance on the usefulness of the approach.

Planning and Optimisation in Pervasive Computing Environments					
System	Alg. Select.	Mul. Alg.	DSM	MW	Evaluation
GPT	✓	✓	✓	X	Qual. + Quant.
itSimple	X	X	✓	X	Qual + Quant.
Gaia Planning	X	X	✓	Gaia	Qualitative
SmartBay	X	X	X	System S	Qualitative

Table 3.2: Summary of reviewed planning and optimisation approaches in pervasive computing.

The automated planning community are focused on solving a planning problem. The endpoint for GPT and itSimple is a plan rather than a planning application. Generated plans are structured as policies that provide a mapping from state to action. Gaia and SmartBay originate in the pervasive computing community, which generally does not tackle application level planning or optimisation but focuses on challenges relating to service selection or efficient resource utilisation. Reflecting their origins, Gaia and SmartBay are aware of the need to provide middleware integration.

3.5 Conclusions

This chapter has reviewed work from the pervasive computing and automated planning community. This reflects the multi-disciplinary challenges of developing applications using planning and optimisation algorithms in pervasive computing environments.

The review has established that there is an interest in applying model-driven techniques in pervasive computing environments for purposes such as, managing heterogeneity of devices, masking the complexity of dynamic environments, and promoting code reuse. The review has also examined the work in the automated planning community that relates to real world planning problems. Systems such as GPT and mGPT allow partially observable and dynamic planning problems to be modelled and solved at a theoretical level.

This thesis proposes a methodology that synthesises concepts from the fields of MDE and automated planning. Chapter 1 described the following requirements identified for the methodology.

- R1 - Allow application logic (policy) to be specified at a high-level.
- R2 - Provide abstractions masking run-time conditions of partial observability and

3.5. CONCLUSIONS

uncertain application state.

- R3 - Provide a high-level API to developers of planning and optimisation algorithms for pervasive computing environments.
- R4 - Provide a software engineering model that is extensible; that promotes reuse; and is suitable for use in large-scale applications.
- R5 - Rapidly test algorithm efficacy.
- R6 - Provide a consistent representation of uncertain data.
- R7 - Automatic selection of appropriate planning and optimisation algorithms.

Related work and Methodology Requirements							
System	R1	R2	R3	R4	R5	R6	R7
Olympus	✓	✓	X	X	X	X	X
PAO	✓	✓	X	✓	X	X	X
PervML	✓	✓	X	✓	X	X	X
uPAIN	✓	X	X	X	X	X	X
GPT	✓	X	X	X	✓	✓	✓
itSimple	✓	X	X	X	✓	X	X
Gaia Planning	✓	✓	X	X	X	X	X
SmartBay	✓	✓	X	X	X	X	X

Table 3.3: Related work and methodology requirements.

Table 3.3 shows how the reviewed systems conform to the methodology requirements. The MDE and Gaia approaches shown in rows 3-6 and 9 respectively, largely address the requirements R1, R2, and R4. These are requirements closely related to engineering applications and hence are generally addressed by the pervasive computing community. However the support provided in the form of automated transformations and code generation is generally immature. Also the modelling languages used by these approaches are often predicate based and non-intuitive to use. We note that no work is identified relating to R3, i.e., facilitating the development of planning and optimisation algorithms for pervasive computing environments.

The automated planning approaches shown in rows 7 and 8 of Table 3.3 address requirements R5-R7. The GPT system offers mature support for testing and evaluating a limited set of planning algorithms. However it does not address the engineering requirements R2-R4.

A differentiating factor between the methodology presented in this thesis and the work of the automated planning community, is that our aim in to generate application code instead of plans. The generated code will integrate planning and optimisation algorithms with domain model data and contain hooks to assumed middleware services. When executed, the code is expected to provide application functionality as specified using the

3.5. CONCLUSIONS

methodology domain model and policy. The proposed methodology is intended to complement, rather than compete with, automated planning approaches.

In order to generate executable units we assume that a pervasive computing middleware exists to provide the services needed to support the applications. Middleware assumptions are specified in Chapter 1.

The evaluation strategy used in the related work is shown in Tables 3.1 and 3.2. The MDE approaches rely exclusively on qualitative evaluation techniques while the automated planning approaches use a combination of qualitative and quantitative evaluation criteria. The review indicates that an inter-disciplinary approach is necessary for engineering planning and optimisation algorithms for pervasive computing applications and that substantial work remains to be done to support the development of such applications. Chapters 4 and 5 present the development process and tool support provided by our methodology for this purpose.

— The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence. For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.

Fred Brooks, "No Silver Bullet", 1986.

4

Development Process

This chapter presents the development process supported by the methodology. It describes how applications are programmed using the methodology and focuses in particular on the role of the domain modeller in using the methodology to develop applications.

The presentation in this chapter assumes that the algorithm library is populated with a selection of planning and optimisation algorithms. Chapter 5 discusses how new algorithms are added to the library.

This chapter is structured as follows. Section 4.1 reviews the requirements identified in Chapter 2. These requirements were determined by examining the use of planning and optimisation algorithms, and sensor fusion and inference techniques in pervasive computing environments. Section 4.2 presents the major design decisions derived from the requirements. Section 4.3 presents an overview of the application development process using the tool chain that supports the methodology. The domain modeller is primarily required to specify domain models for applications. The architecture of the domain model is described in Section 4.4. Section 4.5 describes a topological abstraction used to enable domain modellers to program regions of the deployment environment. Section 4.6 presents the detailed design of the domain model layers and describes that type of information that is specified at each layer. Section 4.7 describes how the domain modeller can control the behaviour of an application by specifying application policies. Finally Section 4.8 presents two extensive case studies showing how the domain model is used to program large-scale pervasive computing applications.

4.1 Requirements Review

The methodology has been designed to support developers in the task of developing pervasive computing applications that use planning and optimisation algorithms. This is a multi-stage task that involves specifying application logic, sensor and actuator infrastructure, and the design and implementation of planning and optimisation algorithms.

The following requirements are described in Section 1.5.

- R1 - Allow application logic (policy) to be specified at a high-level.
- R3 - Provide a high-level API to developers of planning and optimisation algorithms for pervasive computing environments.
- R4 - Provide a software engineering model that is extensible; that promotes reuse; and is suitable for use in large-scale applications.
- R5 - Rapidly test algorithm efficacy.
- R6 - Provide a consistent representation of uncertain data.
- R7 - Automatic selection of appropriate planning and optimisation algorithms.

4.2 Design Decisions

The requirements R1-R7 motivate the design of the methodology. The methodology has been designed to reduce the development effort associated with the target class of applications and to systematise the use of planning and optimisation techniques in pervasive computing applications.

The methodology is built around a set of tools provided to domain and AI modellers. The tool-chain automates the development process where deemed possible and where it is not possible to automate, clearly defined points are provided to integrate logic. An example of this is the provision of automated competitive fusion and the manual specification of decision-level fusion.

These requirements are addressed by the following design decisions:

- D1 - Domain Model and Policy Specification.

The domain modeller uses the methodology through the domain model. A set of schemas are provided to a domain modeller and used to define the application state space in XML. A schema is also provided to specify a policy to control the behaviour and deployment region of applications. A reward model is generated from the policy and used to direct the behaviour of the planning and optimisation algorithms. By editing the policy file the domain modeller can focus on application logic exclusively. If the algorithm taxonomy can match an appropriate instance of a planning or optimisation algorithm then the application code can be generated automatically

once the policy and domain model are specified. A policy is specified with a scope that indicates the physical region of the environment over which an application is to be deployed. The scope is a spatial value that can be specified manually or can be specified relative to spatial values associated with domain model elements. The scope can be used to control and alter the deployment region of applications.

- D2 - Topological Abstraction.

A topological abstraction is used to allow a region of the environment to be programmed independently of the run-time conditions in that region. Using the policy specification, the domain modeller specifies desired application behaviour in the deployment environment. Components generated by the transformation engine, use spatial queries to resolve entities such as sensors, actuators, and application state values to regions of the environment at run-time to implement application policy. The topological abstraction underlies the programming model used in the methodology and mandates that model elements have spatial attributes.

Spatial queries are used to map sensor data quality to estimates of the certainty of application state. The requirement to provide estimates of the changing certainty of application state data at run-time led to a design choice to represent state information as variables. Each instance of state data is represented as a state-variable object and is also tagged with a spatial attribute indicating the region of the environment to which it relates. State-variable objects provide an interface for the developer of planning and optimisation algorithms to program against and return an estimate of the value of an application state at a point in time and the uncertainty associated with this estimate.

- D3 - Planning Model.

Planning and optimisation algorithms are programmed to a planning model interface that abstracts away from the environment. Planning-model components provide an API to algorithm developers providing access to estimates of application state and associated uncertainty, reward models and available actions.

High-level support for developing optimisation algorithms is provided in the form of state-generator objects used to generate and evaluate candidate solutions to heuristic optimisation algorithms.

- D4 - Automated Code Generation.

The methodology provides tools for automatic code generation. Instances of planning and optimisation algorithms can be generated and applied to planning model data. The domain model is designed to be extensible and scalable. The topological abstraction allows model elements to be dynamically added and combined thereby allowing large data sets to be created and handled, providing scalability and extensibility. For example, a policy file may specify that a set of states measuring traffic-flow, are to be optimised at all traffic junctions in a region. If new junctions are added

4.3. USING THE METHODOLOGY

to the domain model then the tools for automated code generation can be used to extend the application.

The planning and optimisation algorithms are programmed to an interface provided by the planning model. The planning model provides an algorithm-independent representation of the domain model data. Hence, algorithms can be reused across multiple applications.

- D5 - Domain Specific Knowledge.

The domain model incorporates a domain-specific knowledge layer. This layer can be used to specify which planning and optimisation algorithm to use and to specify values for the free parameters associated with each algorithm. Once the algorithm name and parameter values are specified, the automated code generation tools can be used to create an instance of the application. The performance of these instances can be evaluated and tested.

The possibility of executing models in a simulation environment allows for early experimentation of the system under development to analyse high-risk aspects or alternative solutions.

- D6 - Likelihood Functions.

State-variable objects return likelihood functions that provide an estimate of the value of an application state and that quantify the uncertainty associated with the estimate. Likelihood functions can be used to represent both discrete and continuous data. They can also be combined with any available prior probabilities for state estimates. This combination of prior probabilities and likelihood functions is compatible with a Bayesian inference framework.

- D7 - Algorithm Taxonomy.

A taxonomy mapping problem types and environment properties to algorithms is provided. Domain or AI modellers can manually specify the planning or optimisation algorithm to be used in layer 4 of the domain model. If no algorithm type is specified, the algorithm taxonomy is used by the planning model transformation to select and instantiate a planning or optimisation algorithm. The mapping between problem type and algorithm choice is based on a survey of the literature and empirical observations of algorithm performance. The taxonomy is specified in XML and the selection logic can be easily amended.

4.3 Using the Methodology

In order to systematise the development of the target class of applications the following methodology has been developed. Fig 4.1 shows a data flow diagram for the methodology. The methodology accommodates two development roles:

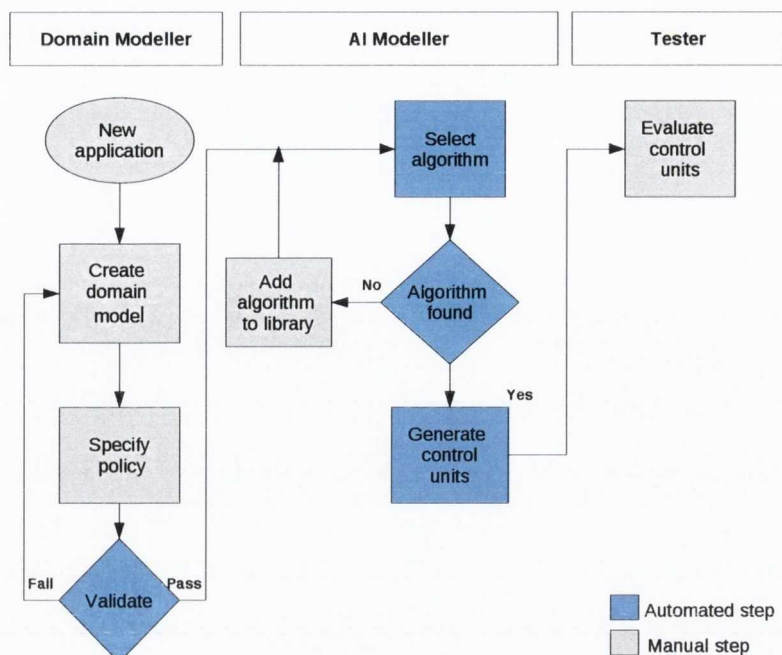


Figure 4.1: Methodology development process.

- A domain modeller creates domain models and specifies the application policy.
- An AI modeller adds new planning and optimisation algorithms to the library and defines mappings from planning model to algorithm components.

The following development process is used to provide planning or optimisation functionality to an application:

1. The domain modeller constructs a domain model using XML schemas provided. The schema formats are described in Section 4.6.
2. The domain modeller writes a policy file specifying the desired behaviour of the application. The policy file format is described in Section 4.7.
3. The first transformation algorithm is then used to validate the domain model and policy file to ensure that they are correctly formatted. If they are valid then the transformation algorithm will generate a planning model.

The first transformation algorithm supports the following commands:

- "python transform1.py -v model-name" - validate the domain model and policy.
 - "python transform1.py -t model-name" - transform the domain model to a planning model.
4. The next step in using the methodology is to choose an appropriate planning or optimisation algorithm. An AI modeller can manually specify with algorithm they

4.3. USING THE METHODOLOGY

wish to use or they can allow the second transformation to automatically select one. Once the algorithm is selected manually or automatically, the second transformation will generate planning or optimisation control units for the application.

The second transformation algorithm supports the following commands:

- "python transform2.py -v" - validate the planning model and the library of algorithms.
- "python transform2.py -s" - show the problem type and which algorithm will be selected.
- "python transform2.py -g" - generate the control units for the application.

The AI modeller can add new algorithms to the library as follows:

1. Add a new entry to the algorithm taxonomy XML file. The entry should specify the problem type and environmental conditions for which the algorithm is suitable. The taxonomy format is presented in Section 5.3.
2. Provide a mapping from the planning model data to the new algorithm. The API provided by planning model components is presented in Section 5.1.

The AI modeller may wish to evaluate the performance of a selected algorithm. Many planning and optimisation algorithms have parameters that can be tuned or customised for each application domain. Parameter values can be specified at layer 4 of the domain model. In order to test the performance of control units an evaluation platform is provided. The services provided by the evaluation platform are described in Chapter 6.

4.3.1 Tool Support

The tool support provided is shown in Fig. 4.2. The following tools are provided:

- A suite of domain model and policy XML and XSD schemas.
- A validation engine to check the validity of domain models. The LXML parser is used to validate and parse XML documents ¹.
- A transformation engine to parse the domain model and construct a planning model for use with planning and optimisation algorithms.
- A transformation engine to choose a suitable algorithm and generate application code. Both transformations are written in Python. Python is a general purpose interpreted programming language. Its simple syntax, dynamic typing and lack of compilation steps make it an ideal choice for rapid prototyping (Lutz 2007, 13).
- A library of planning and optimisation algorithms selected from research literature and implemented in Python.

¹<http://codespeak.net/lxml>

4.4. DOMAIN MODEL ARCHITECTURE

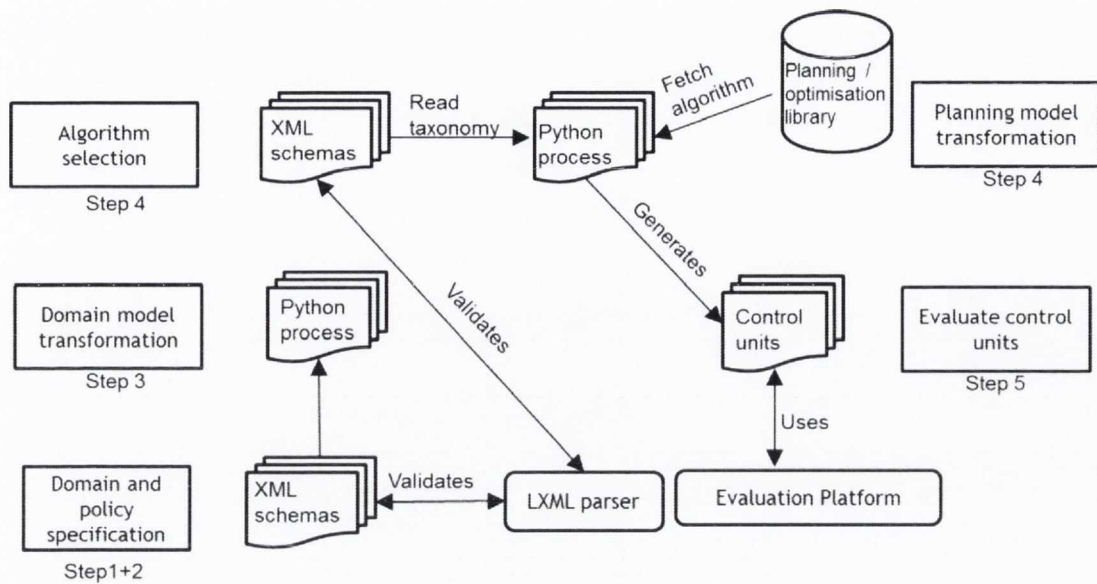


Figure 4.2: Development process tool support.

- An evaluation platform to test the performance of generated application code. This platform provides simulated sensor data and run-time middleware services for sensor and actuator discovery and access.

4.4 Domain Model Architecture

The architecture of the domain model is shown in Figure 4.3 and consists of four layers. The design of each layer is detailed in Section 4.6. This section describes the overall architecture of the domain model.

The domain model layers and schemas act as a template to the developer. By studying the design of each layer, a developer can determine the information that must be provided to use the methodology.

The domain model abstractions provide a high-level method of specifying an application state space. Determining state information in pervasive computing applications often requires accessing one or more sensor and actuator data sources distributed throughout the deployment region of an application. The domain model contains sensor, actuator and state abstractions to facilitate the specification of application state space in pervasive computing environments.

Layer 1 holds information that is fixed and known at design time. This layer is used to specify infrastructural elements that exist in the deployment environment. Infrastructural elements characterise physical artefacts relevant to the application being developed. Physical artefacts relevant to an application in the transportation domain might include

4.4. DOMAIN MODEL ARCHITECTURE

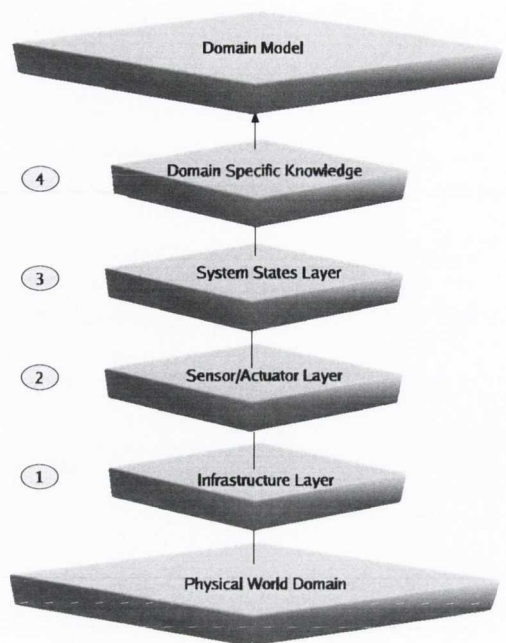


Figure 4.3: Layered domain model.

the location and geometry of traffic junctions and road links, whereas office equipment and layout might be relevant artefacts in a smart-office application. All layer 1 elements have a spatial attribute. For example, the spatial attribute of a road link element may be specified as a polygon object using a standard coordinate reference system.

Layer 2 provides information about sensor and actuator elements in the deployment environment. This information is used to determine the functionality, quality and spread of sensor and actuator infrastructure in the deployment environment. Layer 2 provides sensor and actuator meta-data used to query sensor and actuator infrastructure and used in state inference routines.

Layer 2 elements also have a spatial attribute. The spatial attribute of a sensor includes both its physical location and sensing range. The values provided by sensor and actuator elements modelled at layer 2 are typically not available until runtime.

Layer 3 provides System-State elements that are used to specify templates for application state spaces. System-state elements are typically composed over layer 1 and 2 elements. Examples of system-states include vehicle throughput at a traffic junction, journey time along a road link and power consumption in a room. The layer 1 and 2 element types referenced in a system-state element are mapped at run-time onto physical entities in the deployment environment. When specifying a system-state, the domain modeller is required to provide a reference to an inference algorithm specifying the logic used to calculate a value for the system-state. The value is a function of data that will be provided by the entities mapped to the constituent layer 1 and 2 elements over which the system-

state has been defined. (Note that the compound noun, *system-state*, when used in the remainder of this thesis always refers to a layer 3 element or a specification of a layer 3 element.)

Layer 4 of the domain model represents domain-specific knowledge as a set of attribute:value pairs. Examples of typical layer 4 data include specifications of algorithms to use in control units, prior probabilities for system-state values or parameter values for specific algorithms.

4.5 Domain Model Topology Abstraction

A core feature of the domain model design is the use of spatial information to model pervasive computing environments. The domain model uses a topographical approach to modelling space. The relevant spatial attributes of sensors, actuators, policies, and states are modelled as geometric shapes. Individual shapes are defined by sequences of coordinates based on a chosen, well-known coordinate system.

The relations between objects with spatial attributes are defined by the position of their respective shapes within the chosen common coordinate system. This is particularly useful in large-scale pervasive computing environments with large state spaces distributed over large geographical areas. Applications using spatial attributes can exploit these implicit relations to link diverse information together for an application-specific purpose, without the need to specify explicit interaction between objects (Bauer et al. 2002). They may access spatially-related information, for example, by means of exploiting the distance between shapes or by exploiting containment and intersection relations. This might, for example, enable a vehicle-based information system to retrieve the locations of car parking facilities within a certain distance from its current location.

Spatial attributes are represented by geometric shapes such as points, lines, polygons and combinations of polygons. The shapes may be chosen to reflect the physical space occupied by objects or may describe the sensing zone of sensors or the physical region in which a policy is to be deployed. For example, a city-wide car parking system might use spatial attributes to define the physical locations occupied by its car parks whereas a road weather service might use spatial attributes to outline the locations occupied by weather stations as well as the areas to which reports from individual stations apply.

The geometric shapes are specified using a common coordinate system. Coordinates derived from third party location sensors, such as Global Positioning System (GPS) receivers, can be mapped onto the reference system used by the domain modeller if they are based on another system.

Policies are used by the domain modeller to specify reward-models for possible values of layer 3 system-state elements. The scope of a policy indicates the deployment region of an application. System-state elements are templates used by the transformation engine to generate components called state-variable objects, that provide application state space values. System-state elements also have scope attributes. The scope attribute of a system-

4.6. DOMAIN MODEL LAYERS

state defines a logical region of the deployment environment over which a system-state is defined. (Note that the compound noun, state-variable, when used in the remainder of this thesis always refers to the planning-model component generated to provide run-time values for system-state elements.)

For example, a domain model for a UTC application might define a system-state to detect the number of public transport vehicles at traffic junctions. If layer 1 of the UTC domain model contained traffic junction infrastructure elements, the transformation engine would generate a state-variable object to count the number of public transport vehicles at each location in the deployment region in which a junction infrastructure element is specified.

As an application executes the certainty of state space information is related to the sensor infrastructure in the deployment environment. The application state space is represented by a set of state-variable objects, each of which is tagged with a spatial attribute indicating the region of the environment it relates to. At run-time state-variable objects issue spatial queries to determine the relevant sensor and actuator infrastructure in the region of the deployment environment corresponding to the geometry of their spatial attribute.

The spatial queries used by state-variable objects to determine relevant sensor and actuator infrastructure are created at design time by the transformation engine and execute at run-time. The topological abstraction underlies the programming model used in the methodology and mandates that domain model elements from layers 1-3 have spatial attributes.

The design of the topology abstraction is shown in the Location Reference class in Fig 4.4. It contains a `referenceSystemID` and a `location` attribute. The first attribute identifies the coordinate reference system used. By specifying a reference system identifier it allows elements to be specified in a range of geographic data formats and to be used in the same domain model. The `location` attribute specifies the geometric shape associated with the entity. The location and geometry properties are specified using the Simple Features Specification for SQL (SFS)² standard GIS standard data format.

4.6 Domain Model Layers

Domain model elements are designed to map to physical artefacts that exist in pervasive computing environments such as sensors and actuators and logical artefacts such as application policy and state space.

The domain model is structured using an object-oriented approach. The `Model` element is the base class from which all layer elements are derived. It contains two attributes, the `type` and `dataSourceId`, that identify respectively the domain-model layer to which the element belongs and the data format used to model the element.

²<http://www.opengeospatial.org/docs/99-049.pdf>

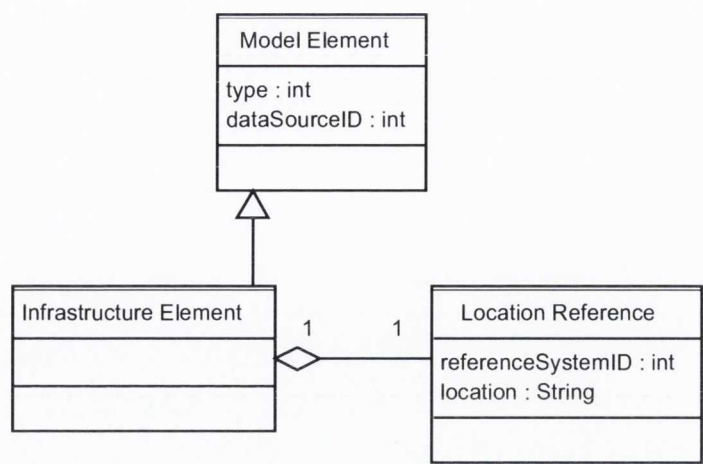


Figure 4.4: Infrastructure layer.

4.6.1 Infrastructure Layer

Layer 1 specifies the physical artefacts that exist in the deployment environment. The domain modeller specifies application relevant physical artefacts and their attributes at this layer. The domain model provides an *Infrastructure element* to model physical artefacts.

The design of this layer is shown in Figure 4.4. *Infrastructure element* is a subclass of a *Model element*. The `type` attribute for model elements is set to 1 to indicate that it is a Layer 1 element and the `dataSourceID` can take the values 1 or 2 to indicate that the data format used to specify the element is SQL or XML respectively. The transformation engine currently support layer 1 data sets specified in XML or SQL.

An instance of an infrastructure element contains a location reference instance that specifies the coordinate reference system used and the location of the physical artefact. Attributes of physical artefacts may be relevant to determining the value of a system-state, for example, the speed limit of a road link may be used in a congestion calculation. In this case the speed limit attribute is treated as a piece of data with a confidence of 1. This confidence value reflects the fact that it is fully observable and known.

Infrastructure elements can be sub-classed by the domain modeller to include application relevant information.

4.6.2 Sensor/Actuator Layer

Layer 2 models the sensor and actuator infrastructure in the deployment environment. Layer 2 holds meta-data describing the data produced by the sensor and actuator infrastructure. The meta-data is used in sensor fusion techniques and is also used to build queries to middleware services for accessing sensors and actuators at run-time. The domain model does not hold sensor data readings or information on actions being executed

4.6. DOMAIN MODEL LAYERS

by actuator infrastructure. It is assumed that sensor data will be accessed at run-time through a pervasive computing middleware service.

Sensors and actuators can be mobile or stationary. If the infrastructure is stationary then the spatial attribute and meta-data may be known at design time and can be entered in layer 2 of the domain model. This information will be written to the planning model generated by the first transformation algorithm and subsequently embedded in the generated control units.

The design of this layer is shown in Figure 4.5. The Sensor and Actuator classes inherit from the Model Element class and both have an associated spatial attribute. A Sensor object contains one or more Data objects used to specify what values the a sensor provides, and an Actuator object contains zero or more Data objects and one or more Action objects used to specify what actions an actuator supports. Data and Action objects are used at run-time by inference algorithms when determining state values.

Sensor and actuator objects contain a mobility attribute that, if set to true, indicates that meta-data required for sensor fusion may need to be provided at run-time. The use of self-describing sensors is one technique for obtaining meta-data at run-time (IEEE 2007). Run-time acquisition of sensor and actuator meta-data is not addressed in this thesis.

4.6.2.1 Sensor Elements

The Sensor class models sensors in the deployment environment and contains the following attributes:

- The name of the sensor.
- The location is the spatial attribute of a sensor.
- The cost associated with using the sensor. This could reflect power or communication charges involved in accessing a sensor.
- A flag indicating the mobility of the sensor. If sensors are mobile then it may not be possible at design time to determine which sensors will be in the deployment region at run-time. In this case the mobility flag is used to indicate that middleware services are required to retrieve the sensor meta-data and spatial attribute of relevant sensor elements discovered at run-time.
- The types of data produced by the sensor. For example a weather sensor could produce numerous data types such as: wind-speed, rain-level and air-temperature.

Sensor Data

Sensor data readings are accessed at run-time using middleware queries that are constructed using the sensor meta-data contained in the domain model. Once the sensor data is retrieved by a middleware it is wrapped in Data objects.

Data elements contain attributes that specify the degree of confidence associated with individual sensor readings and actuator actions and the cost of using them. For example,

4.6. DOMAIN MODEL LAYERS

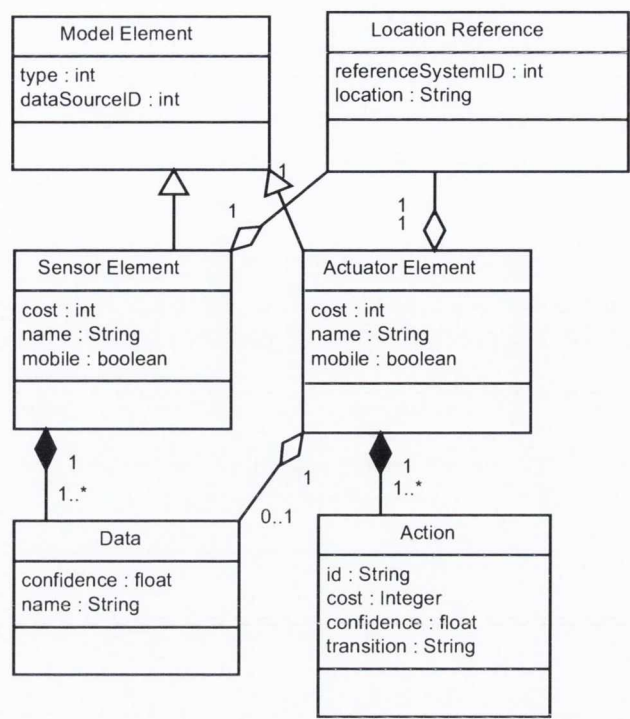


Figure 4.5: Sensor and actuator layer.

a domain modeller could specify an inductive loop sensor element that produces a traffic-demand data reading with a confidence attribute of 0.9 and that a virtual-loop sensor using video imagery to produce the same data reading may have a reduced confidence of 0.75. These values reflect the domain modellers relative confidence in the sensor infrastructure in the deployment region.

In general determining the confidence value for a particular sensor will require either the use of self-describing sensors (IEEE 2007), or else may be obtained from sensor specifications and manufacturer documentation. SensorML can be used to provide a standard digital means of providing specification sheets for sensor components and systems (Robin & Botts 2006).

Data elements are part of a sensor’s specification and have the following attributes:

- A name that identifies the sensor readings. This name is used in sensor discovery queries issued by the inference routines to discover sources that can provide the required layer 2 inputs named in the system-state specification.
- A confidence attribute that indicates the degree of confidence associated with individual sensor readings. For discrete sensor data the confidence measure is a probability value between 0 and 1 indicating the likelihood of sensor data being correct. For continuous sensor data the confidence value is the variance associated with sensor readings. The use of the confidence attribute varies depending on the algorithm

being run by the control unit.

The sensor model presented here contains information identified in Chapter 2, as necessary to support support planning, optimisation and inference activities.

4.6.2.2 Actuator Elements

Planning algorithms require that state-transition systems be specified so that actions can be chosen in accordance with system policy. The state-transition system in a pervasive computing application is provided through the functionality of the actuator infrastructure in the deployment environment.

Actuators implement actions that may effect a change in the state of a system. The spatial attribute of an actuator includes its location and the region of the environment over which its actions have an effect. Actuator elements contain actions. Each action has a confidence attribute that can be used to seed a stochastic state transition system. The greater the confidence the higher the probability assigned to a state transition caused by the actuator.

If a traffic-light controller offers phase switching functionality there may be a 0.95 confidence in the outcome of a phase switch transition. In this case there is a 0.95 chance of a successful phase switch and a 0.05 chance of a traffic-light failure.

The effects of actions are specified using state charts. The domain model implementation uses a modified versions of the State Chart XML (SCXML) language to specify state transition information. SCXML is currently a Working Draft published by the World Wide Web Consortium (W3C) (Barnett et al. 2009). SCXML provides a generic state-machine based execution environment based on Harel State Tables and supports composite state spaces and probabilistic transitions which makes it suitable for specifying state charts for pervasive computing environments.

The SCXML standard defines an actuator as being in a number of states that correspond to the action it is currently executing. Layer 2 of the domain model adopts the same approach as the SCXML standard in describing actions as states in which an actuator can be. The state element in layer 2 characterises the state of an actuator and may be distinct from a system-state element in layer 3 which characterises the state of an application.

Actions are specified as a combination of initial state, action and final state. Stochastic transitions are specified to reflect the possibility of actuator failure with failure rates derived from the actuator confidence attribute.

The domain model implementation provides an XML schema to specify actions. The schema is shown in Listing 4.1. The actuator specification contains one or more state elements modelled on the SCXML state-element specification and extended to include a location attribute.

The state elements in an actuator specification contain the following information about implemented actions:

- The id or name of the action.

```

1 <?xml version="1.0"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <xsd:element name="actiondata">
4     <xsd:complexType>
5       <xsd:sequence>
6         <xsd:element name="details" type="xsd:string"/>
7         <xsd:element name="actuator" maxOccurs="unbounded">
8           <xsd:complexType>
9             <xsd:sequence>
10              <xsd:element name="name" type="xsd:string"/>
11              <xsd:element name="mobile" type="xsd:boolean"/>
12              <xsd:element name="state" maxOccurs="unbounded">
13                <xsd:complexType>
14                  <xsd:sequence>
15                    <xsd:element name="id" type="xsd:string"/>
16                    <xsd:element name="referenceSystemID" type="xsd:string"/>
17                    <xsd:element name="location" type="xsd:string"/>
18                    <xsd:element name="datamodel" minOccurs="0" maxOccurs="unbounded">
19                      <xsd:complexType>
20                        <xsd:sequence>
21                          <xsd:element name="cost" type="xsd:integer"/>
22                          <xsd:element name="confidence" type="xsd:decimal"/>
23                        </xsd:sequence>
24                      </xsd:complexType>
25                    </xsd:element>
26                    <xsd:element name="transition" minOccurs="0" maxOccurs="unbounded">
27                      <xsd:complexType>
28                        <xsd:attribute name="event" type="xsd:string" />
29                        <xsd:attribute name="cond" type="xsd:string" />
30                        <xsd:attribute name="target" type="xsd:string" />
31                      </xsd:complexType>
32                    </xsd:element>
33                  </xsd:sequence>
34                </xsd:complexType>
35              </xsd:element>
36            </xsd:sequence>
37          </xsd:complexType>
38        </xsd:element>
39      </xsd:sequence>
40    </xsd:complexType>
41  </xsd:element>
42 </xsd:schema>

```

Listing 4.1: Action XSD schema.

4.6. DOMAIN MODEL LAYERS

- The location of an action indicates the spatial region of the environment in which the action has an effect.
- A datamodel element providing the cost associated with invoking this action and the confidence associated with the transition events invoked by the action.
- Transition elements enumerating the possible states that the actuator can reach from its current state. Each transition element has the following attributes:
 - event, the name of the action taken by the actuator to initiate the transition.
 - condition, the confidence associated with the transition reaching its target.
 - target, the resultant state reached as a result of the action.

A complete example of the use of the actuator specification is given in the Junction Controller case study in Section 4.8.

4.6.3 System-States Layer

Layer 3 of the domain model contains system-state elements. System-state elements are templates for defining the state space for the application being developed. System-state elements are specified by the domain modeller and are composites of layer 1 and layer 2 elements. The system-state specification identifies the layer 1 infrastructure and layer 2 sensor and actuator elements that are combined to determine the value of the system-state. Each system-state specification has a scope that indicates the region of the deployment environment to which they relate.

The transformation algorithms use these templates to create instances of system-states that execute in the deployment environment at run-time to provide state-space values to planning and optimisation algorithms. Instances of system-states are referred to as state-variable objects. State-variable objects are responsible for querying middleware services to access sensor and actuator data required to calculate state-space values.

An example of a system-state is vehicle throughput at a traffic junction. If a domain model contained a definition for this system-state, the first transformation algorithm would create a state-variable object to provide a value for vehicle throughput at every traffic junction in layer 1 of the domain model. The spatial attributes of traffic junction elements from the Infrastructure layer would become the spatial attribute of state-variable objects.

State-variable objects provide an interface for the AI modeller against which to program planning and optimisation algorithms. A system-state specification includes an inference algorithm whose logic specifies how state values are calculated from the run-time value of sensor data. Sensor and actuator meta-data are combined using the inference function, to derive an observability measure for the system-state value. The observability measure provides a likelihood for the true value of the system-state given the available sensor data, thereby quantifying the uncertainty associated with state values.

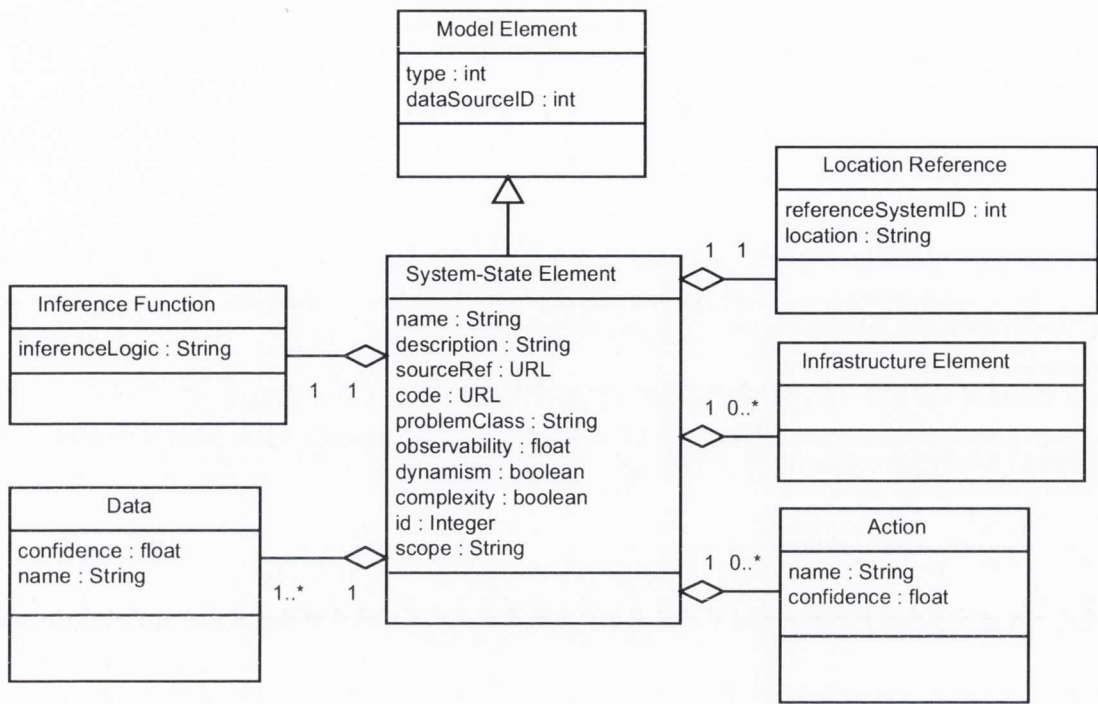


Figure 4.6: System-states layer.

The design of this layer is presented in Figure 4.6 and implemented using an XML schema. Domain modellers specify system-states in XML and the XML schema is used by the domain-model transformation to validate specifications. An excerpt from the the XSD schema is shown in Listing 4.2. The full schema is given in Appendix F.

A system-state specification contains the following information:

- A numeric id and textual name of the system-state.
- The description of the system-state. System-states can be re-used across applications. The description should provide a high-level explanation of the purpose and use of the state.
- The properties of the system-state include the attributes of complexity, dynamism and observability. These properties act as an input to selecting an algorithm to be deployed in the control units. Complexity and dynamism are Boolean values which indicate the requirement for heuristic and online algorithms respectively. The observability attribute can take the values "full" or "partial".

The observability attribute of a system-state indicates whether sensor data is required to calculate the value of the system-state. If set to "partial", low-level sensor fusion techniques are engaged at run-time in associated state-variable objects, to fuse sensor data and to return a likelihood value for the state-variable that is reflective of the sen-

sor spread and quality within the region of the deployment environment referenced by its spatial attribute.

If the observability attribute is set to "full", no sensor fusion techniques are required as all inputs into the system-state are fully observable and hence their values are known.

- The *scope* attribute defines the physical region of the deployment region to which the system-state relates. It is used to generate a spatial attribute for each state-variable object.

Scope is a complex attribute. It contains a mandatory *type* attribute, an optional *number* attribute and a mandatory *value* attribute. The *type* attribute can take three values:

- "element"

This value indicates that the system-state is assigned a scope that maps to an element in the Infrastructure layer. In this case the domain-model transformation algorithm will create a number of state-variable objects corresponding to the number of instances of the associated element in layer 1. Each of these state-variable objects will be assigned a spatial attribute corresponding to the geometry of the layer 1 element.

- "multi-element"

This value indicates that the system-state is assigned a scope that maps to multiple infrastructure or other system-state elements. In this case the domain-model transformation algorithm will create state-variable objects with spatial attributes corresponding to the minimum-bounding-rectangle of the multiple elements. If "multi-element" is specified, then the optional *number* attribute is used to indicate how many infrastructure elements are associated with each state-variable object.

A system-state whose scope is "multi-element", may be defined over a number of other system-states. This ability to have nested system-states allows the flexibility to build complex state-spaces for pervasive computing applications.

- "region"

This value indicates that the system-state is assigned a scope that maps to a region of the deployment environment. In this case the spatial attribute of a state-variable object is assigned a newly created geometry that matches the region specified. A state-variable object is created for each region specified by the domain modeller.

- The *value* attribute specifies the name of the Infrastructure or System-state element to which the *type* attribute relates.

4.6. DOMAIN MODEL LAYERS

- The `inputs` attribute lists the layer 1, layer 2 and layer 3 inputs that are combined in the inference algorithm used to calculate the value of the system-state. An example of a layer 1 input would be "speed-limit" or "length". An example of a layer 2 input would be "temperature" or "queue-length". Layer 3 inputs are the names of other system-states defined in the domain model.

Each input is combined with the state-variable's spatial attribute to build a query to a middleware service to discover and access entities in the deployment environment that can provide information corresponding to the named inputs.

The inputs are resolved using the topology abstraction rather than by explicitly identifying layer elements. Consider the case of resolving layer 2 inputs. There may be multiple sensors that can provide data corresponding to the specified input and even assuming sensor locations are fixed and known at design time, it would be laborious for the domain modeller to have to manually map sensor locations and placements to individual state-variable objects.

- The `implementation` attribute contains `code` and `sourceRef` attributes. The `code` attribute specifies the programming language in which the inference algorithm is implemented. The `sourceRef` attribute provides a reference to an implementation of the inference function used by state-variable objects to combine the inputs and calculate the value of the system-state.

The inference function is provided by the domain expert. The first transformation algorithm resolves the `sourceRef` name to a module and imports the module checking that it provides an inference function. A copy of the inference function is loaded into each state-variable object created to provide state-space values at run-time.

The domain modeller can assume that the values of the specified inputs will be provided at run-time and writes the inference function code used to combine them. An API is provided for writing this inference function and is discussed in the next section.

The system-state schema presented here is similar in concept to Sensor ML process chains (Robin & Botts 2006)³ in that they contain an inference algorithm for calculating the state value from a combination of input parameters.

4.6.3.1 Inference Function

An inference function is provided as part of the specification of a system-state. The inference function can be considered as feature or decision level sensor fusion that can involve fusing together multiple inputs from layers 1-3 to give a final value for the system-state. This task is often highly domain or task-specific and the inference logic is provided by the domain modeller at design time.

³Sensor ML has been approved as a formal standard from the Open Geospatial Consortium. See <http://www.opengeospatial.org>

4.6. DOMAIN MODEL LAYERS

```
1 <?xml version="1.0"?>
2   <xsd:element name="systemState">
3     <xsd:complexType>
4       <xsd:sequence>
5         <xsd:element name="id" type="xsd:integer" />
6         <xsd:element name="name" type="xsd:string" />
7         <xsd:element name="description" type="xsd:string" />
8         <xsd:element name="properties">
9           <xsd:complexType>
10            <xsd:sequence>
11              <xsd:element name="complexity" type="xsd:boolean" />
12              <xsd:element name="dynamism" type="xsd:boolean" />
13              <xsd:element name="observability" type="xsd:string" />
14              .....
15            <xsd:element name="scope">
16              <xsd:complexType>
17                <xsd:sequence>
18                  <xsd:element name="type">
19                    <xsd:simpleType>
20                      <xsd:restriction base="xsd:string">
21                        <xsd:enumeration value="element"/>
22                        <xsd:enumeration value="multi-element"/>
23                        <xsd:enumeration value="region"/>
24                      .....
25                      <xsd:element name="number" type="xsd:integer" minOccurs="0" maxOccurs="
26                        unbounded" />
27                      <xsd:element name="value" type="xsd:string" />
28                    .....
29                  <xsd:element name="location" type="xsd:string" minOccurs="0" maxOccurs="
30                    unbounded" />
31                <!-- Note the inputs etc are defined in scope -->
32              <xsd:element name="inputs">
33                <xsd:complexType>
34                  <xsd:sequence>
35                    <xsd:element name="layer1" minOccurs="0" maxOccurs="unbounded">
36                      <xsd:complexType>
37                        <xsd:sequence>
38                          <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
39                            "unbounded" />
40                        .....
41                      <xsd:element name="layer2" minOccurs="0" maxOccurs="unbounded">
42                        <xsd:complexType>
43                          <xsd:sequence>
44                            <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
45                              "unbounded" />
46                          .....
47                        <xsd:element name="layer3" minOccurs="0" maxOccurs="unbounded">
48                          <xsd:complexType>
49                            <xsd:sequence>
50                              <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
51                                "unbounded" />
52                              .....
53                            <xsd:element name="implementation">
54                              <xsd:complexType>
55                                <xsd:sequence>
56                                  <xsd:element name="code" type="xsd:string" />
57                                  <xsd:element name="sourceRef" type="xsd:string"/>
58                                .....
59                              .....
60                            .....
61                          .....
62                        .....
63                      .....
64                    .....
65                  .....
66                .....
67              .....
68            .....
69          .....
70        .....
71      .....
72    .....
73  .....
74  </xsd:element>
75</xsd:element>
```

Listing 4.2: XML schema for system-states specification.

4.6. DOMAIN MODEL LAYERS

```
1 def calculate(inputs):
2     """
3     =====Inference_function_template
4     """
5
6     # howto access layer 1 data
7     for key, value in inputs["layer1"]:
8         .....
9
10    # howto access layer 2 data
11    layer2_data = inputs["layer2"]:
12    for datum in layer2_data:
13        data_type = datum.name
14        if datum.type == "discrete":
15            (readings, likelihoods) = datum.fused_likelihood_discrete()
16        elif datum.type == "continuous":
17            (mean, variance) = datum.fused_likelihood_continuous()
18
19    # howto access layer 3 data
20    for datum in inputs["layer3"]:
21        for (value, likelihood) in datum:
22            .....
```

Listing 4.3: Inference function template.

The inference function is executed at run-time under the assumption that the run-time values of each of the layer 1-3 inputs named in the system-state specification will be available. The run-time provision of these values is provided by state-variable objects added to the planning model and is hidden from the domain modeller. Details of this process are described in Section 5.1.

The domain modeller unpacks the input values from an input object passed into the inference function.

The structure of an input object is shown in Fig. 4.7 and a template Inference function shown in Listing 4.3 demonstrates how the data within the inputs object is accessed.

The inputs object is structured as a nested hash-table or dictionary. The first tier of the inputs object is indexed by the layer name which can take the values: "layer1", "layer2", "layer3". The second tier of layer 1 data is a dictionary of inputs names and lists of values corresponding to the names. Line 7 of Listing 4.3 shows how key, value pairs of layer 1 data are accessed. Each key is a layer 1 input specified in a system-state specification.

The second tier of layer 2 data is a list of Data objects. Lines 12-17 of Listing 4.3 shows how layer 2 data is accessed. Sensor data can be discrete or continuous. Line 13 shows how the type of the sensor data is accessed. The type corresponds to the layer 2 inputs named in a system-state specification. Line 15 shows (readings, likelihoods) values accessed from a discrete Data object. These values are sensor reading and likelihood pairs. An example of sensor readings and likelihoods measuring traffic demand would be: "high:0.65", "medium:0.3", "low:0.05". Line 17 shows (mean, variance) values accessed from a continuous Data object. An example of a continuous data reading and a variance would be a temperature reading of (17,1) indicating a mean value of 17 degrees and a

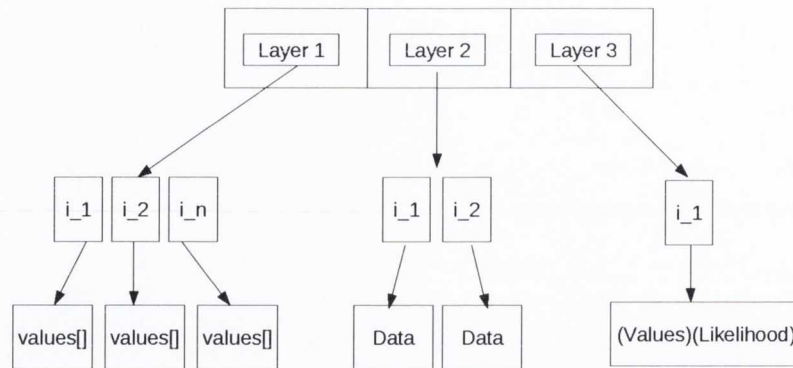


Figure 4.7: Inference function inputs.

variance of 1 degree.

The second tier of layer 3 data is a list of (value, likelihood) pairs. These pairs are returned by the inference functions in system-states specified to be layer 3 inputs.

The data from layers 1-3 is combined in the inference function to produce a value and a likelihood estimate for the system-state. The domain expert is expected to provide the logic involved in combining the inputs to determine the state value. This allows for functions of arbitrary complexity to be specified. The case studies provide simple and complex examples of inference functions. The operation of Data objects is described in more detail in Section 5.1.

4.6.3.2 Factored State Space

A state space is factored if it is defined over a number of random variables that fully describe it (Paquet 2006, 50). Using this state-variable representation each state is represented by a tuple of values of n state variables $\{x_1, \dots, x_n\}$. System-states are used in the domain model to define a pervasive computing application state space. The state space defined is factored in that system-states are defined over inputs from layers 1-3.

The use of a factored state space has a number of benefits.

1. Compact Representation

It allows for a compact representation of a state-transition function as a dynamic Bayesian network and exploits assumed independence relationships between features (Guestrin et al. 2003).

2. Efficient Belief Updating

Planning algorithms in pervasive computing environments do not have direct access to the state space but are required to reason about the state of the environment using a set of beliefs that are derived from sensor data. These beliefs are represented using probability distributions over the state space. The set of all probability distributions over the belief space forms an $(n - 1)$ dimensional continuous space. Exhaustive

searches of this space are intractable with problems consisting of even a moderate number of states (Cassandra et al. 1994).

If a factored state space representation is used then a belief state is a full joint probability distribution over all random variables. The complexity of updating the set of beliefs defined over such a full joint distribution grows exponentially as the number of states increases. However if independent subsets of variable are identified then it is possible to compute belief states as the product of the subset of the variables probabilities. The factored state definitions facilitate the use of approximate solutions by allowing the quick pruning of impossible or unlikely belief states that contain values of constituent variables with low or zero probability thus reducing the size of the search space and the complexity of belief updates (Paquet 2006).

The target class of applications addressed in this work are large-scale with large state spaces. The Junction Controller case study presented in Section 4.8 uses a Bayesian decision network to compactly represent a state transition function and reward model. The CCTV Selection case study uses independence relationships between input variables to reduce the complexity of belief updating.

4.6.4 Domain Specific Knowledge Layer

Layer 4 of the domain model is used to specify domain-specific knowledge. This layer can be used by both the domain and AI modellers. Layer 4 is provided to allow the domain expert or AI modeller to customise the performance of the planning and optimisation algorithms that will be embedded in the generated control units. An algorithm can only be parametrised by a domain or AI modeller if the algorithm has been manually specified for use at layer 4 of the domain model. (Ghallab et al. 2004, 196) state that nearly every successful planning system for real world applications incorporates domain-specific heuristics. Planning competition results indicate the value of incorporating domain-specific knowledge (Fox & Long 2002) (Bacchus 2001).

The AI modeller can customise the algorithms in two ways:

- Algorithm Selection

The AI modeller can use layer 4 to specify which planning and optimisation algorithms should be selected and used in an application. If no algorithm is specified, the planning model transformation will select an appropriate one using the algorithm taxonomy.

- Algorithm Parametrisation

The AI modeller can specify values for the parameters associated with each algorithm. Many algorithms have parameters that control their operation. For example, some optimisation algorithms have temperature parameters which control their rate of exploration and some planning algorithms have horizon parameters that control the

4.6. DOMAIN MODEL LAYERS

number of actions the planning algorithm will plan for. If no parameters are specified for the chosen algorithm then default values are used in the generated control units.

Layer 4 can also be used by the domain modeller to specify prior probabilities for system-state values.

Layer 4 allows the AI modeller to specify which algorithms to use when control units are generated. This allows the domain model data to be rapidly evaluated with a variety of planning and optimisation algorithms to improve application performance. This is very useful because the choice of optimum algorithm will often be determined by empirical observation of algorithm performance. Likewise, the optimum values of parameters can be determined empirically. The evaluation of the CCTV Selection case study in Chapter 6, measures the impact of algorithm selection and parameter customisation on application performance.

Layer 4 data is specified using `name:value` notation. Layer 4 data is structured using headers to indicate whether the data pertains to system-states or to algorithm selection and parametrisation.

Two types of section headers are currently used.

- **System-State.**

A section header can be a system-state name. Entries following until the next section header are associated with the system-state. For example if a `TrafficDemand` system-state is defined at layer 3 and takes three possible values "heavy-medium-low", then a layer 4 entry specifying prior probabilities could read:

```
[TrafficDemand]
prior: 0.4,0.4,0.2
```

This entry associates the prior probabilities with the corresponding discrete value of the system-state. The use of prior probabilities is dependent on both the inference function specified by the domain modeller and the planning and optimisation algorithms selected and embedded in control units. As an example, in the Junction Controller case study, prior probabilities are used to seed the Bayesian Decision network used by an MEU algorithm. However a round-robin action selection algorithm in the same case study does not use prior probability information. This is because a round-robin action selection algorithm does not infer the system-state values from sensor data.

The domain modeller must ensure that the prior probabilities specified are compatible with the range of possible system-state values specified in the policy file. For example, if a discrete state can take 4 values, then a prior probability must be specified for each possible value.

- **Algorithm.**

Entries following an `[Algorithm]` section header are associated with the planning or

optimisation algorithm that is selected for the application. For example, the following layer 4 data could be used by an AI modeller wishing to specify that a multi-objective simulated annealing algorithm be used and that one of its parameters called temperature is to take a value of 7:

```
[Algorithm]
algorithm: SMOSA
temperature: 7
```

It is the responsibility of the AI modeller to ensure that the algorithm named is available in the library and that the parameter names are valid. Default values for all algorithm parameters are provided when an algorithm is added to the library. This is discussed in more detail in Chapter 5. The structure of layer 4 is based on the Python ConfigParser module ⁴.

4.7 Policy Specification

The domain modeller specifies the policy for the application. Policy specification provides a high-level method for the domain modeller to control application behaviour. Application policy is specified by associating rewards with the range of possible system-state values. The transformation algorithms use the policy specification to create a reward model attached to states and/or actions. For planning and optimisation problems the control units attempt to optimise a compound function summing the values returned by the reward model.

An XML schema is provided to validate the policy specification. The schema is shown in Listing 4.4. The policy specification contains the following information:

- Problem Type.

The problem attribute indicates whether the problem is a planning or optimisation problem. This is a mandatory attribute and influences the selection of planning model components. For planning problems, state-variable objects are generated by the domain-model transformation algorithm and written to the planning model. For optimisation problems state-variable objects are not generated by the domain-model transformation. Instead a State-Generator component is generated from domain-model data and written to the planning model. State-generator objects are used at run-time by optimisation algorithms to generate the required run-time components used by the optimisation control-units. This component is described in detail in Chapter 5.

Planning problems are subdivided into single and sequential decision problems. An optional subtype element can be used to indicate whether or not the problem is

⁴<http://docs.python.org/library/configparser.html>

4.7. POLICY SPECIFICATION

a single or sequential decision problem. The subtype field is used as an aid to algorithm selection.

- **Deployment Region.**

The scope attribute identifies the region of the physical environment over which the application is to be deployed. When the planning model is being created, sets of state-variables or state-generator objects are created to provide values for each instance of a system-state in the deployment region.

- **System-States.**

The policy specification contains state attributes that identify the system-states of interest to the application. Each state attribute contains a mandatory `name` attribute. The `name` of each state listed in the policy file is used by the domain model transformation algorithm to determine which system-state elements in layer 3 of the domain model should be parsed. When parsed, the system-states are used to create the state space over which the planning and optimisation control units reason.

- **Reward Model.**

Each state attribute contains a mandatory `reward` sub-attribute. Data extracted from the reward attributes are combined by the domain model transformation algorithm to create a reward model. The reward model is written to the planning model and used by planning and optimisation algorithms to determine application behaviour. Rewards can be specified for values of system-states or for state/action combinations.

Each reward element contains a number of sub-attributes:

- `type` - is a mandatory attribute identifying whether the system-state produces data values that are discrete or continuous.
- `range` - this is a mandatory attribute specifying a hyphenated list of values that state-variable objects associated with the system-state can take. For discrete states the values identify the possible values the data can have, e.g, "high", "medium" or "low". For continuous states the values identify the range that the values can take, e.g, "0-1".
- `action` - is an optional attribute used to associate an action with the state and reward.
- `value` - is an optional attribute used to specify the numeric rewards associated with the range of possible values the state can take. For discrete data the numeric values are enumerated and matched against the hyphenated `range` attribute. For continuous data the value can be specified as "maximise" or "minimise".
- `weight` - is an optional attribute used to prioritise competing system-states. If two system-states have respective reward weights of 1 and 2, then the second is valued as twice as important as the first. The exact interpretation of this weighting is dependent on the algorithm used in the control unit.

4.7. POLICY SPECIFICATION

```
1 <?xml version="1.0"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <xsd:element name="policy">
4     <xsd:complexType>
5       <xsd:sequence>
6         <xsd:element name="scope" type="xsd:string"/>
7         <xsd:element name="problem" type="xsd:string"/>
8         <xsd:element name="subtype" type="xsd:string" minOccurs="0" maxOccurs="1"/>
9         <xsd:element name="state" minOccurs="0" maxOccurs="unbounded">
10          <xsd:complexType>
11            <xsd:sequence>
12              <xsd:element name="name" type="xsd:string"/>
13              <xsd:element name="reward">
14                <xsd:complexType>
15                  <xsd:sequence>
16                    <xsd:element name="type" type="xsd:string"/>
17                    <xsd:element name="range" type="xsd:string"/>
18                    <xsd:element name="action" type="xsd:string" minOccurs="0" maxOccurs="1"/>
19                    <xsd:element name="value" type="xsd:string" minOccurs="0" maxOccurs="1"/>
20                    <xsd:element name="weight" type="xsd:string" minOccurs="0" maxOccurs="1"/>
21                  </xsd:sequence>
22                </xsd:complexType>
23              </xsd:element>
24            </xsd:sequence>
25          </xsd:complexType>
26        </xsd:element>
27      </xsd:sequence>
28    </xsd:complexType>
29  </xsd:element>
30 </xsd:schema>
```

Listing 4.4: Policy XSD schema.

Sections 4.3 - 4.7 have described the programming model supported by the methodology. The next section describes how the programming model is used in two scenarios utilising planning and optimisation algorithms respectively.

4.8 Case Studies

This section demonstrates how the methodology is used to model two large-scale pervasive computing scenarios. In Chapter 5 we describe how the transformation algorithms parse the domain model to generate control units for the two scenarios.

4.8.1 Junction Controller Scenario

The Junction Controller scenario involves developing an application to manage the behaviour of traffic-light controllers in Dublin. The desired behaviour of the system is as follows. Each traffic light controller should access and use any available sensor data to measure the traffic demand. It should also access and use sensor data to detect the presence of emergency vehicles.

When the presence of an emergency vehicle is detected at a traffic junction, a traffic light phase should be chosen to accommodate that vehicle's transit through the junction. In the absence of an emergency vehicle being present the system should, at the end of each phase, switch to the phase that has the highest demand at that time. The emergency vehicle priority system is to be deployed over 247 traffic-controlled junctions throughout Dublin city centre.

4.8.1.1 Scenario Domain Model

This scenario can be modelled as a distributed pervasive computing application. The application is required to select actions for a set of traffic controller actuators. The application's state space includes the level of traffic demand and the presence or absence of emergency vehicles at traffic junctions throughout Dublin. Values for the state space are to be obtained from two distinct types of sensors deployed in the region.

In this section we show how a domain model is constructed to represent the application state space. The domain model contains information on the road network, the sensor and actuator infrastructure and the system-states relevant to the application.

Layer 1

Layer 1 of the domain model contains the information about the scenario that is known at design time. The road network infrastructure is static and is specified as a series of junctions containing traffic phases and connected by road links. A phase is a set of distinct legal turning manoeuvres that allows traffic to flow from one road link to another through a junction. A road link is associated with each lane of traffic and a stretch of road between two junctions can contain multiple links. Each junction contains a number of phases. Phases are specified over road links. Each phase can support multiple non-conflicting



Figure 4.8: Junction Controller scenario layer 1 infrastructure element data.

traffic flows from link to link. This is a standard model used to represent road networks and mirrors the design of the Paramics traffic simulator ⁵.

An example of a set of phases showing the legal turning manoeuvres at a junction is shown in Fig. 4.9(a). Phase 1 allows traffic to move on links connecting junctions 1245 to 816 and from 1245 to 1250. Phase 2 allows traffic to move on links connecting junctions 1250 to 816.

Each traffic junction and each phase are modelled as an infrastructural element at layer 1 of the domain model. There are 247 junction elements and 609 phase elements. Junction infrastructure elements have an alpha-numeric identifier and a spatial attribute. The spatial attribute of a junction is chosen to be a circle of radius 20 metres from the junction centre point. Phase elements have an alphabetic identifier and a series of text attributes identifying the road-links that they connect. The spatial attribute of a phase is chosen to be a multi-polygon geometry, which is the sum of the geometries of the links that it connects.

Layer 1 data is formatted using the SFS spatial data standard and stored in a PostgreSQL GIS database. The use of standard GIS data formats in our domain model allows us to use many GIS applications to view the domain model data. Accordingly a screenshot of the spatial data of layer 1 is shown in Fig. 4.8. This screenshot is generated from the uDig desktop GIS application ⁶.

Fig. 4.8 shows a section of the Dublin city-centre road network surrounding Trinity College. The scope of a traffic junction is represented a pink circle. The road links are specified

⁵<http://www.paramics-online.com/>

⁶<http://udig.refractory.net/>

4.8. CASE STUDIES

```
1 INSERT INTO sensor (id, dataSourceId, confidence, name, data, cost, mobile)
2 VALUES( '0', '2', '0.75', 'emergency_veh', 'emergency_veh', '2', 'False' );
3 UPDATE sensor SET geometry = GeomFromtext( 'POLYGON((312771_
    233752,312770.615705608_233748.09819356,312769.47759065
4 ....._233759.65366865,312770.615705607_233755.901806444,312771_
    233752.000000004,312771_233752)))', -1) WHERE id = '0';
```

Listing 4.5: Emergency-vehicle sensor meta-data excerpt.

```
1 INSERT INTO sensor (id, dataSourceId, confidence, name, data, cost, mobile)
2 VALUES( '247', '2', '0.85', 'traffic_demand', 'traffic_demand', '2', 'False' );
3 UPDATE sensor SET geometry = GeomFromtext( 'MULTIPOLYGON(((316085_
    233921,316087.764019774_233927.75649278,316109.764019774
4 ....._316085_233921)))', -1) WHERE id = '247';
```

Listing 4.6: Traffic-demand sensor meta-data excerpt.

as green polygons and the phases are represented as gray polygons. Only those junctions which are controlled by traffic-lights are shown with phases. The spatial attributes are defined using coordinates from the Irish national grid reference system.

Layer 2

Layer 2 of the scenario domain model contains two sensor elements and one actuator element. An Inductive Loop sensor is used to provide sensor data on traffic demand. These are stationary sensors and are located on road links at all junctions in the road network. Such sensors are actually deployed throughout Dublin city as part of the SCATS system (Fellendorf 1997). We assume that a stationary ambulance-presence sensor is present at each junction and provides sensor data on emergency vehicle identification and location.

Listings 4.5 and 4.6 show meta-data for one emergency-vehicle and one traffic-demand sensor modelled at layer 2.

From Listing 4.5 the emergency vehicle sensor has a confidence of 0.75. This is interpreted as the likelihood $P(\text{EmergencyVehiclePresent}|\text{SensorReading}) = 0.75$.

From Listing 4.6 the traffic demand Sensor has a higher confidence of 0.85 and is also interpreted as a likelihood. As mentioned in Section 4.6.2.1, these values should be obtained from the sensor manufacturer's specifications. A cost of 2 is associated with each sensor reading and the mobility flag is set to false. The data type produced by both sensors is also listed. The spatial attribute of each sensor is shown in Listings 4.5 and 4.6.

The spatial attributes of the sensors are specified as polygon shapes using an SFS function `GeomFromText()` (OpenGeoSpatialConsortium 1999). This function converts a string of coordinates into a spatial geometry. As with layer 1, the spatial attribute for all sensors is specified using the Irish National Grid coordinate reference system. Specifying the spatial attributes of infrastructure elements and sensors and actuators is a laborious task. For stationary elements this task need only be performed once. Moreover spatial data on fixed infrastructure is often available from local authorities and through initiatives such as

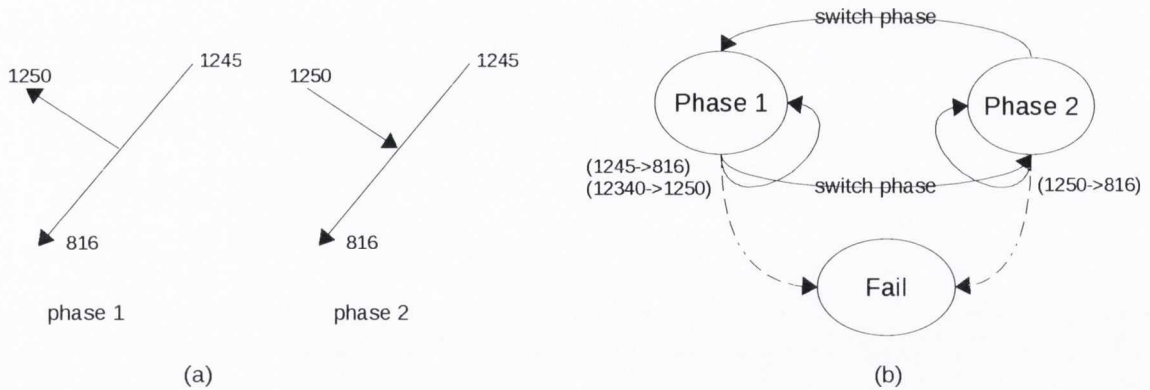


Figure 4.9: Junction 1244 phase elements.

OpenStreetMap ⁷ which distribute spatial data freely.

A traffic light controller actuator is located at each junction and is responsible for switching traffic phases. The actions specified for each junction controller consists of the available traffic-control phases at that junction. An example of the set of actions available at Junction 1244 is shown in Fig 4.9(b).

The actuator specification for Junction 1244 is shown in Listing 4.7. The actuator specification follows the design of the actuator element as discussed in Section 4.6.2.2.

From Listing 4.7 actuator 1244 has a name which is the junction number and a mobility flag which is set to false and provides two actions: phase 1 and phase 2. Following SCXML terminology these actions are described using state attributes. Each state attribute contains the following information:

- an id which is the action name.
- the geometry or region over which the action is executed.
- each datamodel element contains a cost and confidence element indicating the cost of invoking this action on the actuator and the likelihood of the action invocation succeeding.
- one or more transition elements indicating the state-action combinations to which the actuator can transition and the likelihood of the transition succeeding.

Layer 3

Layer 3 contains two system-states for this scenario. A TrafficDemand system-state can take the values: "heavy"; "medium"; and "light". An EmergencyVehiclePresent system-state that can take values: "true" or "false". These system-states are associated with phase infrastructure elements in layer 1 of the domain model and have a scope equal to the spatial attributes of phase elements. Both system-states are dynamic in that their values

⁷<http://www.openstreetmap.org/>

```

1 <actuator>
2   <name>1244</name>
3   <mobile>false</mobile>
4   <state>
5     <id>phase 1</id>
6     <geometry>MULTIPOLYGON(((316280 233765316287019124818
7       233762994535766316265019124818 233685994535766316258 233688316280 233765)
8       ))</geometry>
9     <datamodel>
10      <cost>1</cost>
11      <confidence>0.995</confidence>
12      <!-- link 1245-816 -->
13      <!-- link 1245-1250 -->
14    </datamodel>
15    <transition event = "switch-phase_2" cond = "rand_less_eq_0.995" target = "
16      phase_2"/>
17    <transition event = "switch-fail" cond = "rand_gt_0.995" target = "fail"/>
18  </state>
19  <state>
20    <id>phase 2</id>
21    <geometry>MULTIPOLYGON(((316141 233722316141837163087
22      233724880825917316258837163087 233690880825917316258 233688316141 233722)
23      ))</geometry>
24    <datamodel>
25      <cost>1</cost>
26      <confidence>0.995</confidence>
27      <!-- link 1250-816 -->
28    </datamodel>
29    <transition event = "switch-phase_1" cond = "rand_less_eq_0.995" target = "
30      phase_1"/>
31    <transition event = "switch-fail" cond = "rand_gt_0.995" target = "fail"/>
32  </state>
33 </actuator>

```

Listing 4.7: Actuator definition for junction controller 1244.

4.8. CASE STUDIES

```
1 <systemState>
2   <id>001</id>
3   <name>EmergencyVehiclePresent</name>
4   <description>This state determines if there is an emergency vehicle present
5     on a phase at a traffic junction.</description>
6   <properties>
7     <complexity>>false</complexity>
8     <dynamism>>true</dynamism>
9     <observability>partial</observability>
10  </properties>
11  <scope>
12    <type>element</type>
13    <value>phase</value>
14  </scope>
15  <inputs>
16    <layer1>
17      <name>junction</name>
18      <name>id</name>
19    </layer1>
20    <layer2>
21      <name>emergency_veh</name>
22    </layer2>
23  </inputs>
24  <outputs>
25    <type>boolean</type>
26  </outputs>
27  <!-- Implementations -->
28  <implementation>
29    <code>python</code>
30    <sourceRef>EmergencyVehiclePresent</sourceRef>
31  </implementation>
32</systemState>
```

Listing 4.8: Excerpt from EmergencyVehiclePresent system-state definition.

can fluctuate independently of the action taken by the traffic light. Likewise both system-states are partially observable as their values are inferred from sensor data. The complexity attribute is set to false for each system-state.

The specification of the EmergencyVehiclePresent system-state is shown in Listing. 4.8. Using the properties attributes, the domain modeller specifies this system-state as being dynamic and partially-observable. The scope of the system-state is defined as being of type "element" and of value "phase".

The domain model transformation algorithm will create the state space for this application using the layer 3 specifications. State-space components will be generated to measure the value of each system-state at physical regions of the deployment environment in which phase infrastructure elements are located.

In this scenario, there are 609 phase elements defined in the domain model and the domain model transformation will therefore create and store 609 state-variable objects for the EmergencyVehiclePresent system-state in the planning model.

The inputs attribute in Listing. 4.8 specifies data from layers 1 and 2 relevant to determining the value of the system-state. The layer 2 input referred to a sensor datum

4.8. CASE STUDIES

```
1 def calculate(inputs):  
2     for datum in inputs["layer2"]:  
3         if datum.name == "emergency_veh":  
4             return datum.fused_likelihood_discrete()
```

Listing 4.9: EmergencyVehiclePresent system-state inference function.

`emergency_veh` which is accessed at run-time through middleware invocations.

At run-time, state space components will use the `emergency_veh` datum name and spatial attribute of individual phase elements to issue queries to middleware services to discover and invoke sensors providing data on the presence of emergency vehicles.

Finally, the `implementation` attribute specifies that a module named `EmergencyVehiclePresent`, written in the python programming language contains the inference function used to combine the inputs to produce and output value for the system-state.

The inference logic for this system-state is simple and is shown in Listing 4.9. The domain modeller provides this code. This code is written under the assumption that the run-time values of each of the inputs specified are available in an `inputs` object.

The inference function from Listing 4.9 iterates through all layer 2 inputs. If a data object matching the `emergency_veh` name is found, a `fused_likelihood_discrete()` method is invoked and returns a probability mass function specifying the likelihood of each possible value of the system-state, i.e., heavy, medium, light, given the current sensor data.

The specification of the `TrafficDemand` system-state is similar to the `EmergencyVehiclePresent` system-state.

Layer 4

Layer 4 of the domain model contains prior probabilities for the system-states. The `TrafficDemand` state is assigned a prior probability mass function (0.4,0.4,0.2) and the `EmergencyVehiclePresent` state is assigned a prior probability mass function (0.1,0.9). These priors indicate that traffic demand will be heavy or medium with probability 0.4 and light with probability 0.2. The emergency vehicle has a prior probability of being present of 0.1 and 0.9 of not being present. Layer 4 data is optional. If no priors are specified then non-informative or uniform priors are used. No planning algorithm has been specified for use by the domain modeller. The domain modeller is therefore allowing the planning model transformation to select an appropriate planning algorithm from the library.

4.8.1.2 Policy

The policy for this scenario is shown in Listing. 4.10. The policy file is used to specify the behaviour of an application and to determine the deployment region of the application.

The "global" scope attribute establishes the deployment region for the application. A global scope attribute indicates that the domain modeller wishes to apply the policy to all instances of the two system-states. Each system-state has in turn been assigned a

```

1 <policy>
2   <scope>global</scope>
3   <problem>planning</problem>
4   <subtype>single decision</subtype>
5   <state>
6     <name>EmergencyVehiclePresent</name>
7     <reward>
8       <type>boolean</type>
9       <range>>false-true</range>
10      <action>switch-phase</action>
11      <value>0-30</value>
12      <weight>1</weight>
13    </reward>
14  </state>
15  <state>
16    <name>TrafficDemand</name>
17    <reward>
18      <type>discrete</type>
19      <range>heavy-medium-light</range>
20      <action>switch-phase</action>
21      <value>10-5-1</value>
22      <weight>1</weight>
23    </reward>
24  </state>
25 </policy>

```

Listing 4.10: Excerpt from junction controller scenario policy definition.

scope associated with the phase infrastructure elements from layer 1 of the domain model. By specifying a policy with global scope, the domain modeller is ensuring that run-time components will be generated to support a state space measuring system-state values at all phase elements in the traffic system recorded in layer 1 of the domain model.

The problem and subtype attributes specify that the problem is a single-decision planning problem. This indicates that the application is to choose an action maximising the expected utility at each time step.

The two system-states from layer 3 are referenced in Listing 4.10. A reward attribute for each system-state specifies the information required to build a reward model for the application. The reward model information extracted from the policy is shown in Table 4.1.

The type attribute classifies each system-state as being: "discrete" or "continuous" and the range attribute lists the possible values that each system-state can produce. An action attribute is used to associate a "switch-phase" action with each reward. The value attribute provides a numeric value for each corresponding value of the system-state. For example the TrafficDemand system-state can take values: "heavy", "medium" and "light" with associated rewards 10, 5, 1. Finally, in line 22, both reward elements are assigned an equal weight.

The domain-model transformation combines all the reward values specified in the policy file linearly. This process varies for planning and for optimisation problems and is described in detail in Chapter 5.

Once the domain modeller has compiled a domain model and specified a policy for

4.8. CASE STUDIES

EmerVeh	True			False		
Traffic Demand	Heavy	Medium	Light	Heavy	Medium	Light
Switch-Phase	40	35	31	30	5	1

Table 4.1: Reward model for the junction controller scenario.

the scenario, the transformation algorithms can be used to generate the application code. Chapter 5 describes the operation of the transformations in detail and describes how application code in the form of control units, is generated for the Junction Controller scenario.

4.8.2 CCTV Camera Selection Scenario

The CCTV Selection scenario involves developing an application to optimise the use of CCTV camera infrastructure in Dublin city centre. The following scenario is assumed. In Dublin city there are hundreds of CCTV sensors placed at locations around the city. City Council staff monitor these cameras to detect and respond to traffic accidents and congestion. At any one time there may be up to 10 staff on duty to monitor streaming camera footage. There are 30 screens that can be used to monitor CCTV image streams.

The desired behaviour for the application being developed is to select the 30 most interesting CCTV data streams to display from the hundreds of available cameras. The criteria by which a CCTV camera is considered interesting, are defined to be a function of weather, the presence of pedestrians and traffic demand. There is a further requirement that the set of useful CCTV cameras should be chosen to also provide the maximal geographic coverage over the city road network.

This application therefore requires a bi-criteria optimisation algorithm. The problem is further complicated by the fact that the application is to be deployed in a pervasive computing environment and some of the data on the criteria are sourced from sensor infrastructure deployed in the region. This creates a requirement for handling uncertain data and supporting inference techniques to infer properties of the system for which we cannot directly measure.

4.8.2.1 Scenario Domain Model

Layer 1

The two scenarios share the layer 1 data representing the road network in Dublin city centre. Layer 1 of the domain model for this scenario contains a series of junctions containing traffic phases connected by road links. The infrastructure layer elements for this scenario mirror those described in the previous section.

Layer 2

Layer 2 of the scenario domain model includes three sensor elements and one actuator element. As in the previous scenario inductive loop sensors are used to measure traffic demand. Weather station sensors are modelled to provide information on current weather conditions throughout the region of interest. We assume that a stationary weather sensor

4.8. CASE STUDIES

is present at each junction and provides data on rain fall levels at each junction. We also assume that a stationary pedestrian presence sensor is present at each junction and provides data on the number of pedestrians present at each junction. The spatial attribute of the weather and pedestrian sensors is specified as an ellipse shape representing the area of coverage. The spatial attribute is specified using the SFS data standard used in the Junction Controller scenario.

A CCTV Camera actuator is assumed to be located at each junction to provide streaming images to council staff. If the algorithm selects this junction for display the CCTV camera is invoked to return a data stream. The layer 2 meta-data for this scenario are similar to those in the Junction Controller scenario so are not shown here.

Layer 3

We identify three system-states for this scenario. It should be noted that this is just one way to decompose the problem. The selection and modelling of system-states is a design choice faced by a domain modeller when developing any application.

- A `JunctionInterest` system-state is specified to be a function of weather, pedestrian presence and traffic demand. To specify further we say the degree of interest of a CCTV image stream is a monotonically increasing function of worsening weather conditions, increasing numbers of pedestrians and increasing traffic demand.

The specification of the `JunctionInterest` system-state is shown in Listing 4.11. We specify that the state is dynamic and partially observable. The value of the `JunctionInterest` system-state can fluctuate independently of actions taken by the system. The value of the system-state is also partially observable as its value is inferred from sensor data. The complexity attribute of this system-state is set to false.

The scope of the system-state is defined as being of type "element" and of value "junction". The domain-model transformation will create components to measure the value of this system-state at physical regions of the deployment environment in which junction infrastructure elements are located.

The inputs attribute in Listing 4.11 specifies data from layers 1 and 2 relevant to determining the value of the system-state. The layer 2 inputs include three sensor types that will be combined with the spatial attribute of individual junction elements to issue queries to middleware services to discover and invoke sensors providing relevant data.

Finally, the implementation attribute specifies that a module named "JunctionInterest", written in the python programming language, contains the inference function to be used to combine the inputs to produce an output value for the system-state. As discussed in Section 4.6.3.1, inference function logic is often highly domain specific and must be specified by the domain modeller.

The inference function referenced in the `JunctionInterest` system-state uses a Bayesian network to combine the inputs. The suitability of Bayesian networks as an appropri-

```
1 <systemState>
2   <id>001</id>
3   <name>JunctionInterest</name>
4   <description>This state measures the degree of interest of a single Junction<
   /description>
5   <properties>
6     <complexity>false</complexity>
7     <dynamism>true</dynamism>
8     <observability>partial</observability>
9   </properties>
10  <scope>
11    <type>element</type>
12    <value>junction</value>
13  </scope>
14  <inputs>
15    <layer1>
16      <name>id</name>
17      <name>geometry</name>
18    </layer1>
19    <layer2>
20      <name>rain</name>
21      <name>pedestrians</name>
22      <name>traffic_volume</name>
23    </layer2>
24  </inputs>
25  <!-- Implementations -->
26  <implementation>
27    <code>python</code>
28    <sourceRef>JunctionInterest</sourceRef>
29  </implementation>
30 </systemState>
```

Listing 4.11: JunctionInterest system-state definition.

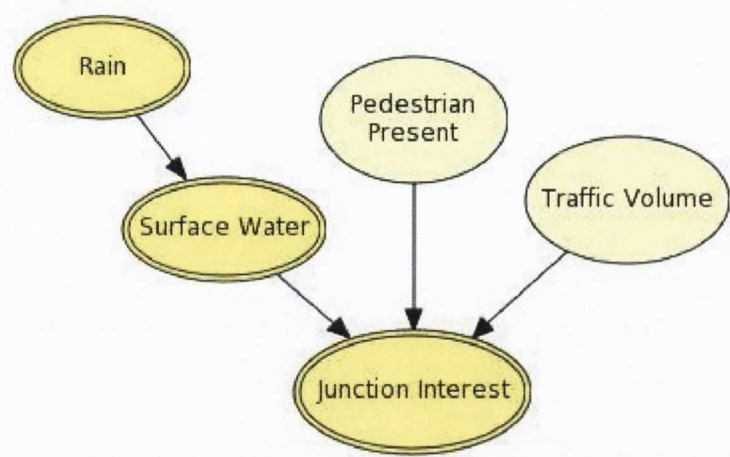


Figure 4.10: Bayesian network used to calculate junction-interest.

ate inference technique for pervasive computing applications is discussed in Chapter 2. The domain modeller uses the inference function to provide the structure and conditional probabilities of the Bayesian network shown in Fig. 4.10. The inference function is written at design time and referenced in the system-state specification. The domain model transformation loads a copy of the inference function into each state-variable object responsible for generating state space values at run-time.

At run-time the instantiated state objects issue middleware requests to discover sensors providing data matching the specified layer 2 inputs within their scope. Any sensor data found and returned is entered into the Bayesian Network and the updated value of the JunctionInterest variable is computed. This updated value is then available for use in application control units.

The Bayesian network designed for the CCTV camera selection is hybrid containing discrete and continuous valued nodes. The SurfaceWater, Rain and JunctionInterest are continuous. The Pedestrian node is Boolean indicating whether pedestrians are present or not and the TrafficVolume node is discrete and takes the values: "high"; "medium" and "low".

The Bayesian network is designed using the Hugin Bayesian Network Engine (Andersen et al. 1990). This tool provides a high level graphical interface for specifying a Bayesian network structure and associated probabilities. The Hugin file associated with this scenario is given in Appendix E.

The inference algorithms used in Hugin imposes constraints on the design of a hybrid Bayesian network (Jensen & Nielsen 2007, 95). Continuous nodes are assigned a conditional Gaussian distribution with a constant variance and a mean specified as a linear function of configurations of their discrete parents or a mean specified as a linear function of the mean value of continuous parents. A continuous variable with

4.8. CASE STUDIES

Rain		Surface Water		Traffic Volume	
Mean	0.0	Mean	0.0	Low	0.2
Variance	0.25	Rain	1.2	Medium	0.4
		Variance	0.25	High	0.4

Junction Interest						
Pedestrians	False			True		
Traffic Volume	Low	Medium	High	Low	Medium	High
Mean	1	3	5	3	5	7
Surface Water	5	5	5	5	5	5
Variance	2	2	2	2	2	2

Figure 4.11: Tables for junction-interest network.

no parents is assigned an unconditional normal distribution and no discrete variable may have continuous parents.

The conditional probabilities specified in the scenario network are summarised in Table 4.11. An example is $P(\text{SurfaceWater}|\text{Rain}) = N(1.2 \times \text{Rain}, 0.25)$. This can be read as saying that the continuous variable `SurfaceWater` has a mean value that is 20% higher than the values reported by the `Rain` sensor and a constant variance of 0.25. Such a specification might reflect the belief of the domain expert that the rain sensors in general underestimate the amount of surface water by 20%.

The distribution over `JunctionInterest` is constructed as a multivariate Gaussian over the continuous variables and a conditional Gaussian over the discrete parents. An excerpt from this distribution as specified in Table 4.11 reads:

$$P(\text{JunctionInterest}|\text{Pedestrian}, \text{TrafficVolume}, \text{SurfaceWater}) = N(1 + 5 \times \text{SurfaceWater}, 2) \text{ Pedestrian} = \text{False}, \text{TrafficVolume} = \text{Low}$$

which specifies that the value `JunctionInterest` is normally distributed with a mean of $(1 + 5 \times \text{the mean of the SurfaceWater variable})$ and a constant variance of 2 when there are no `Pedestrians` present and the `TrafficVolume` is low.

The value for `JunctionInterest` increases when `Pedestrians` are present, `TrafficVolume` is high, and is linearly dependent on the level of `SurfaceWater` present. Note that in this scenario the `JunctionInterest` is specified as a factor of 5 times the mean `SurfaceWater` level. This is shown in Table 4.11:

$$P(\text{JunctionInterest}|\text{Pedestrian}, \text{TrafficVolume}, \text{SurfaceWater}) = N(7 + 5 \times \text{SurfaceWater}, 2) \text{ Pedestrian} = \text{True}, \text{TrafficVolume} = \text{High}$$

The inference function is shown in Listing 4.12 and connects the Bayesian network to the sensor data accessed at run-time. Section 4.6.3.1 showed how the inference function accesses domain model and sensor data using the inputs API.

In lines 7-13 the function reads the Bayesian network structure shown in Fig 4.10 and Table 4.11. The network structure and conditional probabilities are specified by the domain modeller using the Hugin tool and written to a file called "cctvScenaio.net". In line 8, a Hugin domain object is created and used to hold all information associated with a Bayesian network. Domain objects contain node objects representing discrete and continuous random variables, decisions and utilities (Jensen 2007, 26).

In line 15 an `enter_sensor_data()` function is called which takes two arguments: a domain object and the set of layer 2 inputs defined for this system-state. The layer 2 inputs are a collection of data objects representing the sensor data discovered at run-time.

An `enter_sensor_data()` function is shown in lines 21-42. This function iterates through the sensor data objects. Each sensor data object is matched to a corresponding node in the Bayesian network. For a discrete sensor data object and a discrete Bayesian network node the sensor data is entered as a likelihood. For a continuous sensor data object and a continuous Bayesian network node the sensor data is entered as a mean value. In line 42 a Hugin function called `domain.propagate()` is invoked to calculate the posterior probability of the nodes given the sensor data entered (Jensen 2007, 113). Finally, in line 19 the `JunctionInterest` node is selected and its mean and variance are returned to the control unit.

- **DegreeOfInterest**

The application is required to select sets of 30 CCTV cameras that maximise the degree of interest and geographic coverage criteria. The `JunctionInterest` system-state described previously provides a measure of the degree of interest for an individual junction. A second system-state, `DegreeOfInterest` is now defined and used to combine `JunctionInterest` readings from sets of 30 individual junctions. The specification of this system-state is shown in Listing 4.13.

The properties attribute of the system-state mirrors that of the `JunctionInterest` system-state and is specified to be dynamic and partially observable and the complexity attribute is set to false.

The scope attribute defines the system-state to be of type "multi-element", number "30" and value "junction". The domain model transformation will create state space components that can measure the value of `DegreeOfInterest` system-states defined over sets of 30 junction elements. The spatial attribute of a `DegreeOfInterest` system-state is therefore generated at run-time and is the geometric shape created when the spatial attributes of 30 junction infrastructure elements are combined.

The inputs attribute in Listing 4.14 specifies a layer 3 input of name "JunctionInterest".

4.8. CASE STUDIES

```
1 from hugin import *
2
3 def calculate(inputs):
4     #Calculate Junction Interest.
5
6     pl = MyParseListener()
7     try:
8         domain = Domain("cctvScenario.net", pl)
9     except ExceptionIO:
10         log.error("Hugin_domain_could_not_be_constructed.")
11         raise
12     domain.compile()
13     domain.propagate()
14
15     enter_sensor_data(domain, inputs["layer2"])
16     interest_node = domain.getNodeByName("JunctionInterest")
17     interest_ccnode = interest_node.castToContinuousChance()
18
19     return interest_ccnode.getMean(), interest_ccnode.getVariance()
20
21 def enter_sensor_data(domain, inputs):
22     for datum in inputs:
23         for node in domain.getNodes():
24             if datum.name == node.getAttribute("HR_Desc"):
25                 #match found between discrete sensor data and discrete Bayesian
26                 network node
27                 if datum.type == "discrete" and node.getKind() == H.KIND.DISCRETE:
28                     :
29                     fused_likelihoods = datum.fused_likelihood_discrete()
30                     discrete_node = node.castToDiscreteNode()
31                     num_states = discrete_node.getNumberOfStates()
32                     for i in range(num_states):
33                         for (x,y) in fused_likelihoods:
34                             if x == discrete_node.getStateLabel(i):
35                                 #found node state that matches this entry in the
36                                 data prob. mass function
37                                 discrete_node.enterFinding(i, y)
38
39                 elif datum.type == "continuous" and node.getKind() ==
40                 H.KIND.CONTINUOUS:
41                     #match found between continuous sensor data and continuous
42                     Bayesian network node
43                     (mean, sig) = datum.fused_likelihood_continuous()
44                     ccnode = node.castToContinuousChance()
45                     ccnode.enterValue(mean)
46
47     domain.propagate()
```

Listing 4.12: Inference function for JunctionInterest system-state.

4.8. CASE STUDIES

```
1 <systemState>
2   <id>002</id>
3   <name>DegreeOfInterest</name>
4   <description>This state measures the degree of interest of a set of CCTV
5     cameras</description>
6   <properties>
7     <complexity>>false</complexity>
8     <dynamism>>true</dynamism>
9     <observability>partial</observability>
10  </properties>
11  <scope>
12    <type>multi-element</type>
13    <number>30</number>
14    <value>r_junction</value>
15  </scope>
16  <inputs>
17    <layer1>
18      <name>id</name>
19      <name>geometry</name>
20    </layer1>
21    <layer3>
22      <name>JunctionInterest</name>
23    </layer3>
24  </inputs>
25  <outputs>
26    <type>boolean</type>
27  </outputs>
28  <!-- Implementations -->
29  <implementation>
30    <code>python</code>
31    <sourceRef>DegreeOfInterest</sourceRef>
32  </implementation>
33 </systemState>
```

Listing 4.13: Excerpt from DegreeOfInterest system-state definition.

At run-time, state-space components will use the JunctionInterest datum name and the spatial attribute of DegreeOfInterest system-states to issue middleware queries for layer 3 elements of name "JunctionInterest" within in a geometry defined over sets of 30 junction elements. This process is analogous to the discovery and access of layer 2 sensor data at run-time.

The implementation attribute specifies that a module named "DegreeOfInterest", written in the python programming language contains the inference function used to combine the inputs to produce a value for the system-state.

The inference logic for this system-state is shown in Listing 4.14. This function iterates over the 30 layer 3 inputs summing the mean value of each individual Junction-Interest state. Note that only the mean values are used and no use is made of the variance of each estimate.

The complexity of the inference function of Listing 4.14 is $O(n)$ where n is the number of junction infrastructure elements, i.e., the complexity scales linearly with the number of junctions. Furthermore, this calculation is easily parallelizable as

4.8. CASE STUDIES

```

1 def calculate(inputs):
2     #sum the degree of interest from each junction
3     sum = 0
4     for datum in inputs["layer3"]:
5         sum += datum[0]
6
7     return sum

```

Listing 4.14: DegreeOfInterest inference function.

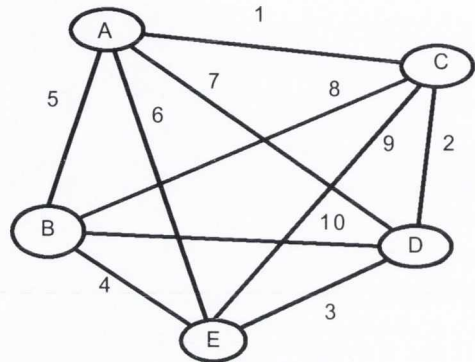


Figure 4.12: MaximalDistance calculation.

this criterion can be independently evaluated for each junction. As a result, the complexity property for this system-state is specified to be false.

- MaximalDistance

A third system-state is specified to compute the geographic spread of selected sets of 30 junctions. To measure the maximal-distance, the set of selected junctions are modelled as nodes in a fully connected network. The length of all network edges (distance between junctions) is measured in metres to obtain the total length of the network and used as a measure of coverage.

This approach is shown in Fig. 4.12 for a five-node network with 10 edges. More generally in a fully connected network of N nodes the number of edges E is calculated from $\frac{N*(N-1)}{2}$. The maximal distance is then

$$\sum_{i=1}^E L_i$$

where L_i is the length of the i^{th} edge in the network.

The specification for this system-state is shown in Listing 4.16. The properties attributes of this system-state are the inverse of all others presented in the two case studies. We specify the complexity attribute as true. The complexity of an exhaustive search of this space is $O(c^n)$ where n is the number of nodes, i.e., the complexity

4.8. CASE STUDIES

grows exponentially with increases in the problem size. This problem has similar characteristics to a brute force solution for the travelling salesman problem. When the two criteria are considered together the application is a multi-objective combinatorial optimisation problem of complexity $O(c^n + n)$.

This system-state is measuring a physical property of the road network and this property is fixed for a given set of junction infrastructure elements. The dynamism attribute is therefore set to false. The only input required to calculate the value of the system-state is the Cartesian coordinates of each junction and therefore no sensor data is required and the observability property is set to full.

The scope attribute of the MaximalDistance system-state is similar to that of the DegreeOfInterest system-state.

The implementation attribute specifies that a module named "MaximalDistance", written in the python programming language contains the inference logic used to combine the inputs to produce an output value for the system-state. The inference logic for the system-state is shown in Listing 4.16. This function iterates through the layer 1 inputs and extracts the (x,y) coordinates of the junction geometry. The length of line segments connecting each pair of junction coordinates is measured and summed to produce the system-state value.

4.8.2.2 Policy

The policy for this scenario is shown in Listing 4.17. The scope attribute is of value "global", indicating that the policy is to be applied to all instances of the system-states defined. The problem attribute specifies that the problem is an optimisation problem.

Of the three system-states defined, only two are referenced in the policy. When the policy is parsed by the domain model transformation, the system-state specifications for the MaximalDistance and DegreeOfInterest system-states are read. Listing 4.13 shows that a JunctionInterest system-state is a layer 3 input required to evaluate the value of the DegreeOfInterest system-state.

A reward attribute for each system-state specifies the information used to build a reward model for the application. The type attributes show that the two system-states referenced in the policy are continuous valued and that the optimisation algorithm should attempt to maximise the value of each system-state. The weight attribute specifies that each system-state is weighted equally when evaluating criteria. The reward model extracted from the policy file is used to create an objective function used by optimisation algorithms in application control units to evaluate selected sets of junctions.

4.8. CASE STUDIES

```
1 <systemState>
2 <id>003</id>
3   <name>MaximalDistance</name>
4   <description>This state measures the distance between the Cartesian
5     coordinates of
6     a set of CCTV cameras.</description>
7   <properties>
8     <complexity>true</complexity>
9     <dynamism>false</dynamism>
10    <observability>full</observability>
11  </properties>
12  <scope>
13    <type>multi-element</type>
14    <number>30</number>
15    <value>r_junction</value>
16  </scope>
17  <!-- Geometry SFS statement containing point associated with the junction -->
18  <inputs>
19    <layer1>
20      <name>id</name>
21      <name>geometry</name>
22    </layer1>
23  </inputs>
24  <!-- Implementations -->
25  <implementation>
26    <code>python</code>
27    <sourceRef>MaximalDistance</sourceRef>
28  </implementation>
```

Listing 4.15: Excerpt from MaximalDistance system-state definition.

```
1 def calculate(inputs):
2     from math import sqrt
3     import string
4     maxDist = 0; dist = 0
5     junctions = [string.split(string.strip(datum['geometry'], \
6     'POINT()'), ) for datum in inputs['layer1'] if datum['geometry']]
7
8     for x1,y1 in [[int(x), int(y)] for x,y in junctions]:
9         for x2,y2 in [[int(x), int(y)] for x,y in junctions]:
10             dx,dy=x1-x2,y1-y2
11             dist=sqrt(dx*dx + dy*dy)
12             maxDist += dist
13     return maxDist / 1000 #convert to kms
```

Listing 4.16: Inference function for MaximalDistance system-state.

```

1 <policy>
2   <scope>global</scope>
3   <problem>optimisation</problem>
4   <state>
5     <name>MaximalDistance</name>
6     <reward>
7       <type>continuous</type>
8       <range>0-10</range>
9       <value>maximise</value>
10      <weight>1</weight>
11    </reward>
12  </state>
13  <state>
14    <name>DegreeOfInterest</name>
15    <reward>
16      <type>continuous</type>
17      <range>0-1</range>
18      <value>maximise</value>
19      <weight>1</weight>
20    </reward>
21  </state>
22 </policy>

```

Listing 4.17: CCTV selection scenario policy excerpt.

4.9 Conclusion

This chapter has described how the domain model and policy is used to program a pervasive computing application. The next chapter describes the operation of the transformation algorithms on the domain model and policy. The transformation algorithms generate application code required to run planning and optimisation algorithms in pervasive computing environments. The next chapter also describes the role of AI modeller in developing and adding algorithms to the library.

The operation of the transformation algorithms is demonstrated using the case studies presented in this chapter.

— As an aside I would like to insert a warning to those who identify the difficulty of the programming task with the struggle against the inadequacies of our current tools, because they might conclude that, once our tools will be much more adequate, programming will no longer be a problem. Programming will remain very difficult, because once we have freed ourselves from the circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity.

Edjar Dijkstra The Humble Programmer ACM Turing Lecture 1972

5

Model Transformations and Code Generation

This chapter presents the transformation algorithms that parse the domain model and policy specified by the domain modeller to generate application code.

The first transformation algorithm, the "domain-model transformation", generates a planning model that provide abstractions to mask the dynamic and noisy nature of pervasive computing environments.

The second transformation algorithm, the "planning-model transformation" selects appropriate planning or optimisation algorithms for use in applications. The planning-model transformation uses planning model components to construct control units for the application. A control unit is a piece of executable that uses a planning or optimisation algorithm to control application behaviour in accordance with a specified policy.

This chapter is structured as follows. Section 5.1 describes the components of the planning model and the services provided by these components. In Section 5.2 the domain-model transformation logic is presented. This transformation parses the domain model and policy and produces a planning model. Section 5.3 discusses automated algorithm selection and the addition of new algorithms to the library of supported algorithms. Section 5.4 presents the planning-model transformation logic. Section 5.5 presents the operation cycle of the control units generated by the planning-model transformation. Finally Section 5.6 continues the presentation of the case studies from the previous chapter and demonstrates the use of the methodology to develop two pervasive computing applications. This section focuses on the generation of control units to manage the behaviour of the Junction Controller and CCTV Camera Selection applications.

5.1 Planning Model Components

This section describes the planning model components and the services they provide. The planning model components provide the interface between planning and optimisation algorithms and pervasive computing environments.

The planning model components are designed to implement an abstract model for partially observable domains provided by (Ghallab et al. 2004, 393). This abstract planning model was introduced in Chapter 2 and is a five-tuple $\Sigma = (S, A, T, O, R)$ where:

- $S = \{s_1, s_2, \dots\}$ is the set of system-states;
- $A = \{a_1, a_2, \dots\}$ is the set of actions;
- $T(s, a, s')$ represents a stochastic state-transition function that gives the probability $P(s'|s, a)$ of moving to state s' if the action a is performed in state s .
- $O = \{o_1, o_2, \dots\}$ is the set of observations that are produced by the sensor infrastructure in the region. An observation or sensor model function $O(s', a, o)$ gives the probability $P(o|a, s')$ of observing o if action a is performed and the resulting state is s' . This function describes how sensors are affected by the actual state of the system.
- $R(s)$ is the reward function and $R(s, a, s')$ represents the immediate reward the agent will get if it performs action a while in state s and moves to state s' .

The domain-model transformation algorithm parses the domain model and instantiates a set of components, collectively referred to as a planning model. The planning model components are used by the planning and optimisation algorithms in the library. The composition of planning model components depends on the information specified by the domain modeller. For some applications it may not be possible to specify a transition system because there is no mathematical model representing the open-loop dynamics of the system. For example, in an urban traffic control system, no model is likely to be built for the traffic flow and queues. Given the highly non-linear and uncertain aspects of human behaviour, this would be a virtually hopeless task in complex multiple-intersection networks (Spall 2003, 154).

Algorithms will use various subsets of planning-model components. For example, a state-transition system is required for a sequential planning algorithm formulated as a Markov Decision Problem but is not required for some stochastic approximation optimisation algorithms.

5.1.1 Modelling Partially-Observable State Spaces

The requirement to provide estimates of the changing value and certainty of application state data at run-time led to a design choice to represent application state information as variables.

5.1. PLANNING MODEL COMPONENTS

The domain-model transformation algorithm uses system-state specifications as templates to create planning model components that execute at run-time to provide values for the application state space to planning and optimisation algorithms. These components are referred to as state-variable objects. Each instance of application state data is represented as a state-variable object and is tagged with a spatial attribute indicating the region of the environment to which it relates.

The distinction between system-states and state-variable objects is important to understanding the operation of the transformation algorithm. System-states are used as templates, to create a high-level model of the state space of an application. State-variable objects on the other hand are used to provide run-time support to the application. They are the low-level implementation of the application state space.

State-variable objects provide an interface for the AI modeller to program against. State-variable objects have a spatial attribute indicating the deployment region over which they are defined and use their spatial attribute to issue topological queries to access the sensor and actuator data required to calculate state-space values.

State-variable objects provide an interface that is used by the planning and optimisation algorithms to determine the run-time value of a state at a location in the environment. State-variable objects return upon request a likelihood function over the values of the state at that location and at that instant in time.

There is a one-to-many relationship between system-states and state-variable objects. The number of state-variable objects created from each system-state is determined by deployment criteria embedded in the policy specified for each application. The policy scope and the scope of each individual system-state determine the number of state-variable objects. An instance of a state-variable object is created in the deployment environment wherever the value of a system-state needs to be resolved.

Chapter 4 described a factored state space, i.e., a state defined over a number of random variables. The random variables are elements from layers 1-3 of the domain model representing infrastructure and sensor/actuator data. Each state-variable object contains references to the random variables over which it is defined. At run-time each state-variable object is responsible for providing access to this random variable data.

5.1.1.1 Planning Model Sensor Fusion Support

State-variable objects provide sensor fusion and state inference services. By default state-variable objects perform low-level automated competitive fusion of sensor data. High-level inference functions defined by the domain modeller are executed by state-variable objects at run-time. Sensor fusion techniques are described in detail in Chapter 2. This section describes the run-time provision of low-level sensor fusion.

A state-variable object combines its spatial attribute with layer 2 input names and performs sensor and actuator lookup operations by invoking middleware services. A monotonic or direct mapping is assumed, between layer 2 input names and sensor data

types.

If multiple sensor readings are returned for a single layer 2 datum, they are combined to provide a fused likelihood function. The combination of multiple sensor readings is referred to as competitive fusion and was introduced in Chapter 2. The competitive fusion model implemented by state-variable objects is presented here.

Assume that a set of N independent sensor readings $\{y_1, y_2, \dots, y_n\}$ relating to the true value of a layer 2 input Θ is obtained from sensor infrastructure at time t . It is assumed that they come from a common (perhaps unknown) distribution f_θ , e.g., independent observations of the same value. The sensor model from layer 2 of the domain model provides a conditional probability $P(y_i|\Theta)$ i.e., the likelihood of the value of the layer 2 input given the sensor data observed.

Multiple sensor readings can be combined to give: $L(\theta) = f_\theta(y_1, y_2, \dots, y_n)$ and can be written as the product of the individual probability mass or density functions (Cowan 1998, 70):

$$L(\theta) = P(y_1|\theta) \times P(y_2|\theta) \dots P(y_n|\theta) = \prod P(y_i|\theta) \quad (5.1)$$

Equation 5.1 can be used to calculate a fused likelihood for discrete data.

Continuous data can be fused similarly. As an example consider the case where a single scalar observation y_1 from a normal distribution with a known variance σ^2 is received. The likelihood function is $p(y|\theta) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma^2}(y-\theta)^2}$

This can be specified using the parameters $\{\mu, \sigma\}$ alone.

If another observation y_2 becomes available, it can be combined with the existing one to produce an updated or fused likelihood function:

$$L(\theta) = \prod_{i=1}^2 P(y_i|\theta)$$

The normal distribution is conjugate in that the product of two normal distributions is also normal and can be computed easily. The parameters of a fused likelihood function $N(\mu, \sigma)$ can be calculated from N pieces of sensor data as:

$$\hat{\mu} = \frac{\sum_i y_i / \sigma_i^2}{\sum_i 1 / \sigma_i^2} \quad \text{and} \quad \hat{\sigma} = \frac{1}{\sum_{i=1}^n (1 / \sigma_i^2)} \quad (5.2)$$

Under the assumption of normally distributed errors this is also the maximum likelihood estimate, the weighted least squares estimate, and the linear estimate whose variance is less than that of any other linear unbiased estimate (Maybeck 1979). It can also be calculated as a weighted mean $\hat{x} = \sum_{i=1}^n w_i y_i$ where the weight w_i is $1/\sigma_i^2$. Equation 5.2 can be used to calculate a fused likelihood for continuous data.

A fused likelihood value is automatically calculated by state-variable objects for each layer 2 input. The fused likelihood values are embedded in Data objects.

Multiple sensor readings received for a single layer 2 input are assumed to be independent of each other. We assume that the middleware can select sets of sensors within the deployment region without duplication.

Providing support for automatic competitive fusion is appropriate given the nature of

pervasive computing environments. The sensor and actuator infrastructure may be mobile so it may not be possible to determine at design time which sensors and actuators will be available. Due to sensor mobility, there may be regions of the deployment environment with a proliferation of sensors and others regions with few or no sensors. A region with many sensors providing data for the same state input should have a lower uncertainty than a region with less sensor data.

Data Objects

A data object is created for each layer 2 input specified in a system-state definition. When a state-variable object issues a discovery query for this datum, all readings returned by sensors are loaded into a single data object. Data objects provide functions to combine all readings to create a fused value for a layer 2 input.

Data objects can model discrete and continuous data. Discrete data objects contain a list of tags and a list of likelihoods. The tags enumerate the range of values the discrete data can take and the list of likelihoods provides a measurement of how likely each possible value is given the most recently received sensor data. For example a discrete data object taking values: "high", "medium" and "low" might have a possible set of likelihoods: 0.05, 0.10, 0.85 indicating a high certainty that the value is indeed "low".

Continuous data objects provide a mean and a variance attribute describing the distribution of the sensor data. The current design assumes continuous data is normally distributed.

Data objects contain the following attributes and methods:

- type - discrete or continuous.
- tags[] - categories for discrete data for e.g., "heavy" "medium" "light".
- data[] - mean values for continuous data.
- likelihoods[] - conditional probabilities for discrete and variance for continuous data.
- fused_likelihood_discrete() - this method implements Eqn. 5.1. It assumes independence of low-level sensor data and returns a dictionary of tags and associated likelihoods of a discrete layer 2 datum.
- fused_likelihood_continuous() - this method implements Eqn. 5.2. It assumes normality and independence of sensor data and returns a tuple containing the mean and variance of a continuous layer 2 datum.

Competitive fusion example

To clarify the competitive fusion support provided by state-variable objects, the following example from the Junction Controller scenario is presented. Consider the code in Listing 5.1. This code is the inference function referenced in a system-state definition. In line 4 the results of a fused_likelihood_discrete() function call are returned to a planning algorithm. This call engages a series of middleware invocations and sensor fusion operations, the results of which are shown here.

5.1. PLANNING MODEL COMPONENTS

```
1 def calculate(inputs):
2     for datum in inputs["layer2"]:
3         if datum.name == "traffic_demand":
4             return datum.fused_likelihood_discrete()
```

Listing 5.1: TrafficDemand inference function.

in get_layer2_values results are:

```
[Data Object 'traffic_demand',
type: 'discrete',
readings: ['light', 'light', 'light', 'medium', 'medium', 'light',
'light', 'heavy', 'light']]
```

['heavy']

combining data with tag and likelihood: "heavy" 0.75

1 sensor providing data "heavy" results in confirming evidence of 0.75

6 pieces of conflicting sensor data "light" received with likelihood 0.75

after processing "light" the likelihood is 0.00018310546875

2 pieces of conflicting sensor data "medium" received with likelihood 0.75

after processing "medium" the likelihood is 1.14440917969e-05

['light']

combining data with tag and likelihood: "light" 0.75

6 sensors providing data "light" results in confirming evidence of 0.1779

1 piece of conflicting sensor data "heavy" received with likelihood 0.75

after processing heavy the likelihood is 0.0444946289062

2 pieces of conflicting sensor data "medium" received with likelihood 0.75

after processing "medium" the likelihood is 0.00278091430664

['medium']

combining data with tag and likelihood: "medium" 0.75

2 sensors providing data "medium" results in confirming evidence of 0.5625

1 piece of conflicting sensor data "heavy" received with likelihood 0.75

after processing "heavy" the likelihood is 0.140625

6 pieces of conflicting sensor data "light" received with likelihood 0.75

after processing "light" the likelihood is 3.43322753906e-05

state object returning normalised likelihoods from get_value()

```
[('heavy', 0.004048582995951417),
```

```
('light', 0.98380566801619429),
```

```
('medium', 0.012145748987854251)]
```

5.1. PLANNING MODEL COMPONENTS

Output from `fused_likelihood_discrete()` function.

This is a trace from the run-time evaluation of a control unit from the Junction Controller scenario. It shows multiple sensor readings returned for the discrete datum `traffic_demand`. In this example nine sensors are found in the state scope of which six provide a reading "light", two provide a reading "medium" and one provides a reading "heavy".

Sensor data for each of the three possible values are combined by calculating:

$$\text{fused_likelihood} = \text{likelihood}^{\{\text{count}\}}$$

where `likelihood` is the confidence in the sensor reading and `count` is the number of confirming readings. This produces an unnormalised likelihood in support of the value. This likelihood is refined by taking into account any conflicting sensor readings. Conflicting sensor readings are combined:

$$\text{fused_likelihood} \times (1 - \text{likelihood})^{\{\text{count}\}}$$

where $(1 - \text{likelihood})$ is the probability of the conflicting sensor giving an erroneous reading. When fused likelihoods are calculated for each possible value of the layer 2 input they are normalised and passed to the higher-level inference function specified by the domain modeller.

In this example the normalised fused likelihoods for the datum `traffic_demand` are ('heavy', 0.004), ('light', 0.984), ('medium', 0.012). These values represent the likelihoods associated with a `traffic_demand` state object at an individual junction given the then available sensor data. These likelihood values are returned to the planning algorithm.

5.1.1.2 Planning Model State Inference Support

State-variable objects provide a high-level inference function which combines all input data from layers 1-3 specified in the system-state definition and returns a likelihood function for the current value of the state. High-level state inference is also referred to as feature- or decision-level inference and was introduced in Chapter 2. The domain modeller is required to specify the inference function logic for each system-state at design time. Chapter 4 describes the domain modeller's role in specifying the inference function logic.

The domain-model transformation copies the inference logic into each state-variable object instantiated. Listing 5.1 shows a very simple fusion function. More complex examples of fusion functions using layer 1, 2 and 3 data are shown in Listings 4.16, 4.9 and 4.14 respectively.

State-variable objects provide a decentralized inference topology as shown in Fig. 2.8 (b). Inference is performed on a state-by-state basis. The topology is dependent on the system-state scope and each state-variable object associated with the system-state implements low-level competitive sensor fusion and high-level state inference algorithm.

5.1. PLANNING MODEL COMPONENTS

The state inference function provided by a state-variable object is invoked by planning and optimisation algorithms to access application state. State-variable objects provide a `get_value()` method that collects layer 1-3 input data and invokes the state inference function.

Layer 1 input data is known at design time and written into state-variable objects when they are instantiated. The `get_value()` method then performs competitive sensor fusion for all specified layer 2 inputs and stores the results in a collection of sensor data objects. The `get_layer3_values()` method then uses middleware queries to find state-variable objects within scope that are associated with specified layer 3 inputs.

The layer 1-3 data are packed into an `inputs` object and passed to the inference function. The structure of the `inputs` object and the interface provided to access its data are described in Chapter 4.

Planning and optimisation algorithms can combine likelihood values returned by state-variable objects with any available prior probabilities to calculate posterior probability distributions over the application state space. The combination of prior probabilities and likelihood functions is compatible with a Bayesian inference framework as discussed in Chapter 2.

The use of likelihood data varies across planning and optimisation algorithm types. For example, a Bayesian decision network containing chance nodes will enter the sensor data readings as a likelihood with an associated uncertainty, whereas optimisation algorithms may not make use of the uncertainty of data and may operate only the mean value or most likely value returned by the inference function.

5.1.2 Optimisation Algorithm Support

The generation of state-variable objects varies across optimisation and planning problem types. For planning problems the domain-model transformation algorithm instantiates the set of state-variable objects required to model the application state space. The set of required state-variables is known at design time because the design of planning-problem control units is formulated so that the state space is enumerable at design time.

However the design of optimisation control units does not permit this approach. For complex optimisation problems the state space will often be too large to enumerate fully and the overhead of creating a full set of state-variable objects is impractical. As described in Chapter 2, heuristic optimisation algorithms manage complexity by exploring random subsets of an application state space.

The domain-model transformation creates `State-Generator` objects for optimisation problems. State-generator objects are state-variable factories used by optimisation algorithms to produce state-variable objects on demand at run-time. They are provided to facilitate the exploration of complex application state spaces. State-variable objects generated for optimisation problems are functionally identical to those used in planning problems.

State-generator objects provide support for generating random sets of state-variable

5.2. DOMAIN-MODEL TRANSFORMATION ALGORITHM

```
1
2 def initialise_states()
3     # create all initial candidates for optimisation problems
4     # if there is a mix of single and multi element then we the same random
      scopes are used across all elements
5
6 def deploy()
7     # connect objects to nameserver
8
9 def generate_state(number, system_state)
10    # generate number of state objects for the system_state template
```

Listing 5.2: Python State-Generater API for optimistaion problems.

objects for a given system-state template. A collection of state-variable objects evaluated by an optimisation algorithm is referred to as a candidate solution.

The State-generator API is shown is Listing 5.2 and provides the following methods:

- `initialise_states()`

This method instantiates a set of state-variable objects providing values for the system-states specified by the domain modeller in the policy. The set of state-variable objects provides an initial candidate solution to the optimisation problem.

The set of state-variable objects is created to measure application state space at random areas of the deployment environment. This is done by randomly selecting system-state scope values and creating state-variable objects with corresponding spatial attributes. The newly created state-variable objects then discover and access relevant sensor and actuator infrastructure within their scope and return these values to the optimisation algorithm.

- `generate_state(number, system_state)`

This method generates the specified number of state objects from the given system-state template.

- `deploy()`

This method registers each newly created state object with the middleware services developed for evaluating the generated control units. The evaluation platform is described in Chapter 6.

5.2 Domain-Model Transformation Algorithm

The domain-model transformation algorithm parses the policy and domain model and populates a planning model with the extracted information. The planning model provides an algorithm-independent representation of the available knowledge of the problem structure.

5.2. DOMAIN-MODEL TRANSFORMATION ALGORITHM

The transformation algorithm iterates over the set of system-states for which the domain modeller has specified a policy and identifies the infrastructure layer and the sensor and actuator layer data that are relevant to the application.

The planning model that is produced, contains components designed to be used at run-time to interface with middleware services to access the data required for sensor fusion and provides a state-inference interface against which planning and optimisation algorithms are programmed against.

The inputs to this algorithm are a domain model and a policy. The transformation algorithm logic is presented in Algorithm 6.

5.2. DOMAIN-MODEL TRANSFORMATION ALGORITHM

Input: D: a conformant domain model; P: a policy
Output: Planning Model

```
1 Create Planning Model Structure;
2 Open and validate policy;
3 Read system scope and system-states from policy;
4 Get problem class and subtype;
5 foreach system-state within policy scope do
6   Open and validate each system-state specification;
7   Create system-state template;
8   Read state scope;
9   Read properties: complexity; observability; dynamism;
10  Read layers 1, 2 and 3 input variables;
11  Read inference function reference;
12 end
13 if problemClass == planning then
14   foreach system-state do
15     Calculate the number of state-variable objects required;
16     Create the set of state-variable objects;
17     Load layer 1 element data into each state-variable object;
18     Set the spatial attribute for each state-variable object;
19     Load layer 2 and 3 input data types into each state-variable object;
20     Load inference function reference into each state-variable object;
21     Add system-state template and associated set of state-variable objects to
        planning model;
22   end
23   Open actuator elements and validate schema;
24   foreach actuator do
25     foreach action in actuator do
26       Get cost, confidence and transition targets;
27     end
28     Add actions to planning model;
29   end
30   Generate reward model and add to planning model;
31   Write domain specific knowledge to planning model;
32 end
33 else if problemClass == optimisation then
34   Create and write system-state templates to planning model;
35   Generate reward model and add to planning model;
36   Add domain specific knowledge;
37   Create a state-generator object and add to planning model;
38 end
```

Algorithm 6: Domain-model transformation logic.

The major steps of the domain-model transformation algorithm are:

1. Parse the policy file. (Lines 2-4)

The policy file is validated using the policy XSD schema. The policy scope is read and indicates the extent of the region over which the application is to be deployed. The problem type and subtype are read. The type will be either planning or optimisation. This choice determines the composition of planning model components. The set of relevant system-states are read from the policy.

2. Parse each system-state specification. (Lines 5-11)

From lines 5-11 the algorithm iterates over the system-state elements referenced in the policy file. Each system-state definition is validated against the XSD schema for layer 3 elements. A template object is created for each system-state. Template objects hold the information that has been specified for system-states and are used when generating state-variable objects. The scope and properties of each system-state are recorded. Complexity, dynamism and observability properties are read for each system-state. The set of layer 1, 2 and 3 inputs are read for each system-state and a reference to its inference function is recorded. This function is used to combine the layer 1, 2 and 3 inputs to produce a value for the state. The transformation algorithm now diverges to handle planning or optimisation problems.

3. Planning Problem 1. (Lines 14-21)

In line 15, the algorithm calculates the required number of state-variable objects. State-variables are planning model components that provide the values of the system-states in the deployment environment. For a system-state with a scope of type infrastructure element, a state-variable object is created for each matching element within the policy scope.

In line 16, the set of state-variable objects are instantiated and added to the planning model. Because layer 1 data is static and is specified in the domain model at design time, it is now available and is written into state-variable objects corresponding to the system-state. Layer 2 meta-data is read and used to create topological queries that are also written into the sets of state-variable objects. These queries are used to invoke middleware services at run-time to discover and access sensor and actuator data.

In line 20 a copy of the inference function provided by the domain modeller is loaded into each state-variable object instance. The state-variable objects are now written to the planning model. The set of state-variable objects provide support for accessing state space values at run-time.

4. Planning Problem 2. (Lines 23-32)

In line 23, each actuator element specified in layer 2 of the domain model is validated

against the XSD schema for layer 2 actuators. Each actuator specification contains the actions provided and the cost and confidence associated with invoking them. Actuator meta-data is available at design time for static actuator infrastructure. Actuator objects are created and added to the planning model.

In line 30, a reward model is built using the reward elements contained in the policy, and is written to the planning model. The policy specification is described in Chapter 4. The reward model is used as an objective function to determine the utility of state-variable values. For planning problems, the reward model entries provide utility values for state-action combinations ($R(s, a, s')$) over the range of possible values of the state variables. It is structured as a multi-layered lookup table of the form: [state $\rightarrow$ range $\rightarrow$ value].

Any domain-specific knowledge specified at layer 4 of the domain model is written to the planning model and stored using associative arrays with valid key names consisting of system-state names and the "algorithm" keyword.

5. Optimisation Problem. (Lines 35-40)

System-state template objects are created and written to the planning model. These templates are used to produce instances of system-states when required.

In line 35 a reward model is generated and written to the planning model. The reward model structure for optimisation problems differs from that used in planning problems. Optimisation problems use the reward model to determine the relative weighting for system-states and whether the system-state values are to be maximised or minimised.

In line 37, any domain-specific knowledge contained in the domain model is added to the planning model. Finally a state-generator object is instantiated and written to the planning model. State-generator objects are used by optimisation algorithms at run-time to explore application state space. Given a system-state template, a state-generator object can instantiate state-variable objects to measure the value of the system-state at regions of the deployment environment.

The planning model is now populated with all domain model data and is now available as an input to the planning-model transformation algorithm.

5.3 Algorithm Library and Selection Support

Algorithm selection is the task of automatically choosing an algorithm from the library to apply to a problem. It is invoked if the domain modeller does not specify which algorithm to use. Algorithm selection is a function of problem type and domain characteristics.

The problem types addressed are planning and optimisation problems. Planning problems are subdivided into single and sequential decision problems. For optimisation problems the library currently provides support for stochastic or heuristic optimisation.

5.3. ALGORITHM LIBRARY AND SELECTION SUPPORT

A library of algorithms have been integrated with the methodology tool-chain. The library of supported algorithms is shown in Table 5.1. Multiple algorithms can be used for each problem. The algorithms listed in bold in Table 5.1 are the ones automatically selected.

As algorithms are added to the library, a mapping function is defined by an AI modeller to transform planning-model components into algorithm components. Chapter 2 contains a survey of the algorithms added to the library - a selection of which are used in the case studies presented in Chapters 4, 5 and 6.

The characteristics of pervasive computing environments relevant to the selection of algorithms are: dynamism, complexity and partial observability. Each of these characteristics is defined and discussed in Chapter 2.

Algorithm selection is performed by selecting all algorithms matching the problem type. This selection is then refined using the domain properties of the system-state templates in the planning model. Usually an application will contain a number of system-states which may have different sets of properties.

The interpretation of these properties varies according to the problem type. For both planning and optimisation problems, if any of the system-states are dynamic then an online algorithm is chosen. For optimisation problems, if any of the system-states are complex then heuristic algorithms are chosen. The interpretation of partial-observability varies according to the AI modeller who integrates new algorithms into the library. For planning problems with partial observability, noise-tolerant planning algorithms are selected. For the optimisation case study, the uncertainty of application state is ignored and the mean or most likely estimate of state values as returned by state-variable objects is used to evaluate candidate solutions.

Inference Problems				
Type	Principle	Properties	Algorithm	
Competitive Feature & Decision Level	Fused Maximum Likelihood Estimate	Partial-Observability	Weighted-Average Mean	
	Bayesian Inference	Partial-Observability	Bayesian Network	
Planning Problems				
Type	Principle	Properties	Algorithm	
Single Decision	Maximum Expected Utility	All	Bayesian Decision Network Random Action Selection Round Robin	
Sequential-Decision	Markov Decision Process (MDP)	Partial-Observability	POMDP	
Sequential-Decision	MDP	Dynamism	Online POMDP	
Sequential-Decision	MDP	Complexity	Online Approx POMDP	
Optimisation Problems				
Optimisation	Stochastic Search	Dynamism	Local Search	
Optimisation	Stochastic Search	Complexity	Simulated Annealing	

Table 5.1: Algorithm library and taxonomy.

5.3. ALGORITHM LIBRARY AND SELECTION SUPPORT

Table 5.1 is divided into three sections detailing the library algorithms provided for inference, planning, and optimisation problems. The first column of each section lists the problem type. The second column shows the principle upon which the solution is derived. The third column lists the system-state properties deemed relevant to selecting an algorithm and the fourth column lists the algorithms provided for this combination of problem and system-state properties.

For example in Table 5.1, competitive inference problems are addressed using fused maximum likelihood estimates provided using a weighted average mean algorithm in partially observable environments. Data objects described in Section 5.1.1.1 provide this service.

Feature/decision-level state inference problems, in which multiple pieces of sensor data must be fused, are addressed using Bayesian inference. A Bayesian network algorithm has been integrated into the library and is used for these problems. The mapping between the Bayesian network algorithm and the planning model is illustrated in Listing 4.12. The use of Bayesian network algorithms requires some expertise from the domain modeller. However the use of Bayesian networks is optional and the domain modeller can decide to use another inference technique at their discretion.

All single decision problems are addressed using the principle of maximum expected utility and implemented using Bayesian Decision Networks, which are described in Chapter 2.

Sequential decision planning problems are addressed using an MDP framework. If the environment is partially observable and dynamic, then an online approximate POMDP algorithm is chosen.

Solutions to optimisation problems are based on heuristic algorithms. If the environment is dynamic a stochastic approximation algorithm is chosen. For applications with complex state spaces, a simulated annealing algorithm is chosen.

The association of problem type to algorithm selection is informed by reference to the literature in Chapter 2 and by empirical testing of algorithms and problem types shown in Chapter 6. The selection can be amended by editing the taxonomy XML file shown in Listing 5.3.

If an algorithm is selected automatically then algorithm parameters are set using the default values provided by the AI modeller. Default values for parameters are provided when the initial mapping from algorithm to planning-model components is defined by the AI modeller. If the domain modeller specifies which algorithm they wish to use, they may also specify values for the free parameters using the domain-specific knowledge layer of the domain model.

There may be a significant trade-off between the ease of using the automated transformations and the loss of performance when compared to a hand-crafted solution from a planning or optimisation expert. This trade-off is explored using the results of the evaluation case studies presented in Chapter 6.

The taxonomy is implemented in XML and shown in Listing 5.3. The AI modeller can

5.3. ALGORITHM LIBRARY AND SELECTION SUPPORT

implement multiple algorithms for each problem type and property set and can specify which algorithm to use in the domain model. However if automatic algorithm selection is used then only algorithms referenced in the taxonomy are considered for selection. The algorithms available for automatic selection are shown in bold print in Table 5.1.

5.3.1 Extending the Algorithm Library

The algorithm library is extensible so that existing algorithm implementations can be replaced or new algorithms added to the library as required. The AI modeller must have the specialist knowledge required to implement planning and optimisation algorithms. The planning-model components provide an interface between pervasive computing environments and the planning and optimisation algorithm implementations.

To add a new algorithm to the library, the AI modeller preforms the following steps:

1. add a new entry to the algorithm taxonomy XML file specifying the problem type and environment properties for which the algorithm is suitable.
2. provide an implementation of the algorithm.
3. provide values for free parameters associated with the algorithm.
4. provide a function mapping the planning-model components to the new algorithm.

The planning and optimisation algorithms and their mapping functions are provided in high-level programming language libraries that are resolved using a conventional naming scheme. The planning-model transformation searches for algorithm implementations and mapping functions respectively, by importing programming library modules named `initialise_{algorithm_name}` and `generate_{algorithm_name}`.

Once an algorithm is added to the library as above, the algorithm is available for use in all problems matching the specified type and domain properties, that are modelled using the domain model. In this fashion, the current planning-model transformation prototype described in the case studies, can generate control units providing decision networks or simulated annealing algorithms for all planning and optimisation problems respectively, which can be modelled using the domain model.

5.3.1.1 Algorithm Engineering Example

To illustrate the process of adding algorithms to the library, this section describes how an AI modeller integrates the SMOSA algorithm into the library.

The SMOSA algorithm is presented in Algorithm 5 in Chapter 2. It is a multiple-objective optimisation algorithm based on simulated annealing. Table 5.1 shows that this algorithm is automatically selected for complex optimisation problems in dynamic environments.

The AI modeller performs the following steps:

```

1 <library>
2   <algorithm>
3     <name>MEU</name>
4     <problemClass>
5       <type>planning</type>
6       <subtype>single decision</subtype>
7       <properties>
8         <uncertainty>true</uncertainty>
9         <dynamism>true</dynamism>
10        <complexity>false</complexity>
11      </properties>
12    </problemClass>
13    <implementation>file</implementation>
14  </algorithm>
15  <algorithm>
16    <name>AO-POMDP</name>
17    <problemClass>
18      <type>planning</type>
19      <subtype>sequential decision</subtype>
20      <properties>
21        <uncertainty>true</uncertainty>
22        <dynamism>true</dynamism>
23        <complexity>true</complexity>
24      </properties>
25    </problemClass>
26    <implementation>file</implementation>
27  </algorithm>
28  <algorithm>
29    <name>SMOSA</name>
30    <problemClass>
31      <type>optimisation</type>
32      <subtype>heuristic</subtype>
33      <properties>
34        <uncertainty>true</uncertainty>
35        <dynamism>true</dynamism>
36        <complexity>true</complexity>
37      </properties>
38    </problemClass>
39    <implementation>file</implementation>
40  </algorithm>
41  <algorithm>
42    <name>SA</name>
43    <problemClass>
44      <type>optimisation</type>
45      <subtype>heuristic</subtype>
46      <properties>
47        <uncertainty>true</uncertainty>
48        <dynamism>true</dynamism>
49        <complexity>false</complexity>
50      </properties>
51    </problemClass>
52    <implementation>file</implementation>
53  </algorithm>
54 </library>

```

Listing 5.3: Algorithm taxonomy.

5.3. ALGORITHM LIBRARY AND SELECTION SUPPORT

```
1 for temp in coolingSchedule:
2     done = False
3
4     candidate.peturb_candidate(20)
5     candidate.deploy()
6     candidate.get_values()
7
8     if numEvaluations >= maxEvaluations:
9         done = True
10
11     numEvaluations += 1
12     archived = archive.submit(candidate.solution) #submit for archiving
13     #if archived then use it as the current solution
14     if archived == True:
15         archive.prune() #remove all dominated solutions
16         current = archive.current
17     else:
18         #probabilistically accept this solution
19         p = Pb(current, candidate.solution, temp)
20         r = random.random()
21         if p < r:
22             #metropolis proposal accepted
23             current = candidate.solution
24
25     #deregister state objects from middleware
26     candidate.shutdown()
27     if done: break
```

Listing 5.4: Execution cycle of SMOSA optimisation control unit.

1. the taxonomy XML file is edited to specify the algorithm name and selection criteria.
2. an implementation of the algorithm is provided. An extract from the implementation of the SMOSA algorithm is shown in Listing 5.4. This listing shows the execution cycle logic of the SMOSA algorithm. The logic follows the conceptual model of optimisation control units described in Section 5.5.2.

Prior to the execution of the loop in line 1 the SMOSA algorithm parameters are initialised and an initial candidate solution is generated. A candidate solution is a collection of state-variable objects evaluated by an optimisation algorithm.

From line 1, each execution of the loop is run with a temperature variable from the annealing schedule. The annealing schedule controls the exploration rate of the SMOSA algorithm.

The planning model-components shield details of the pervasive computing environment, allowing the developer to focus on specifying the algorithm logic. The optimisation control unit will run the candidate generation, execution and evaluation cycle for a specified number of evaluations. In lines 12-16, the solution is evaluated by submitting it to a solution archive. The archive is the set of non-dominated Pareto optimal solutions found by the control unit. Pareto-optimality is described in Chapter 2.

5.3. ALGORITHM LIBRARY AND SELECTION SUPPORT

Lines 18-23 show a Metropolis step that probabilistically accepts dominated solutions. The Metropolis algorithm is described in Chapter 2.

The SMOSA algorithm ceases exploring the application state space after a specified number of evaluation cycles.

3. default values are provided for SMOSA parameters. The parameters associated with this algorithm are:
 - `maxEvaluations`. This parameter controls the termination condition for deciding when to stop searching for a solution and return the best candidate found to date.
 - `StartTemp`. The higher the initial temperature the greater the degree of exploration of the state space. More specifically, a higher temperature increases the acceptance probability of a candidate solution with a lower utility in order to escape local minima. See Section 2.5.1 for a description of this.
 - `alpha`. This parameter specifies the rate at which the temperature is reduced over the lifetime of the algorithm. The `StartTemp` and `alpha` parameters are used to build a cooling schedule for the control unit.

4. a function mapping planning model components to algorithm components is provided. The planning-model transformation algorithm executes the mapping function. By convention mapping functions are named `generate_{algorithm_name}`.

A candidate solution is a collection of state-variable objects evaluated by an optimisation algorithm. To ease the task of developing optimisation algorithms, a `Candidate` class is provided to AI modellers. This is implemented as a wrapper around a state-generator planning model component. It provides an API that is very similar to the state-generator API described in Section 5.1.3. A candidate object provides a `get_values()` method that returns the values of all state-variable objects contained in the candidate solution. It contains `deploy()` and `shutdown()` methods to connect and disconnect state objects to middleware services. It provides a `perturb_candidate()` method used to stochastically generate new candidate solutions for evaluation. The `Candidate` class can be reused across optimisation algorithm implementations.

Lines 4-6 show a candidate object used to perturb or randomly alter 20% of the current candidate solution, deploy the state-variable objects contained in the solution and to obtain the values required to evaluate the candidate solution.

For the AI modeller these three API calls are sufficient to generate a collection of state-variable objects that conform to the system-states in the domain model, deploy them at run-time in a pervasive computing environments, and engage the competitive- and decision-level fusion algorithms to return posterior distributions over the values given available sensor and actuator data.

The reward model generated from the policy, is encoded as a multi-dimensional evaluation function with dimensionality equal to the number of system-states referenced in the policy file. Candidate solutions are evaluated along each dimension with direct comparisons between values of successive candidates.

5.4 Planning-Model Transformation

If the domain modeller has specified an algorithm type to use then the control unit is instantiated with the specified algorithm. Otherwise the planning model transformation can use the problem type and domain characteristics to select an algorithm from the library. This selected algorithm is then used when instantiating a control unit.

The transformation-algorithm logic is presented in Algorithm 7. It is used to generate a control unit for a pervasive computing application. The inputs are a planning model, a library of algorithms and a taxonomy mapping problem types to algorithms.

Input: PM: a planning model, L: algorithm library, T: algorithm taxonomy
Output: Control Unit

- 1 Open and validate algorithm library ;
- 2 Extract algorithm selection logic from taxonomy;
- 3 Unpack planning model;
- 4 Get problem and subtype;
- 5 **foreach** *system-state template in planning model* **do**
- 6 Read system-state properties;
- 7 Build index into taxonomy;
- 8 **end**
- 9 Select planning or optimisation algorithm from taxonomy;
- 10 Map algorithm to planning model components ;
- 11 Generate control units;

Algorithm 7: Planning-model transformation logic.

The major steps of the planning-model transformation algorithm are:

1. Obtain algorithm selection logic. (Lines 1-2)
The algorithm library is validated using the library XSD schema. A list of the available planning and optimisation algorithms is made. Information indexing algorithms by problem type and environment properties is read from the algorithm taxonomy.
2. Determine the problem type and domain characteristics. (Lines 3-7)
The problem attribute is read to determine whether the planning model has been generated for a planning or optimisation problem. The complexity, observability, and dynamism attributes are read for each system-state template contained in the planning model. Any values have been set by the domain modeller are used. Otherwise the observability and dynamism properties are determined as described in Section 5.4.1.

3. Algorithm Selection. (Line 9)

The problem and subtype attributes in system-state templates are used to identify the root branch of the algorithm taxonomy and the domain characteristics are used to select a particular algorithm from the taxonomy of Fig 2.1. The domain modeller can manually specify the algorithm type to be used in layer 4 of the domain model. If no algorithm type is specified the planning-model transformation selects one using the problem type and domain characteristics. Algorithm selection is discussed in detail in Section 5.3.

4. Algorithm Engineering. (Lines 9-10)

The algorithm library contains an implementation of each algorithm specified in the taxonomy. When algorithm selection is determined, the algorithm is mapped to the planning model. When an algorithm is added to the taxonomy a mapping function is defined by the AI modeller to transform planning model components into algorithm components. This is a one-time effort and once the mapping has been specified the algorithm can be repeatedly applied to new matching problem instances. The mapping logic varies with the algorithm type. Section 5.6 describes the mapping logic used in the case studies.

The structure of control units varies between planning and optimisation problems. Control unit structure is described in Section 5.5. The planning-model transformation finishes by instantiating application control units.

5.4.1 Determining System-State Properties

Each system-state element in layer 3 has attributes of complexity, dynamism and observability. These properties are used in algorithm selection. The domain modeller can specify the characteristics of the domain manually or the planning-model transformation algorithm can determine the characteristics automatically.

Evaluating Dynamism

Dynamism is specified as a Boolean value. If it is not specified by the domain expert, the planning-model transformation algorithm sets this attribute on a per system-state basis by testing to see if layer 2 or 3 inputs are required to calculate the value of the system-state. Any system-state element that only relies on layer 1 puts is static. If a system-state relies on sensor or actuator data to determine its value then it is labelled as being dynamic. Otherwise, it would require all inputs to be provided by actuators in which the domain modeller has 100% confidence. Also in pervasive computing environments many sources of sensor data are outside of the control of the system, e.g., weather, noise or traffic-demand. As discussed in Chapter 2, dynamism is a characteristic feature of pervasive computing environments. If the planning model contains dynamic system-states then online algorithms are selected.

Evaluating Observability

Observability is specified as "partial" or "full". If this property is not specified manually

5.5. CONTROL-UNIT OPERATION

by the domain expert, the planning-model transformation algorithm can measure this property on a per system-state basis.

A fully observable system-state is one in which all inputs required to calculate the system-state value are observable with 100% confidence. Layer 1 inputs are by definition fully observable. However, layer 2 sensor data inputs are highly unlikely to be fully observable unless the domain modeller believes that the sensors providing the information are completely accurate. If a system-state relies on sensor or actuator data to determine its value then it is labelled as being partially observable.

Each system-state is labelled as having either "full" or "partial" observability as it is written into the planning model. If the planning model contains partially-observable system-states then noise-tolerant algorithms are selected.

Evaluating Complexity

Chapter 2 described complexity metrics and stated that static measures of the complexity of a problem are severely limited. Therefore the planning-model transformation does not attempt to measure the presence of complexity and it is the responsibility of the domain modeller to specify the presence of complexity.

Complexity is specified as a Boolean value and if this property is not specified by the domain modeller to be true, it is assumed false.

5.5 Control-Unit Operation

Control units are generated by the planning-model transformation algorithm and used to control the behaviour of a pervasive computing application. Control units are executable programs that combine state inference and planning or optimisation algorithms with planning model components.

For planning problems a control unit produces plans to govern the actions taken by an application. Given access to a pervasive computing middleware, the control unit can issue queries access the current state of the system and will search for a policy mapping states to actions.

For optimisation problems a control unit generates candidate solutions which maximise or minimise properties of an application as specified. Again using a pervasive computing middleware, an optimisation control unit will search for a configuration of states that minimises a loss function, or maximises a reward function, based on the policy specified by the domain modeller.

5.5.1 Control Units for Planning Algorithms

The conceptual model of the operating cycle of a planning control unit is shown in Algorithm 8.

5.5. CONTROL-UNIT OPERATION

Input: PM: a planning model; Alg: an instance of a planning algorithm.

```
1 foreach State  $\in$  PM do
2   | integrate sensor evidence into PM ;
3   | calculate  $P'(S)$ ;
4 end
5 foreach action  $A \in$  PM do
6   | calculate  $\text{Alg}(A, S')$ , the reward for taking action  $A$ ;
7 end
8 return the best action from  $A$ ;
```

Algorithm 8: Planning problem control unit.

In lines 1-3, the state inference function provided by each state-variable object in the planning model is invoked to update the application state values. In lines 5-6, the control unit uses the policy specified by the domain modeller, and transformed into a reward model, to calculate the utility of invoking each available action given the updated state information. The action selection logic is implemented by the planning algorithm embedded within the control unit. For single-decision planning problems, the control unit returns the action that maximises the reward at each time step. For sequential planning problems the control unit selects an action that maximises the reward over a search horizon.

5.5.2 Control Units for Optimisation Algorithms

The conceptual model of the operating cycle of an optimisation control unit is shown in Algorithm 9.

Input: PM: a planning model; Alg: an instance of an optimisation algorithm.

```
1 generate candidate set(s)  $\{S(\theta) \in \Theta\}$ ;
2 while not finished do
3   | foreach  $\theta \in S(\theta)$  do
4     | integrate sensor evidence into  $\theta$  ;
5     | calculate  $P'(\theta)$ ;
6   | end
7   | foreach  $\theta \in S(\theta)$  do
8     | evaluate the loss function  $L(\theta)$  ;
9   | end
10  | generate new candidate set(s)  $\{S(\theta) \in \Theta\}$ ;
11 end
12 return the best solution from  $S(\theta)$ ;
```

Algorithm 9: Optimisation problem control unit.

In line 1, a candidate solution θ , from the domain of possible solutions Θ , is initially generated. This solution is generated subject to the system-state specifications. Heuristic optimisation algorithms generate initial candidate solutions stochastically. Lines 2-4, invoke the state inference function provided by state-variable objects to obtain values for candidate

5.5. CONTROL-UNIT OPERATION

solutions. In line 7, the control units use $L(\theta)$, a loss function generated from the policy specified by the domain modeller to evaluate the candidate. The logic governing candidate generation and evaluation is specific to the optimisation algorithm contained within the control unit. The stopping criterion tested in line 2 and the generation of new candidate solutions in line 10 are also specific to the optimisation algorithm contained within the control unit.

5.5.3 Control Units and Middleware Services

Control units require middleware services to execute in pervasive computing environments. To facilitate testing of the control units, an evaluation platform has been built to allow the control units to be tested in a simulation environment. The evaluation platform provides a lightweight pervasive computing middleware and is described in Chapter 6. Control-unit evaluation, using the simulated middleware service, proceeds as shown in Algorithm 10.

5.5. CONTROL-UNIT OPERATION

Input: PM: a planning model, Alg: an algorithm instance

```
1 if evaluate then
2   if problem == planning then
3     foreach state-variable object do
4       | Deploy state-variable object;
5     end
6     Open actuator elements from planning model;
7     foreach actuator do
8       | foreach action in actuator do
9         | Get action scope from spatial attribute ;
10        | Find state-variable objects in scope;
11        | Invoke state inference function of state-variable objects;
12        | Calculate action utility;
13      | end
14      | return highest utility;
15    end
16  end
17  else if problem == optimisation then
18    while not finished do
19      | Generate solution candidate using the state-generator object ;
20      | Deploy candidate;
21      | Invoke inference function from state-variable objects in candidate
22      | solution;
23      | Evaluate candidate;
24      | Perturb candidate;
25    end
26 end
```

Algorithm 10: Control-unit evaluation.

Planning control-unit evaluation. (Lines 2-16)

As described in Section 5.1 state-variable objects provide runtime services to the algorithms. In line 4, each state-variable object in the planning model is deployed by registering it with a middleware.

Lines 6-14 show middleware services required to execute planning control units. In line 9, the spatial attribute of an actuator action is used to lookup state-variable objects in the deployment region of the environment. At each decision step, the utility of each action supported by an actuator element is calculated by combining the reward model entries with the updated application state values provided by state-variable objects. The encoding of the reward model for use in a control unit is algorithm specific. Line 11 engages the low-level competitive sensor fusion and high-level state inference function of

a state-variable object to return updated application state values.

Optimisation Control-unit evaluation. (Lines 17-25)

Candidate solutions to optimisation problems are generated using state-generator objects that allow heuristic optimisation algorithms to explore random regions of a pervasive-computing application state space. As the optimisation control unit executes, candidate solutions are generated and evaluated until algorithm specific termination criteria are met.

In line 20, a candidate solution is deployed by registering its constituent state-variable objects with a middleware. In line 21, the state inference function is invoked from each state-variable object in the candidate solution. The operation of state-variable objects is the same for both planning and optimisation problems. Candidate solutions are evaluated using the values returned by their constituent state-variable objects and the loss function generated from the reward model in the planning-model.

Sections 5.1 - 5.4 have described the planning-model components, model transformations, and algorithm selection and engineering. The next section describes how the transformations are applied to the case studies introduced in Chapter 4.

5.6 Case Studies

Two case studies were presented in Chapter 4 to demonstrate the programming model supported by the methodology. The operation of the transformation algorithms on the case-study domain models and policies specified by the domain modeller is demonstrated to show how planning models and control units are generated. The performance of resulting control units and the impact of the methodology on developing the case studies is evaluated in Chapter 6.

5.6.1 Traffic Junction Controller

The Junction Controller scenario involves developing a pervasive computing application to manage the behaviour of traffic-light controllers in Dublin (See Section 4.8).

5.6.1.1 Domain-Model Transformation

This section describes the operation of the domain-model transformation algorithm on the domain model and policy. The major steps of the domain-model transformation algorithm are as follows:

1. Parse the policy file.

The policy file, from Listing 4.10, is validated and the policy scope and system-state names are read. The scope is specified as "global" and the problem and subtype variables indicate that the scenario is to be modelled as a single-decision planning problem.

5.6. CASE STUDIES

The policy references two system-states: `EmergencyVehiclePresent` and `TrafficDemand`. The domain-model transformation will generate planning model components measuring the values of these system-states throughout the geographical region referenced by the domain model.

2. Parse each system-state definition.

The transformation iterates over the system-state elements referenced in the policy file. The system-state definitions are validated and a template object is created for each system-state. Both the system-states have scope of type "element" and value "phase". This specifies that an `EmergencyVehiclePresent` and `TrafficDemand` state-variable object is to be created for every phase element in layer 1 of the domain model. The system-states are dynamic, partially-observable and not complex. These properties are recorded in the planning model.

3. Planning Problem 1.

As described in Section 4.8.1, the domain model contains 609 phase elements at layer 1, representing the UTC system in Dublin. The domain-model transformation queries the number of phase elements and 609 `EmergencyVehiclePresent` and `TrafficDemand` state-variable objects are created and added to the planning model. Each state-variable object is assigned a unique spatial reference matching the geometry of the traffic phase to which it refers.

The layer 1 element data relevant to each state-variable object is read from the domain model and stored in each object. The layer 2 and 3 input names are stored in each state-variable object.

The instantiated state space for this scenario consists of 1218 state-variable objects stored in the planing model.

4. Planning Problem 2.

The junction controller actuator elements from layer 2 are parsed and validated. The set of actions provided by each actuator correspond to the phases available at each junction. A state-transition component containing the set of actuators and actions are added to the planning model.

A reward model component is generated and added to the planning model. The reward model encodes the policy specified by the domain modeller as utilities for state-action combinations over the range of possible values of the system-states. The policy file in Listing 4.10 specifies a reward of 30 for the `switch-phase` action when the value of an emergency vehicle state-variable is true. It also shows a reward of 10 for the `switch-phase` action when the value of traffic demand is high, and 5 when it is medium, and 1 when it is low.

The domain-specific knowledge read for this problem is a set of prior probabilities for the two system-states. If no prior probabilities are specified then a uniform set

are used.

The planning model is serialised and written to disk. It is now available to be used in the planning-model transformation algorithm.

5.6.1.2 Planning-Model Transformation

The planning-model transformation is applied to the planning model to select an appropriate planning or optimisation algorithm and to instantiate control units to manage the application in accordance with the policy specified by the domain modeller. The major steps of the planning-model transformation algorithm are as follows:

1. Determine the problem type and characteristics.
Table 5.2 shows the domain characteristics and problem class extracted from the system-states defined for this scenario. The planning-model transformation algorithm records the scenario to be a planning problem under partial observability and dynamism.
2. Algorithm Selection.
No algorithm type is specified by the domain modeller and hence one is selected automatically by the planning-model transformation. From the algorithm taxonomy of Table 5.1, if a one-shot or single decision problem is specified then the planning model transformation will select a planning algorithm based on the principle of maximum expected utility (MEU) and implemented using the Bayesian decision network algorithm presented in Chapter 2.

Junction Controller	Partly Observable	Dynamism	Complexity
Traffic Demand	✓	✓	X
Emergency Vehicle	✓	✓	X
Junction Controller	Planning	Optimisation	Inference
Traffic Demand	✓	X	✓
Emergency Vehicle	✓	X	✓

Table 5.2: Junction Controller scenario domain properties and problem types.

3. Algorithm Engineering.
The planning-model transformation imports and executes an initialisation function `initialise_MEU()` and a mapping function called `generate_MEU()`. The mapping function maps the algorithm instance to the planning model components. These functions are provided by the AI modeller when the MEU algorithm is added to the planning and optimisation algorithm library.

The MEU implementation in the algorithm library uses the Hugin Bayesian decision network library to implement single-decision MEU control units.

The planning-model transformation invokes the MEU initialisation function to create a template Bayesian decision network for junction controllers. The planning-model transformation then invokes the mapping function which maps instances of the template decision network to the planning model components.

The `initialise_MEU()` function performs the following actions:

- create the decision network structure.

Decision networks contain three node types: chance nodes (ovals) to represent random variables or states of the system; decision nodes (rectangles) to represent the decision being made at each time step and utility nodes (diamonds) to represent the utility function or reward model.

The `initialise_MEU()` function generates the structure of the decision network using the following logic. A chance node is created for each system-state template object. Chance nodes can be discrete or continuous depending on the system-state data type. A decision node is created for actions supported by the actuator functionality, and a utility node is created and seeded with data contained in the reward model contained in the planning model.

The structure of the decision networks generated by the planning-model transformation for the junction controller control units is shown in Figure 5.1.

- set the prior probabilities of chance nodes in the decision network.

The `initialise_MEU()` function searches the domain specific knowledge in the planning model for prior probabilities for system-states related to chance nodes. If none is found, then the decision network is seeded with uniform prior probabilities.

- configure the utility node with reward model data

The `initialise_MEU()` function encodes the reward model data in the planning model using the discrete node configuration used in the Hugin decision network library.

The utility node is configured as a multi-dimensional array in row-major format with each dimension corresponding to a chance or decision node. If the list of discrete nodes is $(N_1, \dots, N_n)$ and if each node N_k has s_k states. A configuration of the states of these nodes is a list $(i_1, \dots, i_n)$ with $(0 \leq i_k \leq s_k)(1 \leq k \leq n)$. The set of configurations is mapped into the index set $0, \dots, S - 1$ where $S = \prod_{k=1}^n s_k$. S is also the size of the table. A specific configuration of nodes, $(i_1, \dots, i_n)$ is mapped to the index value $\sum_{k=1}^n a_k i_k$ of the utility node where $a_k = s_{k+1} \dots s_n$.

The utility node configuration in the template Bayesian Decision Network is shown in Table 5.3 and provides a reward value for all possible values of the chance nodes.

5.6. CASE STUDIES

Reward	Switch	TrafficDemand	EmergencyVehiclePresent	index
10	Switch=Change	TrafficDemand=heavy	EmergencyVehiclePresent=false	0
40	Switch=Change	TrafficDemand=heavy	EmergencyVehiclePresent=true	1
5	Switch=Change	TrafficDemand=medium	EmergencyVehiclePresent=false	2
35	Switch=Change	TrafficDemand=medium	EmergencyVehiclePresent=true	3
1	Switch=Change	TrafficDemand=light	EmergencyVehiclePresent=false	4
31	Switch=Change	TrafficDemand=light	EmergencyVehiclePresent=true	5

Table 5.3: Junction Controller decision network reward model.

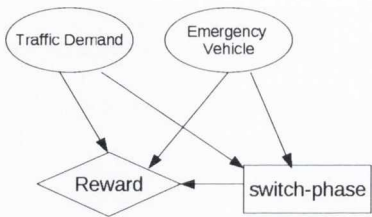


Figure 5.1: A Bayesian decision network generated to run a traffic junction controller.

Control Unit Functionality

The planning-model transformation instantiates the template Bayesian decision network at each junction controller actuator. A total of 247 Bayesian decision networks are generated for this scenario and wrapped in control units. An excerpt from the execution cycle of a control unit is shown in Listing 5.5.

As the control units execute they calculate the posterior value of each chance node given the available run-time evidence derived from sensor and actuator data. The evidence is generated by state-variable objects and returned to the control unit as likelihood functions, where it is mapped to the corresponding chance nodes in each of the 247 decision networks. Once the posterior probabilities are calculated for chance nodes, the utility of each action is calculated and the action with the highest utility is returned. This cycle continues until the control units are halted.

The logic used to map chance nodes to state-variable objects is shown in Algorithm 11.

```
1
2 def run_MEU(plan_model, meu_network):
3     .....
4
5     for actuator in plan_model.actuators:
6         for action in actuator.action_list:
7             add_evidence(action, meu_network)
8             result = get_action_utility(action, meu_network)
9             .....
10    controller.halt_middleware()
```

Listing 5.5: Execution cycle of MEU planning control unit.

Input: A: action; DN: a Decision Network.

```
1 foreach Chance node  $\in$  DN do
2   | find associated state-variable;
3   | get the likelihood function over the state-variable values;
4   | enter into chance node;
5 end
6 update the posterior probabilities across the network;
```

Algorithm 11: Adding evidence to a decision network.

This logic is implemented in an `add_evidence()` function shown in Listing 5.6 and forms part of the mapping function provided by the AI modeller when the MEU algorithm is added to the library.

At each decision cycle, the control unit executes the `add_evidence()` function, which takes as arguments the action being evaluated and the decision network structure.

In lines 15-23 of Listing 5.6, a middleware query is issued to return a state-variable object in the action scope that matches the name of the decision network node.

In line 25 the `get_value()` method on the state-variable object is invoked and engages the competitive sensor fusion and feature/decision-level state inference functions of state-variable objects to obtain a likelihood function over the possible state-variable values. Lines 30-38 show how discrete likelihood data is entered into a chance node for each of its possible state values, and lines 4-45 show continuous likelihood data entered into a chance node.

Likelihood Data and Uncertain Evidence

The likelihood data returned by state-variable objects is uncertain evidence. The semantics of uncertain evidence and belief updating/posterior probability calculation are unclear. Techniques for representing and using uncertain evidence for belief updating are discussed in Section 2.3.3.2.

Ultimately the meaning and treatment of uncertain evidence is at the discretion of the AI modeller who defines the mapping from the planning-model components to a planning or optimisation algorithm. The design of the state-variable objects allows for uncertain evidence to be represented as soft evidence in the form of posterior probability distributions or as virtual evidence in the form of a likelihood function.

Table 5.4 shows an example of belief updating for a `TrafficDemand` chance node in a decision network using virtual evidence returned from a state-variable object. The prior probabilities for the `TrafficDemand` state as specified in layer 4 of the domain model are shown in column 1. In Section 5.1.1 the competitive fusion example showed the calculation of a likelihood function for a `TrafficDemand` state-variable. This likelihood data is shown in column 2. The posterior beliefs resulting from propagating these findings are shown in column 3.

The updated belief for the finding that the value of the `TrafficDemand` state is "light" is reduced to 0.907. If the AI modeller wishes the updated belief to mirror the values

5.6. CASE STUDIES

```

1
2 def add_evidence(action, meu_net):
3     node_name = ""
4     locator = Pyro.naming.NameServerLocator()
5     ns = locator.getNS()
6     geometry = action.geometry
7
8     for node in meu_net.getNodes():
9         if node.getCategory() == H.CATEGORY.DECISION:
10             pass
11         elif node.getCategory() == H.CATEGORY.UTILITY:
12             pass
13         else:
14             node_name = node.getAttribute("HR_Desc")
15             lookup_name = node_name + "_" + geometry.replace("_", "")
16             try:
17                 URI=ns.resolve(lookup_name)
18             except NamingError, x:
19                 print 'Couldn\'t find object, nameserver says: ', x
20                 raise SystemExit
21
22             # create a proxy for the Pyro object, and return that
23             state_variable = Pyro.core.getProxyForURI(URI)
24
25             val = state_variable.get_value()
26             #now add the evidence to the network node
27             if type(val) == types.NoneType:
28                 print "No sensor data found"
29
30             elif node.getKind() == H.KIND.DISCRETE:
31                 fused_likelihoods = val
32                 discrete_node = node.castToDiscreteNode()
33                 num_states = discrete_node.getNumberOfStates()
34                 for i in range(num_states):
35                     for (x,y) in fused_likelihoods:
36                         if x == discrete_node.getStateLabel(i):
37                             #update node state matching data likelihood function
38                             discrete_node.enterFinding(i, y)
39
40             elif node.getKind() == H.KIND.CONTINUOUS:
41                 #match found between continuous sensor data and continuous
42                 #Bayesian network node
43                 continuous_fused_likelihoods = val
44                 (mean, sig) = continuous_fused_likelihood
45                 ccnode = node.castToContinuousChance()
46                 ccnode.enterValue(mean)
47
48     meu_net.propagate()
49     return
```

Listing 5.6: Mapping likelihood data to chance nodes in a decision network.

Traffic Demand	Prior	Likelihood	Posterior
Heavy	0.1	0.004	0.004
Medium	0.8	0.012	0.089
Light	0.1	0.984	0.907

Table 5.4: Junction Controller scenario - belief updating with virtual evidence.

returned by the fusion functions they can amend the `add_evidence()` function to calculate appropriate likelihood ratios using the calculation presented in Section 2.3.3.2.

5.6.2 CCTV Camera Selection

The CCTV camera selection scenario involves developing a pervasive computing application to optimise the selection of CCTV cameras in Dublin city. The application requirements are described in Chapter 4.

5.6.2.1 Domain-Model Transformation

This section describes the operation of the domain-model transformation algorithm on the domain model and policy. The major steps of the domain-model transformation algorithm are as follows:

1. Parse the policy file.
- The policy file is validated and the system scope and system-state names are read. In Listing 4.17 the scope is specified as "global" and the problem and subtype variables indicate that the scenario is to be modelled as an optimisation problem.
- The policy contains two system-state entries: `DegreeOfInterest` and `MaximalDistance`. The domain-model transformation will generate planning-model components measuring the values of these system-states throughout the geographic region referenced by the domain model.

Scenario B	Partly Observable	Dynamism	Complexity
Degree of Interest	✓	✓	X
Maximal Distance	X	X	✓
Scenario B	Planning	Optimisation	Inference
Degree of Interest	X	✓	✓
Maximal Distance	X	✓	X

Table 5.5: CCTV Selection scenario domain properties and problem types.

2. Parse each system-state definition.
- The transformation iterates over the system-state specifications referenced in the policy file. The system-state definitions are validated and a template object is created for each system-state. Both the system-states have scope of type "multi-element", of value "junction", and number "30". This specifies that state-variable objects will be generated to provide values for the system-states at regions of the environment corresponding to sets of 30 junctions. The system-states are dynamic, partially-observable and not complex. These properties are recorded in the planning model. The `DegreeOfInterest` system-state has a layer 3 input named "JunctionInterest" of scope "element" and of value "junction". The domain-model transformation matches the layer

5.6. CASE STUDIES

3 input to the `JunctionInterest` system-state specification and creates a matching template object.

3. Optimisation problem.

The template objects created for each system-state are written to the planning model. System-state templates are used by state-generator objects to produce state-variable objects when required during the execution of the optimisation algorithms.

A reward model component is generated from the policy and added to the planning model. It specifies that the `MaximalDistance` and `DegreeOfInterest` system-states are to be maximised and that each are of equal weighting. No domain-specific knowledge is specified for this scenario. This will result in default values being used for all optimisation algorithm parameters in the control units.

5.6.2.2 Planning Model Transformation

The planning-model transformation is applied to the planning model to select an appropriate optimisation algorithm and instantiate control units to manage the behaviour of the application in accordance with the policy specified by the domain modeller. The major steps of the planning-model transformation algorithm are as follows:

1. Determine the problem type and characteristics.

Table 5.5 shows the domain characteristics and problem class extracted from the system-states and policy defined for this scenario. The transformation algorithm records the scenario to be an optimisation problem under partial observability, dynamism, and complexity.

The `DegreeOfInterest` system-state is partially observable and dynamic. The partial-observability is established from querying the set of inputs and discovering that there are a number of layer 2 sensor elements used to compute the value of the system-state. The `DegreeOfInterest` of a given set of camera installations is dynamic and alters with variation in traffic flows and weather conditions.

The `MaximalDistance` is a function of junction geometry only and hence is fully observable and static. The complexity property is set to true indicating that maximising the value of this system-state is a difficult combinatorial problem.

2. Algorithm Selection.

No algorithm is specified in the domain-specific knowledge and hence one is selected automatically by the transformation. Table 5.1 shows a Local Search algorithm and a Simulated Annealing (SA) algorithm are available for selection in optimisation problems. The complexity of the problem gives an indication of the appropriate choice of algorithm. The motivation for this selection is that if a problem suffers from combinatorial complexity then a heuristic exploratory approach is motivated rather

than a simpler search algorithm. The problem requires multi-objective optimisation and the SA algorithm has been designed to scale to multiple dimensions.

3. Algorithm Engineering.

The SA implementation in the algorithm library is based on the SMOSA algorithm shown in Algorithm 5, Chapter 2. SMOSA uses a probabilistic selection function to generate candidate selections of CCTV installations. The fitness of the candidate is then evaluated to maximise the values of the `MaximalDistance` and `DegreeOfInterest` system-state values and the candidate solution is perturbed to produce a new candidate solution for evaluation.

Acceptance probabilities for candidates are controlled by a temperature parameter and adapted throughout run-time so that the application can avoid local minima and move to states of lower energy. Evaluation cycles are repeated for a maximum number of iterations and the best solution is returned. An example of algorithm engineering provided in Section 5.3.1.1 describes how an instance of the SMOSA algorithm is created.

No parameter values have been specified in the domain-specific knowledge for the temperature parameters and default values are used in the control unit.

The two system-states are continuous and their associated state-variable objects return a mean and confidence value. The SMOSA algorithm is not noise tolerant and does not incorporate an observation or sensor model. It requires scalar values for the system-state values at each evaluation step and uses only the mean values returned by state-variable objects when evaluating and archiving solutions.

Control Unit Functionality

The SMOSA algorithm has a centralised design, i.e., one instance of the algorithm evaluates all candidates created for the set of CCTV installations in the road network. Due to the centralised algorithm design, the planning-model transformation generates a single control unit for the application.

The control unit generates solutions that are optimal in the sense that no other solutions in the search space are superior to them when all objectives are considered. The SMOSA algorithm archives all Pareto-optimal solutions that it finds in each execution cycle.

5.7 Conclusion

This chapter has described the logic of the domain-model and planning-model transformations. It has also described how planning and optimisation algorithms are added to the methodology and the support provided for automated algorithm selection.

To facilitate the evaluation of the methodology, prototypes of the domain and planning-model transformations have been developed. The prototype transformations are written in Python and implement the designs presented in Chapters 4 and 5. The domain and

5.7. CONCLUSION

planning-model transformations use the LXML libraries (Behnel 2009) to support XML validation. This is used in validating domain models and policy files specified in XML. LXML also supports XPATH which is used for parsing system-state definitions and building and searching the algorithm taxonomy.

The evaluation chapter assesses the impact of the methodology on developing the two case study applications and presents a detailed analysis of the performance of the planning and optimisation algorithms used in the case studies.

— Twenty-two acknowledged concubines, and a library of sixty-two thousand volumes; attested the variety of his inclinations; and from the productions which he left behind him, it appears that both the one and the other were designed for use rather than ostentation.

Edward Gibbon's appraisal of the Emperor Gordian (The History of the Decline and Fall of the Roman Empire, Volume 1, 1776)

6

Evaluation

This thesis describes a methodology for engineering planning and optimisation algorithms for use in pervasive computing applications. This chapter presents an empirical evaluation of the methodology through examining its use in scenarios that are representative of the target class of applications.

The methodology spans the domains of model-driven engineering (MDE) and automated planning. Accordingly there are two axes to the evaluation:

- The first axis evaluates the methodology as a MDE software engineering methodology and measures the impact of the MDE methodology on the development process.
- The second axis evaluates the performance of the planning and optimisation algorithms generated by the automated-planning component.

Chapter 1 identified a set of requirements for engineering planning and optimisation algorithms for pervasive computing applications.

Section 6.1 chapter reviews these requirements and the associated set of design decisions that underpin the methodology. Section 6.2 presents a set of evaluation criteria obtained from the literature relating to both MDE and automated planning. An evaluation strategy is derived from mapping these standard evaluation criteria to observable metrics for the development effort of using the methodology and the performance of the control units generated by the automated planning component. The metrics are used as the basis for an empirical analysis of the methodology based on applying it to the Junction Controller and CCTV Camera Selection scenarios that have been presented in Chapters 4 and 5.

Section 6.4 presents the evaluation platform that has been built to support testing the functionality of the control units. In sections 6.5-6.8, the two case studies are analysed according to the evaluation criteria. Finally section 6.9 summarises the findings of the evaluation chapter.

6.1 Requirements and Design Review

The methodology has been designed to support developers in the task of developing pervasive computing applications that use planning and optimisation algorithms. This is a multi-stage task that involves specifying application logic, sensor and actuator infrastructure, and the design and implementation of planning and optimisation algorithms.

Decomposing this multi-stage task leads to the following requirements:

- R1 - Allow application logic (policy) to be specified at a high-level.
A developer should be able to specify the application state space and desired behaviour without reference to planning and optimisation techniques and without reference to properties of the execution environment.
- R2 - Provide abstractions masking runtime conditions.
One of the difficulties when developing the target class of applications is that runtime conditions in pervasive computing environments are often unknown and unknowable at design time. The following information may be unknown at design time:
 - The availability and quality of sensor and actuator infrastructure. An application may be deployed over variable numbers of possibly mobile sensors with variable quality data.
 - The state of the environment. The state of the environment will often not be directly observable and may be uncertain.

In so far as possible it is desirable that both a domain expert and an AI developer should be able to use the methodology without explicit reference to runtime conditions relating to sensor/actuator infrastructure or uncertain state data.

- R3 - Simplify the development of planning and optimisation algorithms in pervasive computing environments.
A developer should be able to provide implementations of planning and optimisation algorithms without reference to properties of the deployment environment.
- R4 - Provide a development model that is extensible, that promotes reuse, and is suitable for use in large-scale applications.

The development model should allow applications to be engineered using a range of planning and optimisation algorithms. It should allow applications to be easily

6.1. REQUIREMENTS AND DESIGN REVIEW

re-engineered and extended and should also scale to accommodate large data sets associated with large-scale applications.

It should also facilitate domain knowledge and algorithm code reuse. If a developer writes a planning or optimisation algorithm it should be available for reuse across multiple applications. Likewise domain knowledge should also be reusable across multiple applications where appropriate.

- R5 - Rapidly test algorithm efficacy.

At design time the following information may not be known:

- The most appropriate algorithm to use. It may not be possible to theoretically establish the most appropriate algorithm to use in advance. Empirical testing may be necessary to determine which algorithm is most suitable.
- The most appropriate values for algorithm parameters. Likewise it may not be possible to theoretically establish useful values of parameters for algorithms.

Developers should be able to use the methodology to rapidly evaluate the performance of a range of algorithms with a single domain model.

The parameters of these algorithms should be easily configurable as the optimal values will often be domain appropriate.

- R6 - Provide a consistent representation of uncertain data.

Uncertainty in state information is a characteristic of pervasive computing environments. The degree of uncertainty in application state data should be quantified. Handling the uncertainty in state estimates is algorithm specific. This leads to a requirement to provide an algorithm-independent, consistent format to handle uncertain data.

- R7 - Automatic selection of appropriate planning and optimisation algorithms.

Automated selection of appropriate algorithms is necessary to facilitate complete automation of the process that enables a domain modeller with no knowledge of AI to produce a running application incorporating planning and optimisation algorithms.

6.1.1 Design Decisions Review

The requirements R1-R7 motivate the design of the methodology. The methodology has been designed to reduce the development effort associated with the target class of applications and to systematise the use of planning and optimisation techniques in pervasive computing applications.

The methodology is built around a set of tools provided to domain and AI modellers. The tool-chain automates the development process where deemed possible and where it is not possible to automate, clearly defined points are provided to integrate logic. An example

6.1. REQUIREMENTS AND DESIGN REVIEW

of this is the provision of automated competitive fusion and the manual specification of decision-level fusion.

These requirements are addressed by the following design decisions:

- D1 - Domain Model and Policy Specification.

The domain modeller uses the methodology through the domain model. A set of schemas are provided to a domain modeller and used to define the application state space in XML. A schema is also provided to specify a policy to control the behaviour and deployment region of applications. A reward model is generated from the policy and used to direct the behaviour of the planning and optimisation algorithms. By editing the policy file the domain modeller can focus on application logic exclusively. If the algorithm taxonomy can match an appropriate instance of a planning or optimisation algorithm then the application code can be generated automatically once the policy and domain model are specified. A policy is specified with a scope that indicates the physical region of the environment over which an application is to be deployed. The scope is a spatial value that can be specified manually or can be specified relative to spatial values associated with domain model elements. The scope can be used to control and alter the deployment region of applications.

- D2 - Topological Abstraction.

A topological abstraction is used to allow a region of the environment to be programmed independently of the run-time conditions in that region. Using the policy specification, the domain modeller specifies desired application behaviour in the deployment environment. Components generated by the transformation engine, use spatial queries to resolve entities such as sensors, actuators, and application state values to regions of the environment at run-time to implement application policy. The topological abstraction underlies the programming model used in the methodology and mandates that model elements have spatial attributes.

Spatial queries are used to map sensor data quality to estimates of the certainty of application state. The requirement to provide estimates of the changing certainty of application state data at run-time led to a design choice to represent state information as variables. Each instance of state data is represented as a state-variable object and is also tagged with a spatial attribute indicating the region of the environment to which it relates. State-variable objects provide an interface for the developer of planning and optimisation algorithms to program against and return an estimate of the value of an application state at a point in time and the uncertainty associated with this estimate.

- D3 - Planning Model.

Planning and optimisation algorithms are programmed to a planning model interface that abstracts away from the environment. Planning-model components provide an API to algorithm developers providing access to estimates of application state and

associated uncertainty, reward models and available actions.

High-level support for developing optimisation algorithms is provided in the form of state-generator objects used to generate and evaluate candidate solutions to heuristic optimisation algorithms.

- D4 - Automated Code Generation.

The methodology provides tools for automatic code generation. Instances of planning and optimisation algorithms can be generated and applied to planning model data. The domain model is designed to be extensible and scalable. The topological abstraction allows model elements to be dynamically added and combined thereby allowing large data sets to be created and handled, providing scalability and extensibility. For example, a policy file may specify that a set of states measuring traffic-flow, are to be optimised at all traffic junctions in a region. If new junctions are added to the domain model then the tools for automated code generation can be used to extend the application.

The planning and optimisation algorithms are programmed to an interface provided by the planning model. The planning model provides an algorithm-independent representation of the domain model data. Hence, algorithms can be reused across multiple applications.

- D5 - Domain Specific Knowledge.

The domain model incorporates a domain-specific knowledge layer. This layer can be used to specify which planning and optimisation algorithm to use and to specify values for the free parameters associated with each algorithm. Once the algorithm name and parameter values are specified, the automated code generation tools can be used to create an instance of the application. The performance of these instances can be evaluated and tested.

The possibility of executing models in a simulation environment allows for early experimentation of the system under development to analyse high-risk aspects or alternative solutions.

- D6 - Likelihood Functions.

State-variable objects return likelihood functions that provide an estimate of the value of an application state and that quantify the uncertainty associated with the estimate. Likelihood functions can be used to represent both discrete and continuous data. They can also be combined with any available prior probabilities for state estimates. This combination of prior probabilities and likelihood functions is compatible with a Bayesian inference framework.

- D7 - Algorithm Taxonomy.

A taxonomy mapping problem types and environment properties to algorithms is provided. Domain or AI modellers can manually specify the planning or optimisation

algorithm to be used in layer 4 of the domain model. If no algorithm type is specified, the algorithm taxonomy is used by the planning model transformation to select and instantiate a planning or optimisation algorithm. The mapping between problem type and algorithm choice is based on a survey of the literature and empirical observations of algorithm performance. The taxonomy is specified in XML and the selection logic can be easily amended.

6.2 Evaluation Criteria

This section establishes a rationale for the evaluation method. The design decisions D1-D7 combine to provide an MDE methodology for automated planning and optimisation. The methodology is embedded in two principal components:

1. A domain model to model the target applications.
2. A transformation engine that parses the domain model and policy specification and generates a control unit appropriate to the application being developed. The transformation engine supports the selection of one of a set of algorithms for use in the application being developed.

In order to devise an evaluation strategy we present some standard evaluation criteria for MDE and for automated planning systems. We list criteria as E1-E8. These criteria are used in conjunction with the requirements and design decisions to provide an evaluation strategy for the methodology.

6.2.1 Evaluating MDE Systems

Fernandes et al. (2008) discuss the use of MDE for pervasive computing. They argue that by placing the focus on the development of models, the MDE approach enables better software systems development. They identify the following ways in which MDE improves the efficacy of systems development:

- E1 - Productivity gains.
Atkinson & Kuhne (2003) state that productivity improvement from development efforts is the underlying motivation for MDE. They consider short-term and long-term productivity gains. Short-term gains are obtained through the amount of functionality a software artefact can deliver. Long-term gains are obtained by increasing the longevity of the software artefact. To provide long-term productivity gains, models should be clearly described and stored using standard data formats. User-defined mappings should be provided to shield models from the specifics of the technological platform on which they will be deployed.

6.2. EVALUATION CRITERIA

- E2 - Provision of concepts close to the domain and the reduction of the semantic gap. The development of software through models rather than through programming languages enables non-computing specialists with domain expertise to produce systems (Selic 2003).

The use of models also enables the use of concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain.

- E3 - Automation and less sensitivity to technical change. The automated transformation of high-level models to code supports an accelerated development cycle and leads to a reduction of costs (Mellor et al. 2003). In MDE the developer can focus on the application logic by developing PIM models. The PSMs are generated automatically by transformation tools. Initially the PIM to PSM transformation will need to be designed, which is a difficult and specialized task. But this transformation only needs to be done once and can be applied in the development of many systems (Anneke et al. 2003, 9).

Models are easier to maintain and are also less sensitive to the chosen implementation technology thereby increasing the lifetime of the model (Mellor et al. 2003).

- E4 - Capture of expert knowledge and reuse. Expert knowledge can be encoded in mapping functions that provide the transformation of one model to another. This expert knowledge is available for reuse when application logic changes or when its implementation technology changes (Mellor et al. 2003).

6.2.2 Evaluating Automated Planning Systems

Nau & Ghallab (2004) propose evaluation criteria (E5-E7) for automated planning systems. These criteria are applied to automated planners to determine the range of and size of problems they can solve.

- E5 - The scope or range of the planning algorithm. All planning algorithms are developed with some restrictive assumptions in mind. The authors classify a set of assumptions (A1-A7), that restrict the range of problem types that can be solved using the algorithm.
 - A0 - Finite.
The system has a finite set of states.
 - A1 - Fully Observable.
The system is fully observable, i.e., one has complete knowledge about the state of the system at all times.
 - A2 - Deterministic.
The system is deterministic, i.e., for every state s and action a , $|\gamma(s, a)| \leq 1$. If

6.2. EVALUATION CRITERIA

an action is applicable to a state, its application brings a deterministic system to a single other state.

- A3 - Static.

The system is static, i.e., the set of events E is empty. The system has no internal dynamics and stays in the same state until the controller applies some action.

- A4 - Attainment Goals.

The only kind of goal is an attainment goal, which is specified as an explicit goal state s_g or a set of goal states S_g . This assumption excludes states to be avoided, constraints on state trajectories and utility functions.

- A5 - Sequential Plans.

A solution plan to a planning problem is a linearly ordered finite sequence of actions.

- A6 - Implicit Time.

Actions and events have no duration, they are instantaneous state transitions. This assumption is embedded in state-transition systems, a model that does not represent time explicitly.

- A7 - Off-line Planning.

The planner is not concerned with any change that may occur in the system while it is planning. It plans for the given initial and goal states regardless of the current dynamics.

This classification is applied to automated planning systems to evaluate the range of problems they can solve.

- E6 - The control knowledge and tuning required for each planning domain.

When evaluating the performance of an automated planner, any additional control knowledge that will need to be given to the planning system in order for it to address practical problems, should be specified. This includes, for example, whether the planner needs such knowledge, how precise and specific to a problem the knowledge needs to be and how easily this knowledge can be acquired and formalized.

- E7 - The size of the problem that can be solved in a reasonable amount of time for each problem in the scope of the algorithm.

6.2.3 Methodology Evaluation Criteria

The standard MDE and automated planning evaluation criteria described above are combined to provide set of methodology evaluation criteria.

The MDE evaluation criteria (E1-E4): [Productivity, Abstraction, Automation, Reuse], can be applied directly to our methodology. These criteria evaluate the impact of the methodology on the development effort associated with developing planning and optimisation applications for pervasive computing applications.

6.3. EVALUATION STRATEGY

The automated planning evaluation criteria (E5-E7): [Range, Tuning, Size] can also be applied to the methodology tool-chain. The methodology supports multiple planning and optimisation algorithms and can be applied to a wide range of problems. The set of supported algorithms can also be extended to support additional problems. Therefore the range and size criteria must be evaluated for each algorithm supported.

We introduce a final evaluation criterion for the automated planning support. The methodology provides automatic selection of planning and optimisation algorithms appropriate to the problem modelled. E8, Algorithm Selection, is a criterion introduced for measuring the effectiveness of algorithm selection.

The next section will present metrics used to measure the evaluation criteria (E1-E8). These metrics are based on observations and measurements obtained from using the methodology and tool chain to develop the case study applications.

6.3 Evaluation Strategy

The evaluation strategy is to establish if the criteria E1-E8 are supported in the design decisions D1-D7, (note there is no correspondence between the numbers.) Table 6.1 lists the design decision and evaluation criteria and Table 6.2 shows a mapping that relates design decisions to evaluation criteria. Some design decisions map to multiple evaluation criteria. This matrix shows that, with the exception of E7, all evaluation criteria are addressed in the methodology design.

Design Decisions		Evaluation Criteria	
D1	Domain Model and Policy	E1	Productivity Gains
D2	Topology Abstraction	E2	Abstractions
D3	Planning Model	E3	Automation
D4	Code Generation	E4	Reuse
D5	Domain-Specific Knowledge	E5	Problem Range
D6	Likelihood Functions	E6	Control Tuning
D7	Algorithm Taxonomy	E7	Problem Size
		E8	Algorithm Selection

Table 6.1: Design designs and evaluation criteria.

	E1	E2	E3	E4	E5	E6	E7	E8
D1		✓			✓	✓		
D2	✓				✓			
D3					✓	✓		
D4	✓		✓	✓				
D5				✓		✓		
D6					✓			
D7								✓

Table 6.2: Mapping from design decisions to evaluation criteria.

Where possible an empirical approach is adopted to determine how well the criteria are addressed by design decisions. Two case studies have been presented in Chapters 4 and 5. These case studies demonstrate the use of the methodology; how applications are modelled; and how the transformations operate on the models to generate control units.

The MDE evaluation criteria E1-E4 are examined using metrics extracted from observing how the methodology impacts on the development effort associated with the case studies. The automated planning evaluation criteria E5-E8 are examined by observing the performance of the planning and optimisation algorithms embedded in the generated control units. The algorithm selection criterion is evaluated quantitatively by comparing the performance of a range of algorithms over the case-study domain models.

6.3.1 Development Effort of using Methodology

Software engineering approaches are applied in application development to achieve the goal of generating high-quality software (Tian & Zelkowitz 1995). The control units that provide the application functionality are automatically generated by the transformations. The control units embed complex planning and optimisation algorithms whose quality or utility is verified through executing the control units in a simulation environment. The provision of the domain modelling language and transformations are intended to control and reduce the complexity of developing the target class of applications.

Factors that are frequently identified as contributing to complexity are application code size, abstraction and modularity (Banker et al. 1989), (Kemerer 1995), (Khoshgoftaar & Munson 1995), (Davis & LeBlanc 1988). Complex systems can be engineered using a "divide and conquer" approach to manage complexity (Nasr 2002). (Mathiassen & Stage 1990) states that it is generally accepted that complexity in software design can be managed through abstraction and decomposition or modularisation. Empirical studies establishing the usefulness of these two approaches in reducing application complexity include (Mathiassen & Stage 1990) and more recent findings by (Sooriamurthi 2009) and (France & Rumpe 2007).

6.3.1.1 Modularity

The division of a system into reasonably sized, encapsulated modules is a universal means of reducing complexity in any system (Banker et al. 1989). Modular encapsulation reduces complexity by making sure that changes to the internal operation of a module are contained within that module (Tegarden et al. 1995) (Tarr et al. 1999).

The design of the methodology promotes modularity. It provides for the distinct development roles of domain modeller and AI modeller. The domain model design promotes a decomposition of the application state space into distinct system-state specifications. The planning and optimisation algorithms are programmed to an application-independent planning model interface.

Munnelly (2010) surveys quantitative techniques for measuring modularity including coupling metrics for measuring intra-module dependencies. At the present time, Python software measurement tools do not provide support for measuring modularity through coupling in Python source code (Misra & Cafer 2010).

6.3.1.2 Abstraction

Abstraction is widely used as a technique to simplify the development process (van Deursen et al. 2000) and is achieved by providing the developer with concise, expressive constructs that shield low-level functionality. Abstraction provides clarity to the selection and structuring of relevant information for the developer and is an effective approach to combating complexity (Mathiassen & Stage 1990).

Through abstraction, higher levels of programming can be provided to application developers. However the absence of techniques for directly measuring abstraction has been identified as a hard and important open problem (Mernik et al. 2005).

There are some indirect measures of abstraction. The provision of high-level abstractions reduces application size (Mernik et al. 2005), therefore reducing complexity. An examination of major complexity metrics reveals that most of them link the complexity of a program with its size (Banker et al. 1989) (Fenton & Neil 2000).

In an effort to provide some quantitative metrics for the development effort associated with using the methodology, we focus on metrics that indirectly measure the abstraction and complexity of the code provided by the domain modeller and AI expert when using the methodology for the case studies.

6.3.1.3 Metrics for development effort

Four metrics are identified and used to measure the development effort associated with using the methodology: There is some overlap in the evaluation criteria and some metrics are relevant to multiple criteria. The intention is to provide an indication of the effort required to use the methodology and the degree of automation provided by the methodology tool chain.

1. Lines of code (LOC) (E2 - Abstraction)

The LOC metric counts the number of lines of class code. Although crude, LOC is still the most commonly used metric to represent software size (Vickers 2003). We measure the LOC from the application developer's point of view, i.e., how many lines of code the developer implements, not the amount of code generated. We show how many lines of code the domain modeller and the AI modeller have to provide in each of the scenarios. The LOC measure includes both XML documents and Python code providing state inference functionality. The measure currently includes all XML element tags and data. This could be amended to measure only the number of XML entities containing domain model and policy data. Such a change would result in

6.3. EVALUATION STRATEGY

a smaller LOC value, however it would not impact on the Degree of Automation metric described below. This metric measures ratios of LOC values and will provide similar values irrespective of counting XML lines or entities providing data.

The LOC metric is related to evaluation criterion E2 and provides an indication of the effort required to use the methodology in the case studies. This shows the extent to which the domain model abstractions allow the domain modeller to specify in a high-level language the application functionality.

2. Complexity of code (CC) (E2 - Abstraction)

A cyclomatic complexity metric is used to measure the complexity of the code written by the domain and the AI modeller. The cyclomatic complexity of a piece of code is the measure of linearly independent paths through a program (McCabe 1976). A CC measure greater than 7 is generally acknowledged to indicate that the code is complex and may be difficult to understand. Although critiqued, it is nonetheless a widely used metric (Gill & Kemerer 1991) (Shepperd 1988). The CC measure of the code provided by the domain modeller and the AI modeller in each of the scenarios is presented.

The CC metric is related to evaluation criterion E2. The expressiveness of the high-level domain model should result in less complex code being written by the domain expert.

3. Spatial query generation

It would be useful to measure both the expressive power of the domain model abstractions and the utility provided by the domain and planning-model transformations. These measurements would provide a way to demonstrate productivity gains from using the methodology. The following metric is proposed as a crude assessment of these two measures.

Planning model components are entirely produced and consumed by the transformation algorithms and are the most tangible artefacts produced by the transformation algorithms. The largest component of a planning model is the collection of state-variable objects, that provide the backbone of the functionality required by the application. They supply the state space values to the planning and optimisation algorithms embedded in control units and perform sensor invocations and sensor fusion as required.

State-variable objects operate using spatial queries generated from domain-model abstractions at design time by the domain-model transformation algorithm.

We propose to measure the size of the spatial query data produced by the domain-model transformation and used by the planning-model transformations. The size of spatial query data is used as a crude estimate of the development effort contributed by the methodology itself.

6.3. EVALUATION STRATEGY

To obtain the size of the spatial query data, the Pickle Python object serialisation mechanism¹ is used to write the instance data from the planning model components to disk. The serialised data is formatted in a printable ASCII format and only contains instance data relating to spatial queries and reward model data.

The ratio of the domain and AI modelling development effort to the methodology development effort is used as a indicator of productivity gains and provides a rough indication of the degree of automation in application development.

4. Degree of Automation (E1 - Productivity, E3 - Automation, E4 - Reuse)

The LOC metric cannot be directly compared to the size of the generated spatial query data to judge the impact of automation on development effort. Instead the following approach is used.

The development effort required to implement each of the case study applications is recorded using the LOC metric for the domain and AI modeller roles. The size of the spatial query set is also recorded. Each case study is then extended by adding new requirements and the LOC metric is measured for the extended domain and AI models. The increased spatial query data produced by the domain-model transformation algorithm is recorded and used as a proxy measure for the increased functionality in the extended scenario.

The ratio of increased domain and AI modelling development effort can be compared to the ratio of the increase of spatial query data generated. These ratios indicate the expressive power of the domain model abstractions and the contribution of the transformation algorithms in effecting application functionality.

We refer to this ratio as the, "degree of automation" and it is measured as:

$$\delta(DM + AI) / \delta(Planning_Model_Size)$$

where

$$\delta(DM + AI) = \frac{\text{Extended domain and AI model (LOC)}}{\text{Original domain and AI model (LOC)}}$$

and

$$\delta(Planning_Model_Size) = \frac{\text{Extended Spatial Query Size}}{\text{Original Spatial Query Size}}$$

The degree of automation metric is used to measure the related evaluation criteria of productivity gain (E1), automation (E3), and reuse (E4). The degree of automation provides an indirect measure of productivity gains and automation. The extension of the scenarios and the re-running of the transformations shows the reuse of domain model components and AI algorithms. The scenario extensions also shows that the expert knowledge encoded in mappings between planning-model components and algorithms is preserved in the transformation algorithms.

¹<http://docs.python.org/library/pickle.html>

6.3.2 Metrics for automated planning and optimisation

The evaluation of the automated planning component of the methodology is based on empirical analysis of the performance of the generated control units. In contrast to traditional automated planning systems, the methodology supports multiple planning and optimisation algorithms. The standard automated planning metrics (E5 - Range, E6 - Control Tuning, E7 - Problem Size) presented in Section 6.2.2 do not generalise to the set of all algorithms in the algorithm library and can only be applied to individual planning and optimisation algorithms.

These metrics are therefore applied to the planning and optimisation algorithms that have been implemented and used in the case studies.

6.3.2.1 Metrics for Control Unit Performance

The following metrics are used to evaluate the automated planning component of the methodology.

1. Problem range. (E5 - Problem Range)

For each case study, the assumptions that underpin the domain model and algorithm used are presented. The application of the methodology to two case studies shows that the methodology design can be extended to support a wide range of problems.

However, the domain modelling language is used across all problems. There are some design decisions that restrict the range of problems that can be modelled using the domain model. For each scenario, the restrictive assumptions underpinning the design of the domain model are presented.

2. Control tuning. (E4 - Reuse, E6 - Control Tuning)

The impact of control tuning is evaluated by showing how algorithm performance is impacted by tuning parameter values. Expert knowledge relating to control tuning can be captured in layer 4 of the domain model. Parameter tuning is also discussed in the context of reusing algorithms over different domain models.

3. Problem size. (E7 - Problem Size)

The impact of algorithm design on the problem size that can be solved is discussed for each of the case studies presented.

4. Algorithm Selection. (E4 - Reuse, E7 - Algorithm Selection)

We add this evaluation criterion because the methodology supports multiple algorithms and provides automated algorithm selection where no algorithm is specified for use. The performance of the automatically selected algorithm is compared to the performance of another appropriate baseline algorithm to evaluate the usefulness of the selection criteria used in the algorithm taxonomy. The application of multiple algorithms to a common domain model is also used to demonstrate model and algorithm reuse supported by the methodology.

6.4 Evaluation Platform

An evaluation platform was built to validate the design and allow the performance of the control units to be evaluated. The evaluation platform provides a testbed for demonstrating the functionality of the control units. The implementation of the evaluation platform facilitates the empirical evaluation of the case studies presented in this chapter.

6.4.1 Evaluation Platform Services

The evaluation framework provides the following services:

- Instantiation and invocation of state-variable objects.
State-variable objects can be deployed, registered with middleware lookup services and invoked by planning and optimisation algorithms and other state-variable objects. The invocation services use state-variable lookup operations using spatial queries derived from topological abstractions.
- Sensor and actuator data simulation.
Sensor and actuator data is required to evaluate the performance of the inference models and the planning and optimisation algorithms. The evaluation platform allows sensors and actuators to be registered with lookup services and sensor data to be simulated from discrete and continuous probability distributions.
- Sensor and actuator discovery and access.
The evaluation platform provides lightweight middleware services that support sensor identification and selection. Sensor selection is the task of mapping the requirement for an abstract data type to a sensor type that provides that type of data. Sensor identification is the task of mapping a sensor type to a sensor identifier.

Lookup and discovery services are built on spatial queries that operate using the topology abstractions in the domain model. These services are invoked at runtime by queries generated by the domain and planning model transformations.
- State inference services.
The sensor fusion and state inference algorithms make extensive use of Bayesian inference routines. A customised Bayesian inference library is provided by the evaluation platform.

6.4.2 Evaluation Platform Implementation

The evaluation platform is written in Python. Python supports both procedural and object oriented programming styles. This allows system-state inference functions to be specified and also object-oriented design to be used in the algorithm library and planning model implementations. The standard implementation of Python is written in ANSI C and it

6.4. EVALUATION PLATFORM

compiles and runs on virtually any platform. It is automatically compiled to portable byte-code which runs on any platform with Python installed (Lutz 2007, 14).

The evaluation platform contains the following major components:

- **Middleware services.**

The middleware service is built around the Pyro Distributed Object (Pyro) system (de Jong 2009). Pyro is written in Python and provides a lightweight, portable distributed object system for Python objects using TCP/IP. It provides a Naming Service (NS) to record the location of objects. The location of the NS can be discovered by a broadcast mechanism. This allows the state-variable objects to discover the NS and lookup other state-variable objects.

All standard Python types and user defined classes can be distributed using Pyro. The only restriction is that the types used must be supported by the Pickle object serialisation module.

Pyro supports dynamic proxies. Once a handle to a remote object is obtained, direct remote access to attributes is supported using normal Python syntax. The NS can dynamically create and remove new Pyro objects and is multi-threaded to handle multiple requests simultaneously. The ability to dynamically create and remove objects is used by state-generator objects when heuristic optimisation algorithms explore a problem solution space by generating random sets of state-variable objects.

- **Spatial Query Support.**

The topological abstractions use the spatial attributes of domain-model elements to simplify the design and implementation of planning and optimisation algorithms. Spatial operations are provided as a middleware service and used in a wide variety of roles: sensor selection, state-variable object lookup and access and policy scoping. Support for spatial operations is provided by the PostgreSQL engine.

PostgreSQL is an enterprise-level, open source object-relational database system. It runs on all major operating systems, including Linux, UNIX and Windows and is fully ACID compliant. It has full support for foreign keys, joins, views, triggers, and stored procedures. It includes most SQL:2008 data types and has native programming interfaces for many languages including C/C++ and Python.

PostGIS adds support for geographic objects to the PostgreSQL object-relational database. PostGIS allows the PostgreSQL server to be used as a backend spatial database for geographic information systems (GIS). PostGIS implements the OpenGIS "Simple Features Specification for SQL" (SFS) and has been certified as compliant with the "Types and Functions" profile (Obe & Hsu 2010).

SFS provides a well-defined and common way for applications to store and access feature data in relational or object-relational databases, so that the data can be used to support other applications through a common feature model, data store and in-

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

formation access interface (OpenGeoSpatialConsortium 1999). SFS defines geospatial features described using vector data elements such as points, lines and polygons.

The SFS standard is provided by the Open Geospatial Consortium Inc (OGC). The OGC is a non-profit, international, voluntary consensus standards organization that leads the development of standards for geospatial and location based services.

- **Sensor and Actuator Simulator.**

A lightweight simulator is provided to generate sensor and actuator data for use in the evaluation scenarios. The simulator can generate data according to a specified discrete or continuous probability distribution. It does not provide the ability to model or simulate complex phenomena.

- **Inference Engine.**

The methodology provides support for competitive and feature/decision level state inference techniques. The libraries used in competitive fusion were developed as part of the thesis tool chain. However the feature/decision level fusion tasks used in the case studies require a comprehensive Bayesian inference library. No comprehensive Python Bayesian inference library was found to be available for use.

The Hugin Inference Engine (Andersen et al. 1990) is used to provide the required support. Hugin exposes an API in C/C++ and in Java (Jensen 2007). However the methodology tool chain is implemented in Python so a python interface to the Hugin inference engine was implemented.

The interface was constructed using the SWIG software development tool. SWIG is an open-source software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python (Beazley 1996). SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software (Beazley & Lomdahl 1997).

Directives are added to the program's header files and the SWIG tool creates source code that provides the glue between C/C++ and the target language. A Hugin python module was created and can be linked as an extension module. By exposing the Hugin engine as a python module we provide access to a C/C++ library which has no equivalent in the python language.

6.5 Junction Controller Scenario Evaluation

This section evaluates the use of the methodology in developing the Junction Controller scenario. The scenario is examined from both the development effort and the automated planning perspectives. The development effort is quantified using the metrics from Section 6.3.1.3. In Section 6.6 the requirements for the Junction Controller scenario are extended and the development effort re-gauged. The evaluation aims to show that, by using the

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

methodology, the development effort for the first application is reduced and that the development effort for extending the application is much reduced.

The performance of the control units generated for the scenario is evaluated using the metrics from Section 6.3.2.1. The evaluation profiles the range of planning problems supported by the methodology and the correctness of algorithm selection.

6.5.1 Development Effort

In this section we present metrics to gauge the development effort required to use the methodology to develop the Junction Controller application. The development effort metrics are used as a baseline to judge the extent to which the methodology provides productivity gains.

- LOC. Fig 6.1 shows the lines of code required to develop the scenario using the methodology. The domain model is 147 lines long. This measure includes the layer 3 system states definition in XML, the inference functions written in python, and the policy files. Layer 1 and layer 2 data add to the size of the domain model but are excluded from the count. This is because the information from layers 1 and 2 are not related to the application but rather characterise the environment in which the application is deployed.

The MEU algorithm used in the scenario is 420 lines of python code. This code is provided by the AI modeller and is the programmed to the Planning Model API.

- CC. Table 6.3 shows the complexity of the code provided by the domain expert and the AI modeller. Only components with a CC measure of 4 and higher are shown. The domain modeller is required to provide two components, a function to calculate the values of the TrafficDemand and EmergencyVehiclePresent system-states. These functions have a CC value of 4 indicating that they are easy to code. The table also shows four functions from the MEU algorithm implemented by the AI modeller. Three of the functions have a CC value greater than 7 indicating that more effort is required to develop the algorithm that to specify the domain model. However the MEU algorithm need only be implemented once and is available for us in any future planning application.

Component Name	Type	Name	CC
TrafficDemand	Function	calculate	4
EmergencyVehiclePresent	Function	calculate	4
MEU Algorithm	Function	initialise_MEU	14
MEU Algorithm	Function	add_evidence	13
MEU Algorithm	Function	generate_utility_table	7
MEU Algorithm	Function	get_action_utility	4

Table 6.3: Junction Controller scenario cyclomatic complexity.

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

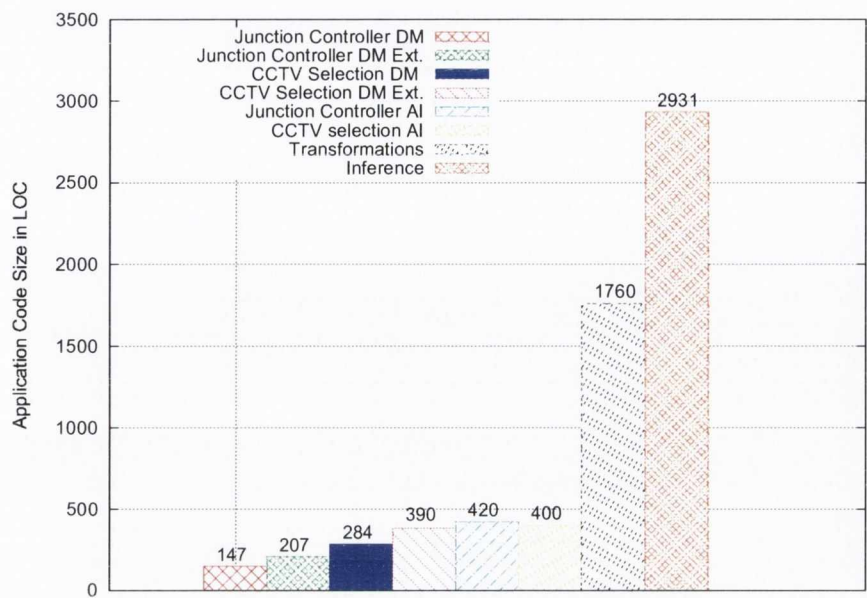


Figure 6.1: LOC effort to implement the Junction Controller and CCTV Selection case studies.

- Spatial Query Generation. From Table 6.4 the size of the planning model instance data generated by the domain-model transformation algorithm is 1432KB in size. The largest component of this is the spatial query instance data from 1218 state-variable objects, that are generated to provide values for the application state space.

The combined size of the domain model and AI algorithm is 567 LOC for the Junction Controller scenario. The CC metric indicates that the most complex code is required for the MEU algorithm which is reusable for future planning problems. The domain-model transformations use the topology abstractions in the domain model to generate 1432KB of spatial query data embedded in 1218 state-variable objects. The domain and AI modeller do not have to provide any code to access sensor and actuator infrastructure and are shielded from interfacing directly with pervasive computing environment.

These figures do not allow for direct comparison but do show that a significant portion of the application functionality is provided by the automated code generation. These figures will be used in Section 6.6.1.1 to gauge the impact of automation and reuse on the development effort associated with extending the Junction Controller functionality.

6.5.2 Automated Planning Evaluation

In this section we present standard metrics to evaluate the automated planning component of the methodology used in developing the Junction Controller application. The evaluation establishes the range of problems that can be solved, the control tuning used and the size

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

Case Study	Run Number	Planning Model Size	State Variable Objects
Junction Controller	-	1432 KB	1218
Junction Controller Ext	-	1907 KB	1827
CCTV Selection	50	69 KB	3050
CCTV Selection	100	69 KB	6100
CCTV Selection	500	69 KB	30500
CCTV Selection Ext	50	118 KB	4600
CCTV Selection Ext	100	118 KB	9200
CCTV Selection Ext	500	118 KB	46000

Table 6.4: Automated component generation.

of the problem instances that can be modelled and solved. We also evaluate the algorithm selection criteria used in the algorithm taxonomy.

6.5.2.1 Range

The range of planning problems that can be modelled and solved using the domain model and MEU algorithm is determined by the design assumptions shown in Table 6.5. (Note these are restrictive assumptions and the fewer that apply the more problems can be modelled and tackled). The MEU algorithm is profiled because this is the one selected by the algorithm-taxonomy for single decision problems in partially observable environments.

Assumption	Domain Model	MEU Algorithm	Note
(A1) Finite	X	X	State-variable objects
(A2) Fully Observable	X	X	Likelihood function
(A3) Deterministic	X	X	Stochastic transitions
(A4) Static	✓	✓	
(A5) Attainment Goals	X	X	Utility function
(A6) Sequential Plans			Single-Decision algorithm
(A7) Implicit Time	✓	✓	
(A8) Off-line Planning		X	Online algorithm

Table 6.5: MEU algorithm and domain model planning assumptions.

The use of state-variable objects to represent application state space allows application state spaces to be easily extended by instantiating more state-variable objects. The domain model does not assume a finite state space and allows applications using discrete and continuous data to be modelled. The use of likelihood functions allows partially observable environments to be modelled. The SCXML standard used to represent actions allows stochastic transitions to be specified and issues such as actuator failure planned for. The domain model does not model events not associated with actuator invocations and hence assumes the environment is static. Dynamism in the environment is addressed through the use of online algorithms rather than modelled in the domain model. The reward model is specified using utility functions rather than through attainment goals. This is a flexible

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

method for setting goals but does not allow explicit goal states or constraints on application behaviour to be modelled. The MEU algorithm is a single decision algorithm and cannot generate sequential plans. Both the domain model and the MEU implementation assume implicit time. Although dynamism in the environment cannot be explicitly modelled the algorithm can run in an online mode to respond to environment changes and replan after every action.

The methodology supports multiple planning and optimisation algorithms and is designed to be easily extensible. As mentioned in Section 6.2.2, the restrictive assumptions A1-A7, are traditionally used to specify the range of problem types that can be solved. For our methodology, it is more useful to think of these assumptions as selection criteria for determining appropriate planning and optimisation algorithms for a problem. However the domain model is common to all problems and the assumptions A4 and A7, of static environments and of implicit time do limit the range of problems that can be modelled.

6.5.2.2 Control Tuning

The operation of the MEU algorithm is presented in Chapter 2. Control tuning is restricted to setting the reward model through the application policy as the algorithm itself has no free parameters to be set.

6.5.2.3 Problem Size

There are no algorithm design decisions that limit the size of the problem that can be tackled with the MEU algorithm. This is not surprising as the complexity of single decision planning is substantially lower than that of sequential planning problems. The use of a state-variable representation in the domain model also allows for applications of arbitrary state-space size to be modelled.

6.5.2.4 Algorithm Selection

Algorithm	Sensor Invocations v.1	Sensor Invocations v.2
MEU	4559	5499
Round Robin	0	0
Random Action Selection	0	0

Table 6.6: Junction Controller scenario sensor invocation.

The algorithm taxonomy in Table 5.1 shows that the MEU algorithm is automatically selected for single-decision planning problems. To evaluate this selection we compare the performance of the MEU algorithm versus the performance of Round Robin and Random Action selection algorithms.

The performance is measured by the reward generated by the control units given the policy specified by the domain modeller. The control units were generated with three

6.5. JUNCTION CONTROLLER SCENARIO EVALUATION

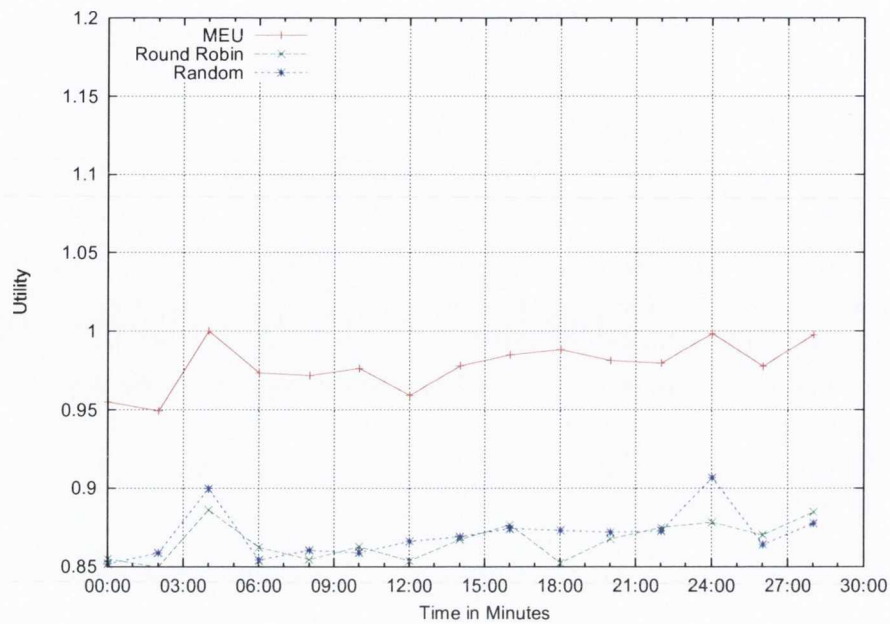


Figure 6.2: Junction Controller scenario control unit performance.

algorithms. The MEU algorithm chooses at each time the phase with the highest utility. A Round Robin action selection strategy cycles through the available phases at each junction running all phases consecutively. Finally, a random action selection strategy chooses phases at random from each junction controller.

Fig 6.2 shows the average reward across all 247 junctions for each of the three action selection strategies. The data is displayed as a time series over 30 minutes. Each phase runs for 2 minutes. The Y axis shows the average reward across all junctions for three action selection strategies over 15 phase decisions.

The reward values shown on the Y axis are normalised so that all algorithms produce readings below 1. The normalised values for each algorithm are calculated as follows:

$$\begin{aligned} & \sum \text{Avg-MEU}_{\text{Reward}} / \sum \text{MEU}_{\text{MaxReward}} \\ & \sum \text{Avg-Random}_{\text{Reward}} / \sum \text{MEU}_{\text{MaxReward}} \\ & \sum \text{Avg-RoundRobin}_{\text{Reward}} / \sum \text{MEU}_{\text{MaxReward}} \end{aligned}$$

The $\sum \text{MEU}_{\text{MaxReward}}$ normalisation factor is used across the result sets of the three algorithms. This allows the results to be directly compared. As expected the MEU algorithm consistently outperforms the other two, returning rewards which on average are 10 to 15 % higher than the round robin and random action selection strategies.

However there is a cost for using this algorithm. The overhead in terms of sensor invocation is shown in Table 6.6 For each phase selection operation occurring every 2 minutes the MEU algorithm makes 4559 sensor invocations. The 30 minute cycle shown in Fig 6.2 required a total of 68,385 sensor invocations. The Round Robin and Random Selection algorithms do not have an associated sensor invocation cost.

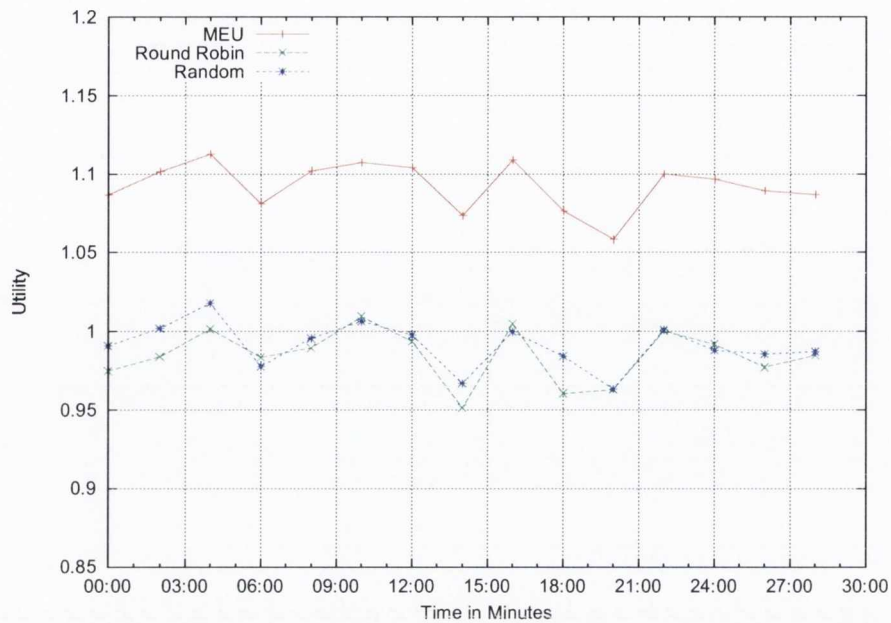


Figure 6.3: Extended Junction Controller scenario control unit performance.

6.6 Junction Controller Scenario Extension

The functionality of the scenario was extended to add support for detecting and responding to the presence of pedestrians at junctions. To accommodate this extended functionality a new system-state `PedestrianPresent` is defined and added to layer 3 and the policy specification is extended to specify a reward for switching to a phase for which pedestrians are waiting. The definition for the `PedestrianPresent` system-state is similar to that shown in Listing 4.8. Using the `properties` element, the domain modeller specifies this state as being dynamic and partially observable. The scope of this system-state is defined as being of type `element` and of value `phase`. The first transformation algorithm will generate a state-variable object for this system-state at every instance of a phase in the domain model. The policy file is extended as shown in Listing 6.1.

To re-engineer the control units to account for the extended functionality the domain modeller re-runs the transformation algorithms. When the domain model transformation algorithm is re-run it extracts a new reward model and combines the pedestrian present reward with the rewards for sensing an emergency vehicle and for traffic demand. It also generates state-variable objects for the three system-states at each phase element in the domain model. The reward model, the transition system and the state-variable objects are written to the planning model.

The planning model transformation reads the problem type and environment properties from the planning model and selects the MEU algorithm from the taxonomy. It then creates a new Bayesian decision network template with chance nodes to match the

6.6. JUNCTION CONTROLLER SCENARIO EXTENSION

```
1 <state>
2   <name>PedestrianPresent</name>
3   <reward>
4     <type>discrete</type>
5     <range>>false-true</range>
6     <action>switch-phase</action>
7     <value>0-5</value>
8     <weight>1</weight>
9   </reward>
10 </state>
```

Listing 6.1: Junction Controller scenario extension to policy.

Reward	Switch	TrafficDemand	PedestrianPresent	EmergencyVehiclePresent	index
10	Change	heavy	false	false	0
40	Change	heavy	false	true	1
15	Change	heavy	true	false	2
45	Change	heavy	true	true	3
5	Change	medium	false	false	4
35	Change	medium	false	true	5
10	Change	medium	true	false	6
40	Change	medium	true	true	7
1	Change	light	false	false	8
31	Change	light	false	true	9
6	Change	light	true	false	10
36	Change	light	true	true	11

Table 6.7: Junction Controller scenario extended decision network reward model.

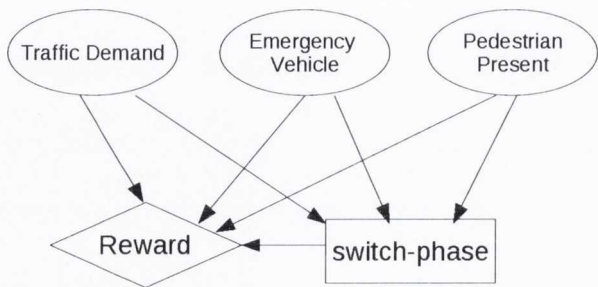


Figure 6.4: The extended Bayesian decision network for junction controllers.

6.6. JUNCTION CONTROLLER SCENARIO EXTENSION

system-states and a reward node to match the new reward model. A total of 247 new decision networks are generated and wrapped in control units. The structure of the new Bayesian decision networks is shown in Fig 6.4. The reward model data is extracted and converted into a form suitable for use in the Bayesian decision network. The new reward model running on the networks is shown in Table. 6.7.

6.6.1 Development Effort for Extension

The addition of the PedestrianPresent system-state increases the domain model by 60 lines: 45 lines of XML and 15 lines of python. Of this increase, 10 lines of XML are for the policy extensions, 35 lines of XML are for the system-state definition and 15 lines of Python code are for the inference function.

The extension increases the size of the domain modelling effort by 40%. However there is no additional AI modelling required as the MEU algorithm mapping logic is unchanged. The domain model increases from 147 LOC to 207 LOC while the AI model, i.e., the MEU algorithm is unchanged. The increase in development effort associated with the extended scenario functionality is: $567/627 \text{ LOC} = 11\%$.

6.6.1.1 Degree of Automation

From Section 6.3.1.3, the increase in spatial query data size in the planning model, is used as a proxy measure for increased application functionality. The degree of automation measures the contribution of the methodology tool chain to application development effort and it is measured as:

$$\delta(\text{DM} + \text{AI}) : \delta(\text{Planning_Model_Size}) = 567/627 : 1432/1904$$

An 11 % increase in development effort results in 33% increase in application functionality. In row 2 of Table 6.4 the planning model generated from the original domain model is shown to be 1432 KB in size and contains 1218 state-variable objects.

In row 3 of Table 6.4 the planning model generated from the extended domain model is shown to be 1907 KB in size and which contains 1827 state-variable objects. This represents an increase of 33% over the size of the original planning model. This increase is accounted for principally by an additional 609 state-variable objects representing the Pedestrian-Present system-state.

Column 2 of table 6.6 shows that the original control units performed 4559 sensor invocations at every decision point. Column 3 of table 6.6 shows that the extended control units performed 5499 sensor invocations at every decision point, an increase of c. 21%.

An 11% increase in development effort results in an increase of the application functionality by 33% and increases the number of sensor invocations by 21%. This result indicates the expressive power of the domain model abstractions.

No additional AI code is required. This shows that the methodology supports reuse

6.7. CCTV SELECTION SCENARIO

and that application logic can be specified and extended using the domain model alone, thereby allowing the domain modeller to focus on application behaviour.

6.6.1.2 Algorithm Selection

Fig 6.3 shows the average reward across all 247 junctions for each of the three action selection strategies. As before, the data is displayed as a time series over 30 minutes. Each phase runs for 2 minutes. The Y axis shows the average reward across all junctions for three action selection strategies over 15 phase decisions.

The rewards shown on the Y axis are normalised using the normalisation constant $\sum \text{MEU}_{\text{MaxReward}}$ from the previous experiment. This allows Figs 6.2 and 6.3 to be visually compared.

Again the MEU algorithm consistently outperforms the other two returning rewards which on average are 10% higher than the round robin and random action selection strategies. The number of sensor invocations required increases due to the addition of an additional sensor type required to detect pedestrian presence. Table 6.6 shows that 5499 sensor invocations are required for every phase selection operation occurring every two minutes. The 30 minute cycle shown in Fig. 6.3 required a total of 164,970 sensor invocations.

6.7 CCTV Selection Scenario

This section evaluates the use of the methodology in developing the CCTV Selection scenario. The scenario is examined from both the development effort and the automated planning perspectives. The development effort is quantified using the metrics from Section 6.3.1.3. In Section 6.8 the requirements for the scenario scenario are extended and the development effort re-gauged. The evaluation aims to show that, as in the Junction Controller scenario, by using the methodology, the development effort for the first application is reduced and that the development effort for extending the application is much reduced.

The performance of the control units generated for the scenario is evaluated using the metrics from Section 6.3.2.1. The evaluation profiles the range of optimisation problems supported by the methodology and the correctness of algorithm selection.

6.7.1 Development Effort

In this section we present metrics to gauge the development effort required to use the methodology to develop the CCTV Selection application. The development effort metrics are used as a baseline to judge the extent to which use of the methodology provides productivity gains.

- LOC. Fig 6.1 shows the lines of code required to develop the scenario using the methodology. The domain model is 284 lines long. This measure includes the layer 3 system-states definition in XML, the inference functions written in Python, and the

6.7. CCTV SELECTION SCENARIO

policy files. The SMOSA algorithm used in the scenario is 400 lines of python code. This code is provided by the AI modeller and is programmed using the planning model API.

- CC. Table 6.8 shows the complexity of the code provided by the domain expert and the AI modeller. Only components with a CC measure higher of 4 and higher are shown. The domain modeller is required to provide three components, a function to calculate the values of the JunctionInterest, MaximalDistance and DegreeOfInterest system-states. The JunctionInterest inference function contains complex logic in the enter_sensor_data() function for mapping sensor data to Bayesian network nodes. The MaximalDistance inference function contains complex logic for calculating the length of a fully connected network of traffic junctions.

The table also shows two functions and four methods from the SMOSA algorithm implemented by the AI modeller. These functions and methods have a CC value greater than 7 indicating that more effort is required to develop the algorithm that to specify the domain model. However the SMOSA algorithm need only be implemented once and is available for us in any future optimisation application.

Component Name	Type	Name	CC
JunctionInterest	Function	enter_sensor_data	13
JunctionInterest	Function	get_beliefs	5
JunctionInterest	Function	calculate	4
MaximalDistance	Function	calculate	7
SMOSA Algorithm	Method	Archive.prune	13
SMOSA Algorithm	Method	Archive.select_solution	11
SMOSA Algorithm	Function	Archive.submit	7
SMOSA Algorithm	Function	run_mosa	7
SMOSA Algorithm	Method	Archive.get_solution_IDs	6
SMOSA Algorithm	Method	Candidate.peturb_candidate	4

Table 6.8: CCTV Camera Selection scenario cyclomatic complexity.

- Spatial Query Generation. From Table 6.4, the size of the planning model instance data generated by the domain-model transformation algorithm is 69KB.

The combined size of the domain model and AI algorithm is 684 LOC for the CCTV Selection scenario. The CC metric indicates that the most complex code is required for the JunctionInterest system-state and the SMOSA algorithm. The SMOSA algorithm implementation is reusable for future planning problems. The domain-model transformations use the topology abstractions in the domain model to generate 69KB of spatial query data embedded in a state-generator component in the planning model.

The CCTV Selection scenario planning model is only 5% of the size of the Junction Controller scenario planning model as shown in Table 6.4. The disparity in size reflects the fact that the composition of planning models for optimisation problems differs from

6.7. CCTV SELECTION SCENARIO

that of planning problems. No state-variable objects are generated at design time for optimisation problems. As discussed in Section 5.1.3, planning models for optimisation problems contain a state-generator object which is used by optimisation algorithms to generate random sets of state-variable objects at run-time to support heuristic exploration of the application state space.

Another factor that explains the difference in size is that the spatial attributes used in the CCTV Selection scenario are geometry points rather than polygons and have a much smaller representation size.

The domain and AI modeller do not have to provide any code to access sensor and actuator infrastructure and are shielded from interfacing directly with pervasive computing environment. However the domain modeller does have to write code to map fused discrete and continuous later 2 input data to Bayesian Network nodes.

These figures do not allow for direct comparison but do show that a significant portion of the application functionality is provided by the automated code generation. These figures will be used in Section 6.8.1 to gauge the impact of automation and reuse on the development effort associated with extending the functionality of the CCTV Selection scenario.

6.7.2 Automated Planning Evaluation

In this section we present standard metrics to evaluate the automated planning component of the methodology used in developing the CCTV Selection scenario. The evaluation establishes the range of the optimisation problems that can be solved, the control tuning used and the size of the problem instances that can be modelled and solved. We also evaluate the algorithm selection criteria used in the algorithm taxonomy.

6.7.2.1 Range

The range of optimisation problems that can be modelled and solved using the domain model and SMOSA algorithm is determined by the design assumptions shown in Table 6.9. The SMOSA algorithm is profiled because it is selected in the algorithm-taxonomy for multi-objective optimisation problems. The design assumptions pertaining to the domain model are the same as presented in Section 6.5.2.1 and are not repeated here.

The use of likelihood functions allows partially observable environments to be modelled however the SMOSA algorithm does not use the uncertainty information and operates on the mean value returned for a continuous state-variable or the most probable value returned for a discrete state variable.

Dynamism in the environment is addressed through the use of online algorithms rather than modelled in the domain model. The SMOSA algorithm can be run in an online mode to respond to changes in the environment.

The SMOSA algorithm is an optimisation algorithm and generates configurations of state-variable objects that maximise or minimise properties of the system. It does not gen-

6.7. CCTV SELECTION SCENARIO

Assumption	Domain Model	SMOSA Algorithm	Note
(A1) Finite	X	X	State-variable objects
(A2) Fully Observable	X	✓	Likelihood functions
(A3) Deterministic			Stochastic functions
(A4) Static	✓	✓	
(A5) Attainment Goals	X	X	Utility function
(A6) Sequential Plans			
(A7) Implicit Time	✓	✓	
(A8) Off-line Planning		X	Online algorithm

Table 6.9: SMOSA planning assumptions.

erate single or sequential decision plans or use state-transition information. The SMOSA implementation assumes implicit time.

6.7.2.2 Control Tuning

The operation of the SMOSA algorithm is presented in Section 2.5.1. This section discusses the impact on the performance of the control unit of tuning the free parameters of the SMOSA algorithm.

There are two free parameters to be tuned when using the SMOSA algorithm: the number of runs and the temperature. The number of runs controls the length of the evaluation cycle used by the algorithm and how many candidate solutions it will evaluate. The temperature parameter controls the exploration rate of the algorithm. The higher the temperature the more the algorithm will explore the landscape of solutions (See Chapter 2). The goal of control tuning is to set the runs parameters at the optimum point to avoid expensive evaluation cycles requiring possibly expensive sensor invocations and to set the temperature parameter at the optimum point to avoid local minima and to scale maxima. In the examples following, the alpha value or decay rate of the temperature was set to 0.95. The decay rate specifies how quickly the temperature cools at each evaluation cycle. The performance of the algorithm is shown in Figs 6.5-6.7.

These figures show the pareto optimal landscapes found by the optimisation algorithm. Pareto-optimality is described in Section 2.5.1. The points shown in Figs 6.5- 6.7 show the DegreeofInterest and the MaximalDistance values of the pareto-optimal candidate solutions. The optimal solutions which maximise both values, appear on the top-right of the graph. All figures are plotted using the same scales so that comparisons between the numbers of runs and temperature are visually meaningful.

Each figure also shows a normalised maximum and an absolute maximum point. The normalised maximum is calculated by weighting equally the sets of DegreeofInterest (DI) and MaximalDistance (MD) pareto-optimal values.

$$\max\{DI_i / \sum_i^n DI + MD_i / \sum_i^n MD\}$$

6.7. CCTV SELECTION SCENARIO

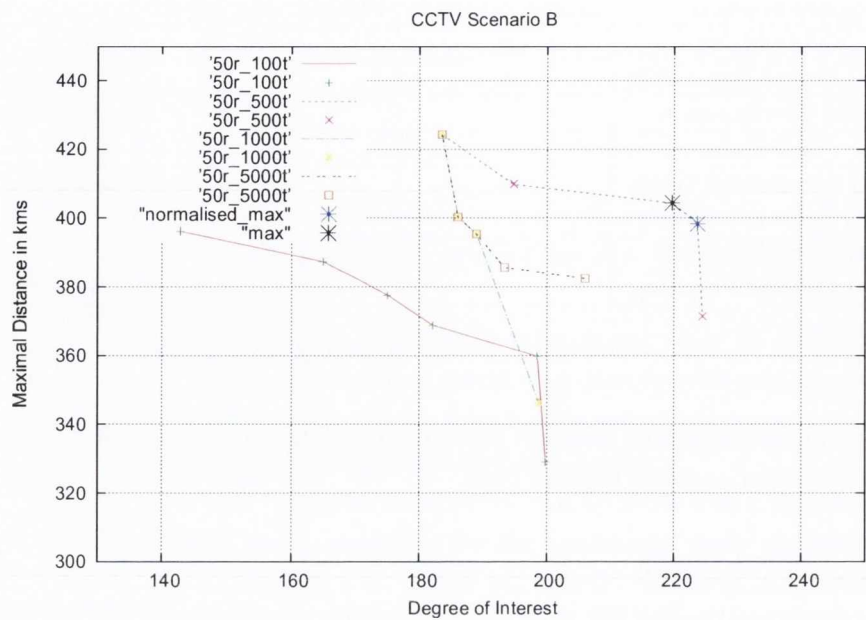


Figure 6.5: CCTV Selection scenario SMOSA performance for 50 runs.

The normalised maximum values prevents one criterion with large absolute values outweighing another criterion with smaller absolute values.

The absolute maximum point is calculated as: $\max\{DI_i + MD_i\}$. This point shows the solution with the maximum absolute values for the combined system-states. The number of sensor invocations required to obtain these values are shown in Table 6.11.

Fig 6.5 shows the pareto landscape mapped by the SMOSA algorithm over 50 evaluation cycles and using a range of starting temperatures: 100, 500, 1000 and 5000. The results show that a starting temperature of 500 clearly outperforms all other temperature values. Both the maximum and normalised maximum points are found when the starting temperature is 500.

Fig 6.6 shows the pareto landscape mapped by the SMOSA algorithm over 100 evaluation cycles and using a range of starting temperatures: 100, 500, 1000 and 5000. The results show the temperatures of 500 and 5000 perform well. The performance of the SMOSA algorithm with a starting temperature of 500 over 100 runs is similar to its performance over 50 runs. However, the algorithm with a starting temperature of 5000 performs much better over 100 runs than over 50 runs. This result may be explained by the fact that high temperatures increase the likelihood of the algorithm moving from a useful solution to a less useful solution to escape local minima. Over the course of 50 runs the algorithm does not have time to converge on good quality solutions. However over the course of 100 runs, the algorithm has time to explore the landscape and to cool sufficiently to converge on useful solutions. However, the temperature of 500 explores more of the landscape as shown by its longer frontier. The shorter frontier associated with the higher temperatures

6.7. CCTV SELECTION SCENARIO

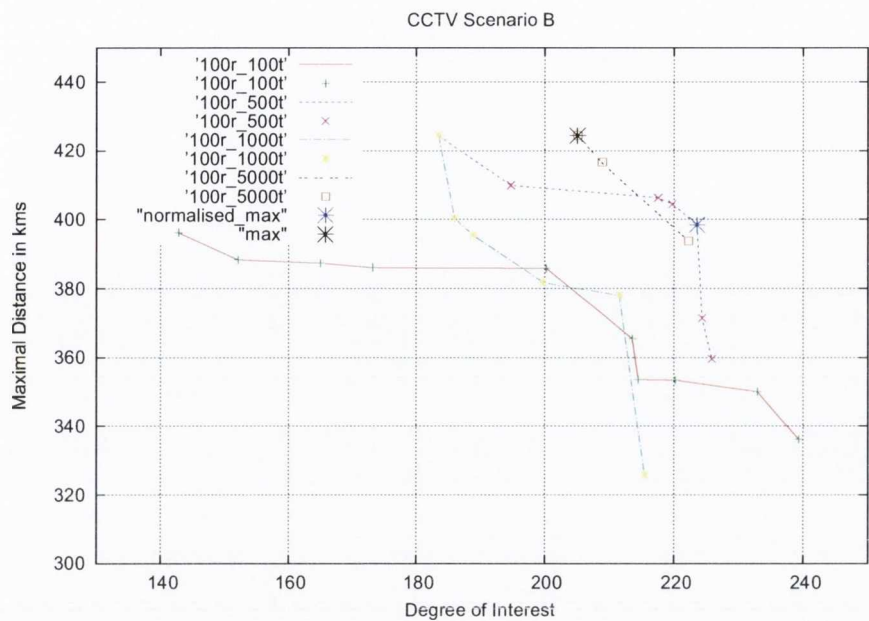


Figure 6.6: CCTV Selection scenario SMOSA performance for 100 runs.

also makes sense as poor quality solutions accepted initially will be pruned by stronger solutions accepted as the temperature cools and the algorithm converges. The normalised maximum is found with a starting temperature of 500 and the maximum is found with a starting temperature of 5000.

Run Count	Temperature	Figure	2D Maximum		Normalised Ranking
50	500	Fig 6.5	223.59	398.34	0.3462
100	500	Fig 6.6	223.59	398.34	0.3462
500	5000	Fig 6.7	221.04	416.91	0.3522
Run Count	Algorithm	Figure	2D Maximum		Normalised Ranking
50	SA	Fig 6.8	191.52	367.75	0.3080
100	SA	Fig 6.9	212.95	359.82	0.3213
500	SA	Fig 6.10	211.01	373.99	0.3259

Table 6.10: CCTV Selection scenario control unit performance.

Fig 6.7 shows the pareto landscape mapped by the SMOSA algorithm over 500 evaluation cycles and using a range of starting temperatures: 100, 500, 1000 and 5000. The results shows that the algorithm performance is similar for starting temperatures of 500 and 5000. The optimal value is obtained using a temperature of 5000, however the starting temperature of 500 again explores more of the landscape. Over the course of 500 runs both the normalised maximum and the maximum points are found with a starting temperature of 5000.

The performance of the algorithm across all runs and temperatures has been analysed.

6.7. CCTV SELECTION SCENARIO

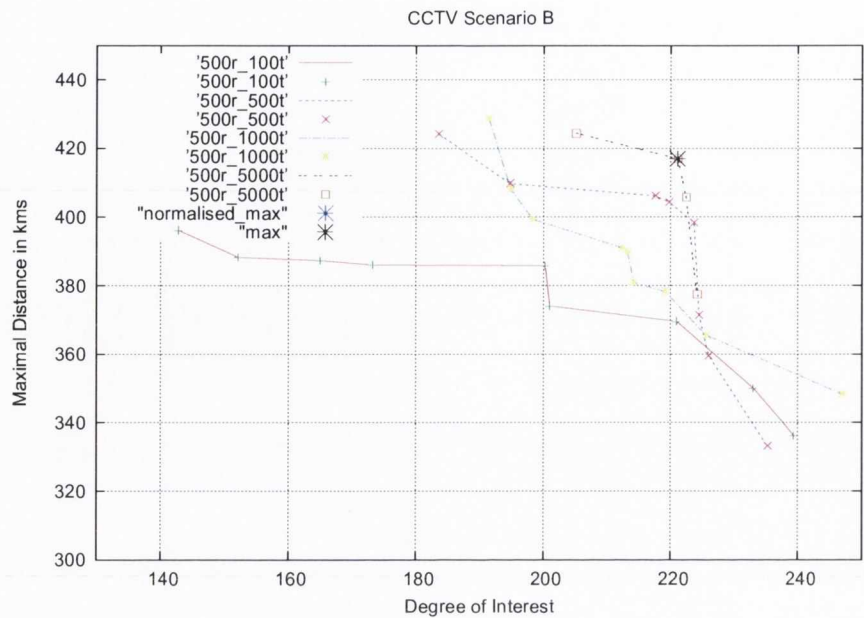


Figure 6.7: CCTV Selection scenario SMOSA performance for 500 runs.

Rows 2-4 of Table 6.10 show the optimal results obtained for this scenario using the SMOSA algorithm. Column 4 shows the normalised maximum values obtained for the optimisation criteria. Column 5 ranks these values to show the optimal value of the run count and temperature parameters across the experiment. A scalar normalised value is calculated to enable a direct comparison between the results. The optimal result for the SMOSA algorithm is obtained over 500 runs using a starting temperature of 5000.

Table 6.11 shows the number of sensor invocations required to produce these results. Varying temperature values result in differing numbers of sensor invocations. Column 3 shows the mean number of sensor invocations for each run count over the range of starting temperatures. The optimal result is obtained using a run count of 500, a starting temperature of 5000, and at a mean cost of 71,712 sensor invocations. However a very close value is obtained over 50 runs at a temperature of 500 using only 7118 sensor invocations. This shows that a carefully tuned algorithm over 50 runs can produce results comparable a poorly tuned algorithm over 500 runs.

The evaluation cost is measured using the number of sensor invocations, which is a function of the simulator setup. However it is clear that increasing the number of runs and evaluation cycles leads to an increase in the number of sensor invocations and hence to an increase of the costs associated with sensor invocations.

6.7.2.3 Problem Size

Theoretically there are no algorithm design decisions that limit the size of the problems that can be tackled with the SMOSA algorithm. Adding optimisation criteria or objectives increase the dimensionality of the problem.

However, based on empirical observations of algorithm performance under increasing dimensionality the following subtle behaviour change is noticed. Due to the pareto-optimal selection criteria, candidate solutions will be archived if even one of the criteria is not dominated by already archived solutions. The logic of the SMOSA algorithm is presented in Alg. 5. Step 5 shows that the Metropolis algorithm used for escaping local maxima is only engaged when a candidate solution is rejected. Increasing problem dimension with the concomitant increase in the number of pareto-optimal criteria, directly increases the probability of archiving solution candidates, thereby reducing the number of invocations of the metropolis algorithm and reducing the chances of escaping from local optima.

This is a subtle but important point and no mention of it is made in the original SMOSA paper (Suppaitnarm et al. 2000) or in a more recent survey of multi-dimensional simulated annealing algorithms (Suman & Kumar 2006). This is an indication of the usefulness of experimentally validating the choice of planning or optimisation algorithm.

6.7.2.4 Algorithm Selection

In the algorithm taxonomy shown in Fig 5.1, the SMOSA algorithm is referenced as the preferred algorithm for online optimisation problems. In this section we compare the performance of the SMOSA algorithm with the performance of a baseline optimisation algorithm. The baseline algorithm chosen is a simple local search (SA) algorithm whose logic was presented in Alg. 3. The SA algorithm is chosen because it is the simplest random search method and has no adjustable parameters that need to be tuned to the problem at hand (Spall 2003, 38). The performance of the SA and SMOSA algorithms in the CCTV Selection scenario is compared to validate the optimisation algorithm selection criteria in the taxonomy.

To compare algorithm performance, the SA algorithm is added to the algorithm library and a mapping is defined from the SA algorithm to the planning model components. The SA algorithm logic is simple and the logic and mapping requires 158 LOC.

The algorithm taxonomy shown in Table 5.1 shows that the SMOSA algorithm is selected for complex problems. If the domain modeller wishes to test the SA algorithm using the CCTV domain model they can do so in two ways:

- Changing the complexity property from true to false in the `MaximalDistance` system-state specification will cause the taxonomy to automatically select the SA algorithm. This specification is shown in Listing 4.15.
- The taxonomy selection can be overridden by specifying the SA algorithm be used in the domain specific knowledge layer.

6.7. CCTV SELECTION SCENARIO

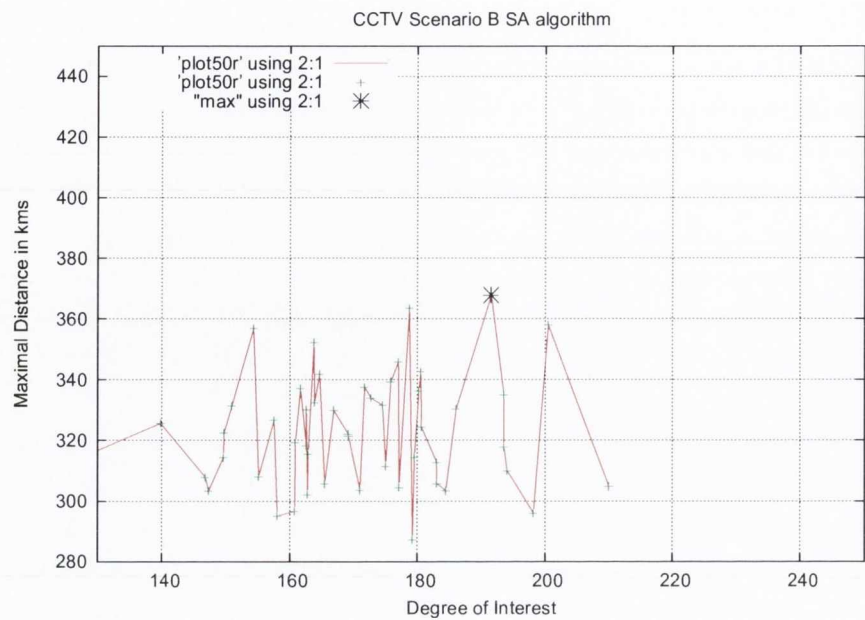


Figure 6.8: CCTV Selection scenario SA performance for 50 runs.

Whichever technique is chosen, only one word in the domain model need be changed and the transformation algorithms will generate a new planning model and control unit with the SA algorithm embedded.

The performance of the SA algorithm using the CCTV Selection domain model is shown in Figs 6.8-6.10. The optimal solutions which maximise both values, appear on the top-right of the graphs. All figures are plotted using the same scales so that visual comparisons between run numbers are visually meaningful.

The SA algorithm only maintains one solution during its operation. The algorithm operates by generating a candidate solution and comparing the value of the new candidate to the existing solution. The utility of each candidate is calculated from summing and comparing the values associated with the optimisation criteria referenced in the policy specification.

Figures 6.8-6.10 show the performance of the algorithm for 50, 100 and 500 runs respectively. The points shown are the candidate solutions generated as the algorithm executes. The solution with the highest value is also shown. The concept of pareto-optimality does not apply to the operation of this algorithm and hence no normalised maximum are calculated. The maximum value is calculated from summing the values of each criteria. If one criterion has large absolute values this will outweigh a number or criteria with smaller absolute values. The number of sensor invocations required to obtain these values is shown in Table 6.11. The sensor invocation overhead associated with the SA algorithm is similar to that associated with the SMOSA algorithm.

The performance of the algorithm across all run counts is analysed in Table 6.10. Rows

6.7. CCTV SELECTION SCENARIO

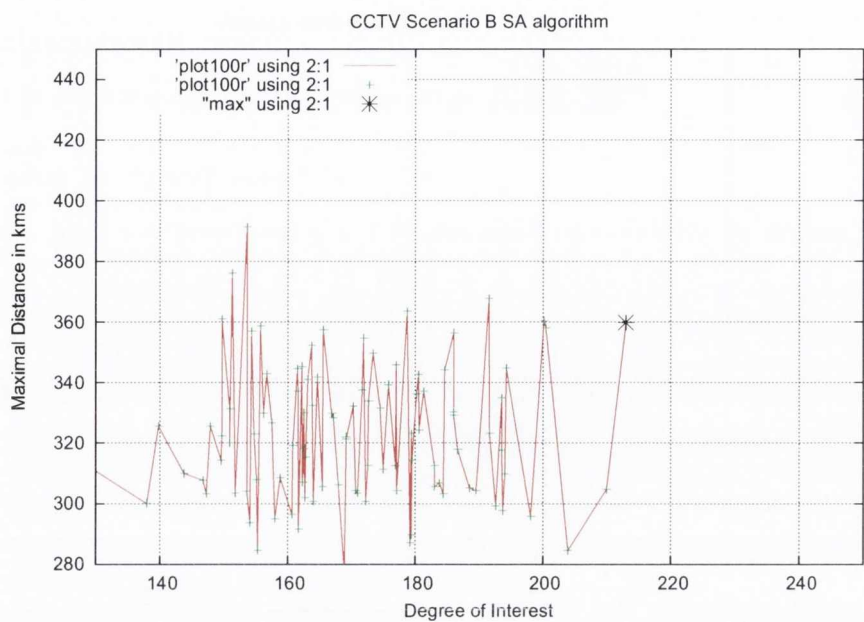


Figure 6.9: CCTV Selection scenario SA performance for 100 runs.

6-8 of Table 6.10 show the optimal results obtained for this scenario using the SA algorithm. Fig 6.8 shows the solution generated by the SA algorithm over 50 evaluation cycles. From Table 6.10 the best solution over 50 runs is found at (191.52, 367.75). This solution is found at a cost of 7173 sensor invocations. The output from this algorithm is visually very different from the corresponding plot of the SMOSA algorithm over 50 runs. As stated above the SA algorithm does not maintain a set of pareto-optimal solutions. Hence, the plot of the solutions is not smooth, rather it is jagged with consecutive solution fitness moving up and down randomly.

Fig 6.9 shows the solution generated by the SA algorithm over 100 evaluation cycles. From Table 6.10 the best solution is found at (212.95, 359.82). This solution is found at a cost of 14344 sensor invocations.

Fig 6.10 shows the solution generated by the SA algorithm over 100 evaluation cycles. From Table 6.10 the best solution is found at (211.01, 373.99). This solution is found at a cost of 71218 sensor invocations.

Column 5 of Table 6.10 shows a normalised ranking of the performance of the SMOSA and the SA algorithms. It shows that the SMOSA algorithm consistently outperforms the SA algorithm for all run counts. In fact the SMOSA algorithm for a run count of 50 with 7118 sensor invocations outperforms the SA algorithm with a run count of 500 and with 71218 sensor invocations. This clearly shows that the choice of optimisation algorithm has a big impact on the quality of solution and the cost associated with obtaining that solution. It also vindicates, for this problem, the selection of the SMOSA algorithm over the SA algorithm for complex problems in the taxonomy.

6.8. CCTV SELECTION SCENARIO EXTENSION

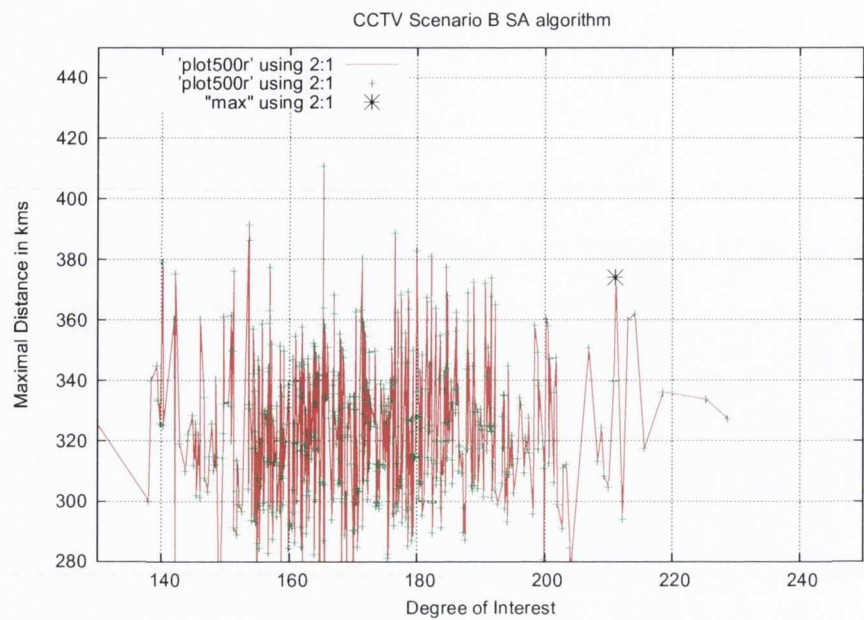


Figure 6.10: CCTV Selection scenario SA performance for 500 runs.

6.8 CCTV Selection Scenario Extension

The optimisation algorithm used in the CCTV scenario selects sets of 30 junctions that maximise the selection criteria specified in the policy. The functionality of the scenario was changed by extending the criteria used to select sets of junctions to display. The optimisation criteria are extended to maximise the number of emergency vehicles, in addition to the existing criteria, at the set of junctions selected.

Layer 1 of the domain model is unchanged. An emergency vehicle sensor type was added to layer 2 for the Junction Controller scenario. This sensor type is also available for use in the extended CCTV Selection scenario.

To accommodate this extended functionality two new system-states are defined in layer 3. An `EmergencyVehiclePresent` system-state is added to the domain model. This system-state definition was used in the Junction Controller Scenario and is reused in the CCTV scenario domain model. The only change required to the definition is that its scope changes from that of a phase element to that of a junction element. This is because in the CCTV Selection scenario it is sufficient to resolve that there is an emergency vehicle present at any phase of a junction. The scope of a junction intersects the scope of all its phases and hence a junction scope can be used in a spatial query to lookup sensors in all regions related to the phases.

As in the previous scenario, the `EmergencyVehiclePresent` state takes one layer 2 input which is a reading from an emergency vehicle present sensor. The original inference function is reused unchanged as shown in Listing 4.9. Although the scope of the system-

6.8. CCTV SELECTION SCENARIO EXTENSION

```
1
2 def calculate(inputs):
3     """
4     sum the number of vehicles present at each junction
5     """
6     sum = 0
7     for datum in inputs["layer3"]:
8         for (x,y) in datum:
9             if x == "true" and y > 0.5: #an emergency vehicle is deemed present
10                here
11                sum += 1
12
13     return sum
```

Listing 6.2: Inference function for NumberEmergencyVehicles system-state.

state has changed the inference function logic remains the same. The spatial queries created by the domain-transformation algorithm will ensure that only sensor readings appropriate to the scope of the state are passed to it at runtime.

A NumberEmergencyVehicles system-state is defined and used to combine the Emergency-VehiclePresent definitions from individual junctions into an overall value for the candidate solution. The specification of this state is similar to that shown in Listing 4.12. The system-state is specified to be dynamic and partially observable and the complexity flag is set to false. The scope element defines the system-state to be of type multi-element, number 30 and value junction. A NumberEmergencyVehicles state-variable object is therefore given a scope of 30 junction elements.

This system-state takes as input a layer 3 element named EmergencyVehiclePresent. At run-time, state-variable objects for the NumberEmergencyVehicles system-state will issue middleware queries for elements of type EmergencyVehiclePresent in a geometry defined over 30 junction elements. The inference function used to combine the inputs and to provide a value for the system-state is shown in Listing 6.2.

Listing 6.2 shows how likelihood data can be used. In line 7, the function iterates through the values returned by the layer 3 system-state inputs. Each of these inputs is provided by a state-variable object returning a likelihood function of the form:

$$[(\text{'false'}, 0.9529), (\text{'true'}, 0.0471)]$$

In line 9, the function evaluates the likelihood associated with a 'true' tag. If it is greater than 0.5 then the count of emergency vehicles in the candidate solution is increased by 1.

The policy file is extended as shown in Listing 6.3, which specifies that the NumberEmergencyVehicles state is discrete, has a weighting equal to the other system-states, and takes a value in the range 0-30. The optimisation algorithm should attempt to maximise the number of emergency vehicles in the solution candidates.

With the addition of these system-states the problem now becomes a three dimensional optimisation problem with the candidate solutions chosen to maximise the Maximal-

6.8. CCTV SELECTION SCENARIO EXTENSION

```
1 <state>
2   <name>NumberEmergencyVehicles</name>
3   <reward>
4     <type>discrete</type>
5     <range>0-30</range>
6     <action>X</action>
7     <value>maximise</value>
8     <weight>1</weight>
9   </reward>
10 </state>
```

Listing 6.3: CCTV Selection scenario extension to policy

Distance, DegreeOfInterest and NumberEmergencyVehicles system-states.

6.8.1 Development Effort for Extension

The addition of the EmergencyVehiclePresent and NumberEmergencyVehicles system-states increase the domain model by 106 lines: 78 lines of XML and 28 lines of python. Of this increase, 10 lines of XML are for the policy extensions, 68 lines of XML are for the system-state definitions and the 28 lines of python code are for the inference functions.

The extension increases the size of the domain modelling effort by c. 40%. However there is no additional AI modelling required as the SMOSA algorithm mapping logic is unchanged. The domain model increases from 284 to 390 LOC while the AI model, i.e., the SMOSA algorithm is unchanged. The increase in development effort associated with the extended scenario functionality is 684/790 LOC = 15%.

6.8.1.1 Degree of Automation

In rows 4-6 of Table 6.4 the planning model generated from the original domain model is shown to be 69 KB in size. In rows 7-9 of Table 6.4 the planning model generated from the extended domain model produces a planning model shown to be 118 KB in size.

The degree of automation measure for the CCTV Selection scenario is:

$$\delta(DM + AI) : \delta(Planning_Model_Size) = 684/790 : 69/118$$

A 15% increase in development effort results in a 71% increase in application functionality as measured by the size of generated spatial query data.

As discussed in Chapter 5, planning models for optimisation problems contain a state-generator object which is used by optimisation algorithms to generate random sets of state-variable objects at run-time to support heuristic exploration of the application state space. State-generator objects hold the set of scopes available to assign to state-variable objects created at runtime. When the CCTV Selection scenario is extended the state-generator object increases in size as it now contains the set of possible scopes for the new system-states. Column 4 of Table 6.4 shows the numbers of state-variable objects used by the

6.8. CCTV SELECTION SCENARIO EXTENSION

control units during an execution cycle. When the CCTV Selection scenario is extended extra state-variable objects are required for candidate solutions.

In row 4 of Table 6.4 the number of state-variable objects used by the SMOSA algorithm for 50 runs using the original domain model is 3050. In row 7 the number of state-variable objects used by the SMOSA algorithm over 50 runs using the extended domain model is 4600. This represents an increase of c. 50% over the size of the original planning model.

Column 2 of table 6.11 shows that the original SMOSA control units performed 7118 sensor invocations over 50 runs. Column 3 of table 6.11 shows that the extended SMOSA control units performed 8823 sensor invocations over 50 runs, an increase of c. 24%. The increased sensor number of sensor invocations reflects the additional functionality provided by the domain model transformation.

A 15% increase in development effort results in an increase of the application functionality by 50% and increases the number of sensor invocations by 24%. This result indicates the expressive power of the domain model abstractions.

No additional AI code is required, demonstrating that the methodology supports reuse and that application logic can be specified and extended using the domain model alone, thereby allowing the domain modeller to focus on application behaviour.

6.8.2 Algorithm Selection

The performance of the SMOSA algorithm in the extended scenario is shown in Figs 6.11-6.13. These 3D figures show the pareto-optimal landscapes found by the optimisation algorithm over the values of the DegreeOfInterest, MaximalDistance and NumberEmergencyVehicles system-states.

The plots are generated using gnuplot and the 3D data are treated as a scattered data set, with a basic low pass filter applied to convert the scattered data to a grid data set (Williams & Kelley 2010). A grid surface is approximated by connecting the points with straight line segments. The dimensions of the grid are derived from a bounding box of the scattered X, Y, Z data. The grid is equally spaced in X (rows) and in Y (columns); the Z values are computed as weighted averages of the scattered points' Z values.

Run Count	Algorithm	2D μ Sensor Invocations	3D μ Sensor Invocations
50	MOSA	7118	8823
100	MOSA	14028	17543
500	MOSA	71712	89276
50	SA	7173	8785
100	SA	14344	17518
500	SA	71218	86872

Table 6.11: CCTV Selection scenario sensor invocation.

Table 6.12 shows the following information:
Each data point is weighted inversely by its distance from the grid point raised to the power of a normalisation constant. A value of 16 has been used to generate the contouring

6.8. CCTV SELECTION SCENARIO EXTENSION

Run Count	Temperature	Figure	3D Maximum			Normalised Maximum
50	500	Fig 6.11	197.10	266.28	22	0.5206
100	500	Fig 6.12	202.88	320.39	19	0.5216
500	500	Fig 6.13	219.55	296.41	22	0.5539
Run Count	Algorithm	Figure	3D Maximum			Normalised Maximum
50r	SA	Fig 6.14	191.52	367.75	14	0.4847
100r	SA	Fig 6.15	212.95	359.82	7	0.4259
500r	SA	Fig 6.16	211.01	373.99	13	0.4933

Table 6.12: CCTV Selection scenario normalised maximum values.

shown in the Figures. The closer the data point is to a grid point, the more effect it has on that grid point. The larger the value of the normalisation constant, the less effect more distant data points have on that grid point.

The optimal solutions which maximise both values appear in red on the top-right of the plots. The plots are colour coded with the progression from grey to green to red indicating that the NumberEmergencyVehicles in the candidate solution is increasing at that region. All figures are plotted using the same scales so that comparisons between run numbers are visually meaningful.

Each figure also shows a normalised maximum and maximum point. The normalised maximum is calculated by weighting equally the sets of DegreeOfInterest(DI), Maximal-Distance(MD) and NumberEmergencyVehicles (NEV) pareto-optimal values.

$$\max\{DI_i/\sum_i^n DI + MD_i/\sum_i^n MD + NEV_i/\sum_i^n NEV\}$$

The normalised maximum value prevents one criterion with large absolute values outweighing a number or criteria with smaller absolute values. In this case it ensures that the NumberEmergencyVehicles state is weighted as much as the MaximalDistance criterion. The importance of this weighting can be seen from the scales used in Fig 6.11 where the NEV variable produces values in the range 0-24 while the MS variable produces values in the range 260-420.

The maximum point is calculated as: $\max\{DI_i + MD_i + NEV_i\}$ and shows the solution with the maximum absolute values for the combined system-states. The numbers of sensor invocations required to obtain these values are shown in Column 4 of Table 6.11.

The earlier analysis of the scenario showed that a starting temperature of 500 returned the best results when averaged over all runs and hence this value is used in the plots presented. To conserve space, plots are not shown for all combinations of run values and starting temperatures.

Fig 6.11 shows the behaviour of the SMOSA algorithm using a temperature of 500 over 50 runs using the extended scenario domain model. From Table 6.12 the normalised maximum is found at (197.10, 266.28, 22). The maximum point is shown at (191.41, 352.99, 12). The physical distance between these two points in Fig. 6.11 illustrates the very different

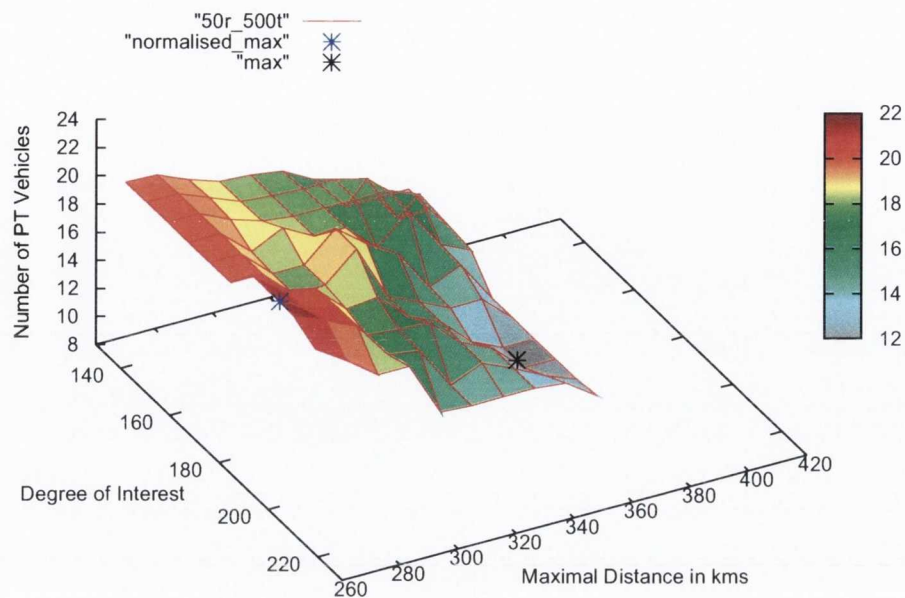


Figure 6.11: Extended CCTV Selection scenario SMOSA performance for 50 runs.

results obtained by weighted and non-weighted results.

Fig 6.12 shows the behaviour of the SMOSA algorithm using a temperature of 500 over 100 runs using the extended scenario domain model. From Table 6.12 the normalised maximum is found at (202.88, 320.39, 19). The maximum point is shown at (194.65, 389.35, 12).

Fig 6.13 shows the behaviour of the SMOSA algorithm using a temperature of 500 over 500 runs using the extended scenario domain model. From Table 6.12 the normalised maximum is found at (219.55, 296.41, 22). The maximum point is shown at (202.69, 409.83, 8). In Fig 6.13 the difference between the normalised and maximum points is greatest.

The dimensions of the plots within the scales of the grid increase as the number of runs increases. This increase in the size of the plots mirrors the increased exploration of the landscape by the algorithm. In Figs 6.11-6.13 the landscape plotted undulates smoothly. The smooth undulation of the solution landscape is explained by the non-dominated property of pareto-optimal solutions.

The performance of the algorithm across all runs and temperatures has been analysed. Rows 6-8 of Table 6.12 show the optimal results obtained for this scenario using the SMOSA algorithm. Column 4 shows the normalised maximum values obtained for the optimisation criteria. Column 5 ranks these values to show the optimal value of the run count and temperature parameters across the experiment. A scalar normalised value is calculated to enable a direct comparison between the results. The optimal result for the SMOSA

6.8. CCTV SELECTION SCENARIO EXTENSION

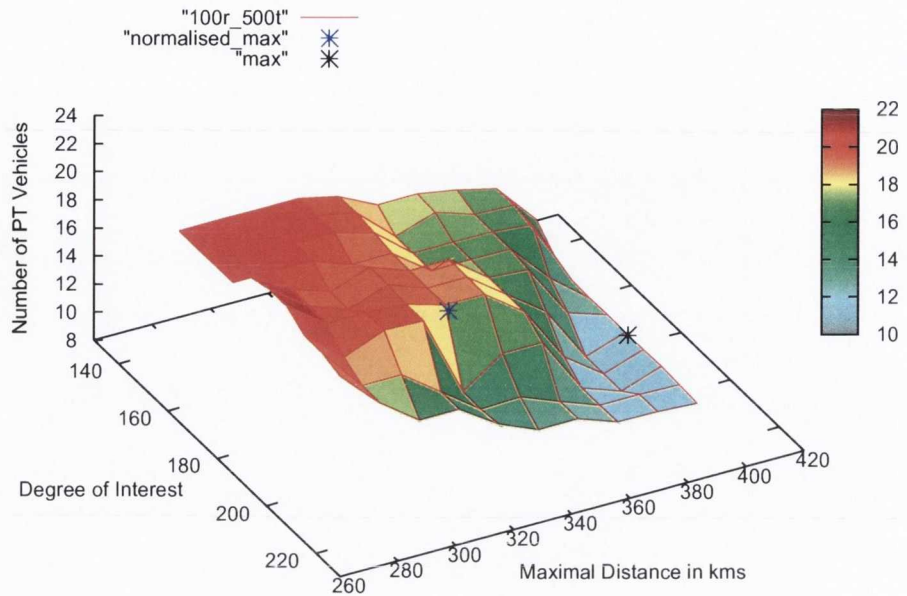


Figure 6.12: Extended CCTV Selection scenario SMOSA performance for 100 runs.

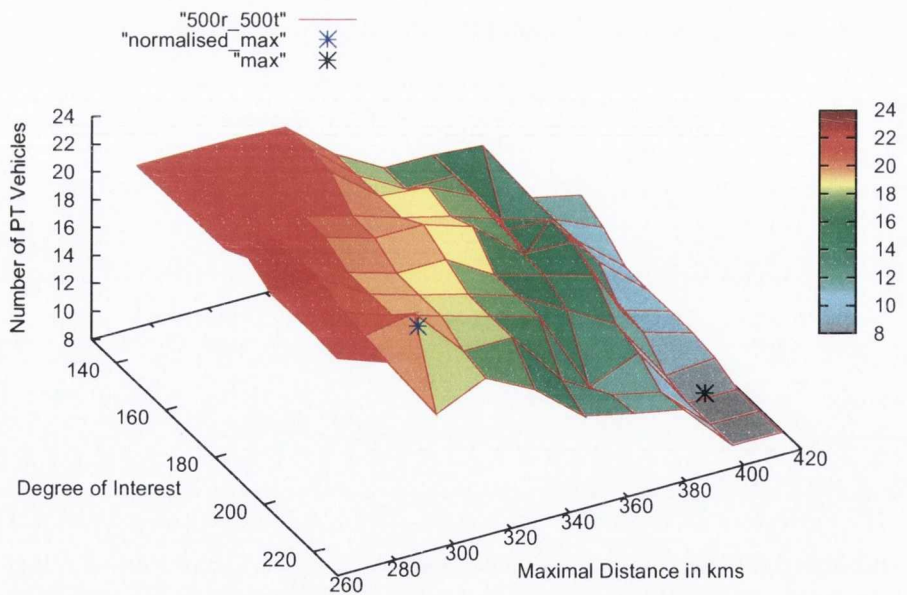


Figure 6.13: Extended CCTV Selection scenario SMOSA performance for 500 runs.

algorithm over the extended domain model is obtained over 500 runs using a starting temperature of 500.

Table 6.11 shows the number of sensor invocations required to produce these results. Varying temperature values result in differing numbers of sensor invocations. Column 4 shows the mean number of sensor invocations for each run count over the range of starting temperatures. However the optimal result normalised at 0.5539 is only 6% fitter than the 0.5206 result achieved after 50 runs. The optimal result is obtained at a mean cost of 89,276 sensor invocations whereas the 50 run result produced useful values after only 8823 sensor invocations, illustrating again the value of careful parameter tuning.

6.8.2.1 SA Algorithm in Extended Scenario

In Section 6.7.2.4, empirical results for the original 2-dimensional scenario were presented to validate the automated selection of the SMOSA algorithm by the taxonomy (Fig 5.1) for complex optimisation problems. In this section we validate the taxonomy selection for the extended 3-dimensional scenario by comparing the performance of a baseline optimisation algorithm over the extended domain model. The local search algorithm from Alg. 3 (SA) is again used as a baseline. Layer 4 of the domain model is used to specify that the SA algorithm is to be used for the scenario. Only one change to the domain model is required to generate a control unit to test the SA baseline performance.

The performance of the algorithm is shown in Figs 6.14-6.16. The set of solutions are mapped with the DegreeofInterest on the X axis, the MaximalDistance on the Y axis and the NumberEmergencyVehicles on the Z axis. As before the plots are generated using a low pass filter gridding technique to generate a 3D surface. All figures are plotted using the same scales so that comparisons between run numbers are visually meaningful. Figures 6.14-6.16 show the performance of the algorithm for 50, 100 and 500 runs respectively.

The (X,Y,Z) points shown are the candidate solutions generated as the algorithm executes. The solution with the highest value is also shown and is generated from summing the values of the criteria from the policy specification. The optimal solutions which maximise all three values appear in red on the top-right of the 3d figure. The number of sensor invocations required to obtain these values are shown in Table 6.11.

Fig 6.14 shows the solutions generated by the SA algorithm over 50 evaluation cycles. From Table 6.12 the best solution is found at (191.52, 367.75, 14). This solution is found at a cost of 8,785 sensor invocations.

Fig 6.15 shows the solutions generated by the SA algorithm over 100 evaluation cycles. From Table 6.12 the best solution is found at (212.95, 359.82, 7). This solution is found at a cost of 17,518 sensor invocations.

Fig 6.16 shows the solutions generated by the SA algorithm over 500 evaluation cycles. From Table 6.12 the best solution is found at (211.01, 373.99, 13). The solution is found at a cost of 86,872 sensor invocations.

The performance of the SA algorithm across all run counts has been analysed. Rows 6-8

6.8. CCTV SELECTION SCENARIO EXTENSION

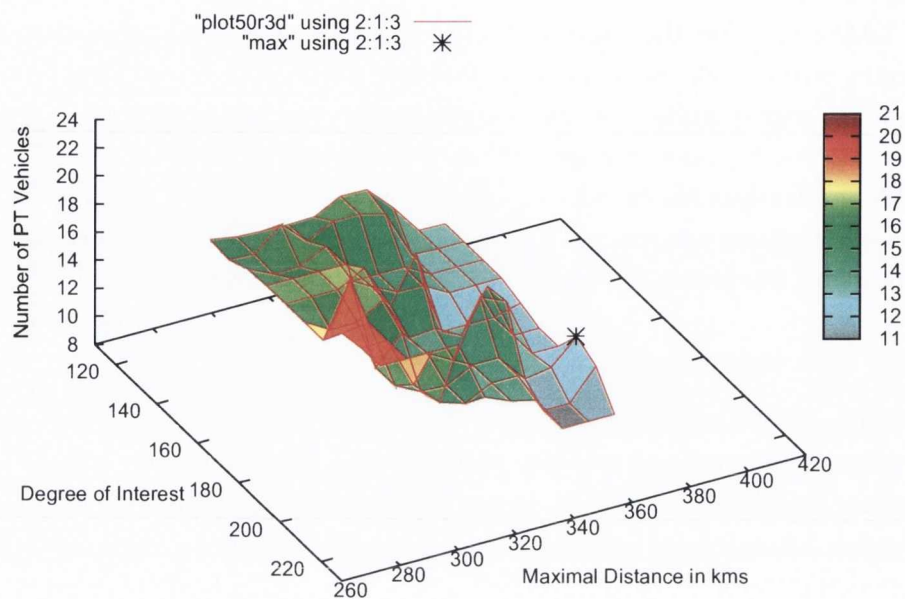


Figure 6.14: Extended CCTV Selection scenario SA performance for 50 runs.

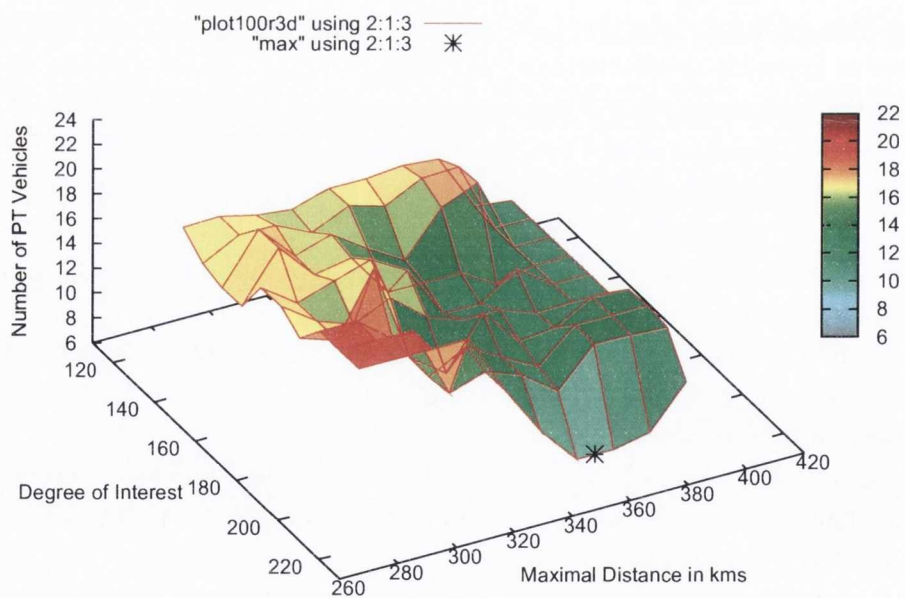


Figure 6.15: Extended CCTV Selection scenario SA performance for 100 runs.

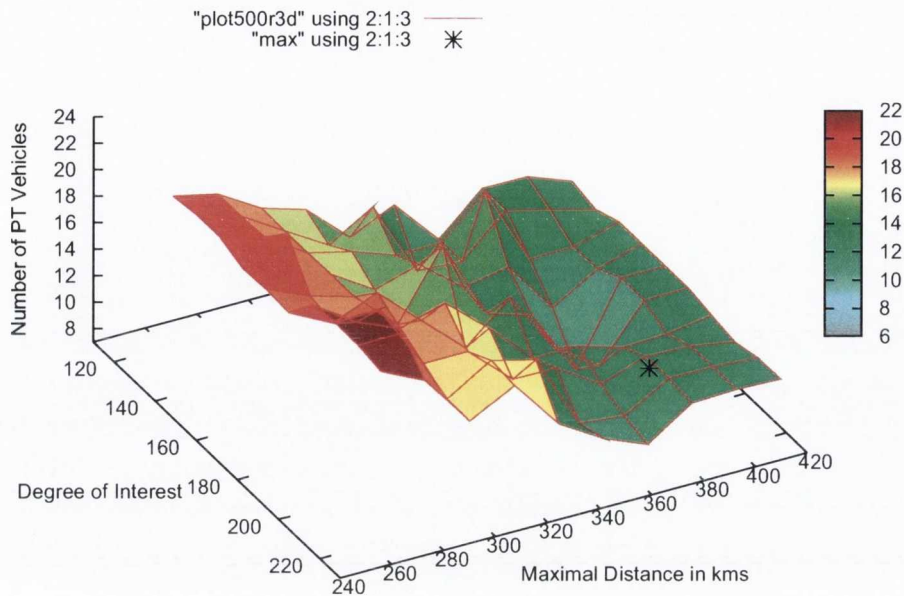


Figure 6.16: Extended CCTV Selection scenario SA performance for 500 runs.

of Table 6.12 show the optimal results obtained for this scenario using the SA algorithm. Column 4 shows the maximum values obtained for the optimisation criteria. Column 5 ranks these values to shown the optimal values obtained for the extended scenario using both the SMOSA and SA algorithms. A scalar normalised value is calculated to enable a direct comparison between the results.

It shows that the SMOSA algorithm consistently outperforms the SA algorithm for all run counts. In fact the SMOSA algorithm for a run count of 50 with 8,823 sensor invocations outperforms the SA algorithm with a run count of 500 and with 86,872 sensor invocations. Using the scalar optimal values from Column 5, the SMOSA algorithm over 50 runs returns a result that is c.6% greater than the SA algorithm over 500 runs. This means that the SMOSA algorithm is more effective than the SA algorithm at finding solution sets of junctions that maximise the system-state values. This clearly shows that the choice of optimisation algorithm has a big impact on the quality of solution and the cost associated with obtaining that solution. It also vindicates the selection of the SMOSA algorithm over the SA algorithm in the taxonomy.

The dimensions of the plots within the scales of the grid increase as the number of runs increases. This increase in the size of the plots mirrors the increased exploration of the landscape by the algorithm. A visual comparison between the SMOSA 3D plots and the SA 3D plots reveals two interesting properties. Firstly, the surface of the 3D SMOSA plots are much smoother than the surface of the 3D SA plots. The choppy surface of the

SA plots stems from the random generation of solutions and the lack of pareto-optimality in selecting solutions. Secondly, the SMOSA plot for a run number covers a larger surface area than the SA plot for the corresponding run number. This indicates that the SMOSA algorithm is more effective at exploring the optimisation landscape for complex problems and provides some empirical validation the selection criteria in the taxonomy.

6.9 Evaluation Summary

This chapter has described the evaluation of the methodology for engineering planning and optimisation algorithms for use in pervasive computing environments. The chapter began by listing the requirements and mapping them to the design decisions underpinning the methodology. The methodology fuses a MDE approach with techniques from automated planning. Accordingly the evaluation is divided along two axes: firstly to assess the impact of the methodology on developing the target class of applications and secondly to assess the automated planning component. The evaluation criteria are obtained from the literature relating to both areas.

The evaluation strategy is derived from mapping these standard evaluation criteria to observable metrics for the development effort of using the methodology and the performance of the control units generated by the automated planning component. The metrics are used as the basis for an empirical analysis of the methodology based on applying it to the Junction Controller and CCTV Selection scenarios.

We summarise the findings relating to the evaluation criteria E1-E8 listed in Section 6.2. A range of requirements for the methodology were identified in Chapter 1. The chapter ends with a discussion relating the evaluation findings to the methodology requirements R1-R7.

6.9.1 Evaluation Findings

- Productivity gains and Automation. (E1, E3)

In the original and extended Junction Controller scenario, the domain and AI models are 567 and 627 LOC respectively. Increasing the scenario functionality by c. 33% required only an increase of 11% in development effort by the domain modeller.

In the original and extended CCTV Selection scenario, the domain and AI models are 684 and 790 LOC respectively. Increasing the scenario functionality by c. 71% required an increase of only 15% in development effort by the domain modeller.

- Abstraction. (E2)

The original Junction Controller scenario policy and system-state definitions contain 147 LOC. Of this c. 92% is specified in XML. The CC metric shows that the complex code for this scenario is restricted to the AI algorithms used in the scenario.

The original CCTV Selection scenario policy and system-state definitions contain 284

LOC. Of this c. 45% is specified in XML. The CC metric again shows that the complex code for this scenario is generally restricted to the AI algorithms used in the scenario. However the inference function used in the `JunctionInterest` system-state contains complex logic which must be specified by the domain modeller in Python.

An examination of the listing shows that there is no scenario specific logic hardcoded in the inference function. The function provides a generic mapping for a Bayesian network constructed using the Hugin Inference engine, to a collection of state-variable objects generated by an optimisation state-generator object. This suggests that a high-level inference API could be provided to simplify the task of specifying inference functions in the domain model.

Both these scenarios show that a high proportion of the domain model can be specified in the high-level XML language. However, when complex inference functions are required the complexity of the domain modelling task increases significantly. This observation motivates a future investigation of what domain model abstractions could be added to support the specification of inference logic.

- Reuse. (E4)

The case studies have demonstrated that a single domain model can be reused with multiple algorithms. The Junction Controller scenario shows a MEU, round-robin, and random-action selection algorithms used with a single domain model. The CCTV Selection scenario shows a SMOSA and SA algorithm used with a single domain model. Both case studies have used a common set of layer 1 domain model elements and share some layer 2 sensor elements, demonstrating that domain model elements can also be re-used in multiple applications.

The analysis also shows that requirements re-engineering resulting in significant functionality changes requires no additional algorithm development effort demonstrating that planning and optimisation algorithms can be reused with multiple domain models.

- Problem Range. (E5)

The analysis of the assumptions underlying the design of the domain model and algorithms highlights that the scope of the methodology is limited by its assumption of implicit time. Also the domain model does not provide a way to represent events not associated with sensor readings or actuator actions and so cannot be used to model the internal dynamics of an environment. However, the domain model and algorithms presented in the scenarios can be used to model and manage partially observable environments and react to dynamism in the environment. The methodology as validated in the scenarios can be used to solve single-decision planning problems and complex optimisation problems in pervasive computing environments.

- Control Tuning. (E6)

The CCTV Selection scenario demonstrates that the methodology supports the speci-

6.9. EVALUATION SUMMARY

fication of control tuning information at layer 4 of the domain model. The importance of control tuning is shown in the original and extended CCTV Selection scenarios. In the original scenario, algorithm tuning showed a 90% reduction in the cost, as measured in sensor invocations, led to only slightly degraded algorithm performance.

- Problem Size. (E7)

The domain model design can scale to accommodate arbitrarily large state spaces. A subtle alteration in the behaviour of the SMOSA optimisation algorithm was observed as the dimensionality of the problem increases. This alteration compromises the ability of the SMOSA algorithm to escape local maxima.

- Algorithm Selection. (E8)

The algorithm taxonomy also encodes expert knowledge in the form of automated algorithm selection criteria. This expert knowledge must be validated empirically on a case-by-case basis. Although it is not possible to offer theoretical guarantees on the selection criteria, the performance of the MEU and SMOSA algorithms in their respective scenarios offer empirical validation of the automated selection of algorithms for a partially-observable single-decision problem and for a complex multi-objective optimisation problem.

6.9.2 Requirements Coverage

- (R1). Allow application logic to be specified at a high-level.

The evaluation findings (E2) show that a large portion of the domain model and all the policy can be specified in XML. The scenario extensions have shown that application logic can be extended without any additional algorithm development.

- (R2, R3). Provide abstractions masking run-time conditions and simplify the development of planning and optimisation algorithms in pervasive computing environments. The case studies in chapter 4 demonstrated the programming model supported by the methodology. Neither the domain or AI modeller is ever required to write any code to access sensors or actuators. The spatial queries generated by the domain-model transformation handle the interaction between the control units and the assumed middleware services and the planning-model components provide an interface to algorithm developers.

- (R4). Provide a development model that is extensible, promotes reuse and is suitable for use in large-scale pervasive computing applications.

The evaluation findings (E1, E3) demonstrate the methodology support for reuse of domain model components and planning and optimisation algorithms. The discussion relating to the range of problems supported (E5) by the methodology has argued that the addition of algorithms to the taxonomy provides a mechanism to widen the range of problems that can be solved using the methodology.

- (R5). Rapidly test algorithm efficacy.

The control tuning support at layer 4 of the domain model coupled with the generation of control units allows algorithm functionality to be quickly customised and tested. The provision of an evaluation platform allows the performance of the algorithms to be analysed. The evaluation findings relating to (E5) provide examples of control tuning and its impact on algorithm performance.

- (R6). Provide a consistent representation of uncertain data.

Uncertain data is modelled using likelihood functions. Section 2.3 provides a detailed survey of techniques for modelling and using uncertain data and justifies the use of likelihood functions within a Bayesian framework.

- (R7). Automated selection of appropriate planning and optimisation algorithms.

The evaluation findings (E8) demonstrate the support for this requirement for a single-decision planning problem and a complex optimisation problem. Algorithm selection criteria must be determined empirically. However, given that the methodology supports the rapid customisation and generation of algorithm instances, it suggests that the domain modeller could experiment with a range of algorithms on a problem by problem basis and determine an appropriate algorithm based on performance observations.

The empirical evaluation indicates that use of the methodology reduces development effort through the provision of expressive abstractions and automated code generation. This result, while pertinent primarily to the case studies, provides encouragement. The evaluation has also shown that the methodology supports reuse of domain-model specifications and planning and optimisation algorithms. The analysis of the performance of the SMOSA and SA algorithms shows the importance of control tuning and algorithm choice. Finally, the evaluation has demonstrated the methodology's support for rapidly testing the performance of a range of algorithms in pervasive computing applications.

Septimus : So the Improved Newtonian Universe must cease and grow cold?

Thomasina : Yes, we must hurry if we are going to dance.

Septimus : When we have found all the mysteries and lost all the meaning, we will be alone on an empty shore.

Thomasina : Then we will dance. Is this a waltz?

Septimus : It will serve.

Tom Stoppard (Arcadia)

7

Future Work and Conclusions

This thesis presents a MDE methodology for applying planning and optimisation algorithms in pervasive computing environments. The design of the methodology combines techniques from the disciplines of software engineering, and planning and optimisation.

7.1 Achievements

The contributions of this thesis are a MDE methodology for a challenging application domain, a domain model suitable for specifying large-scale pervasive computing applications, and model transformation algorithms that support the selection and engineering of algorithms and application code. The generated code contains middleware hooks for accessing sensor and actuator infrastructure in the deployment environment.

The methodology tool chain systematises the development of the target class of applications. The domain-model schemas provide a blueprint for specifying the information required to build applications requiring planning and optimisation algorithms

The empirical evaluation of the impact of the methodology on development effort demonstrates the productivity gains achieved by the combination of expressive domain-model abstractions and automated code generation when developing and extending applications. The case studies demonstrate that the methodology design supports the reuse of domain model data and algorithms and is extensible through the addition of algorithms to the library.

The thesis also investigates the challenging problem of automated selection of appropriate planning and optimisation algorithms for problems and provides a taxonomy linking problem types and domain characteristics to algorithm selection. While the mechanism for

automated selection is presented, no claims are made for the optimality or appropriateness of solutions.

The analysis of algorithm performance presented in the case studies demonstrates the importance of tuning algorithm performance. Even if the algorithm selection criteria are correct, it is difficult to see how useful values for algorithm parameters could be derived from domain-model information. It is more probable that careful tuning of algorithm parameters using empirical analysis of performance using domain-model data will be necessary to derive useful values for parameters. In this regard the methodology allows the domain modeller to rapidly test a variety of algorithms with a single domain model and to tune parameters accordingly.

7.2 Future Work

The research work summarised in the thesis has addressed a broad range of issues and consequently there are many areas of possible future work, however this section restricts itself to listing some practical extensions to the methodology. The future work is divided into the areas of domain modelling, planning and optimisation algorithm support, and code generation

7.2.1 Domain Modelling

A basic extension would be to build a graphical (GUI) tool to support domain modellers in constructing domain models. A domain-model GUI would remove non-essential knowledge required to construct domain models. Examples of such non-essential knowledge includes referencing system-state values in policy files and ensuring that a reward is specified for each possible value of a discrete system-state. A GUI could also use the domain-model schema to validate the models and could perform sanity checks such as checking that infrastructure elements referenced by system-state scopes are contained in the domain model. The suitability of the Graphical Modelling Framework (GMF), provided by the Eclipse platform ¹, to construct such a GUI should be investigated.

The domain model is specified as a series of XML documents and Python functions. The Eclipse Modelling Framework (EMF) and Ecore metamodel could be used to provide a higher-level graphical notation for specifying the XML component of the domain model. Such a step would motivate a more heavyweight implementation of the transformation algorithms using the Java programming language. The use of a higher-level graphical notation might also make it easier to identify opportunities for domain model element reuse across applications.

More interesting future work would be to develop further abstractions to allow more complex application state spaces and policies to be represented using the domain model. The domain model currently uses a topology abstraction to map high-level application

¹<http://www.eclipse.org/gmf/>

7.2. FUTURE WORK

state-space models to sensor and actuator infrastructure. The topology abstraction is very useful for applications executing in the physical environment. However, other useful abstractions exist for modelling application state and behaviour in pervasive computing applications.

Examples of other abstractions that could be investigated include time, role or activity, and identity. Adding a temporal abstraction to the domain model would allow time-dependent actions and rewards to be specified. Likewise adding role and identity abstractions would allow more traditional context-aware applications to be specified using the domain model and to access the planning and optimisation algorithms supported by the methodology.

A related extension would be to extend policy syntax to allow richer reward models to be specified and to specify constraints on application behaviour or destination/termination states for some planning problems.

A final extension of the domain model, would be to extend the sensor model to allow covariance information to be specified for non-scalar sensor types so that covariance intersection (CI) methods could be used for sensor fusion (Julier & Uhlmann 2001).

7.2.2 Algorithm Library

The algorithm library could be extended to provide additional support for sequential-decision planning algorithms. The current algorithm library contains an implementation of an Online-Approximate POMDP algorithm for sequential-planning problems in dynamic partially-observable environments. (See Chapter 2). An interesting area for future work would be to amend the algorithm by varying the planning horizon or search depth, in relation to the degree of observability over application state afforded by the sensor and actuator infrastructure in the deployment environment. This might allow the exploration of the state space to be tailored according to the quality of sensor infrastructure in the deployment region and potentially reduce redundant sensor invocations.

The ability of the planning model transformation algorithm to generate a sequential decision control unit is dependent on a fully-specified transition system being encoded in the domain model. An interesting extension would be to add reinforcement-learning algorithms to the library so that planning techniques can be used when no transition-system information has been specified in the domain model.

7.2.3 Transformations and Code Generation

The transformation algorithms are currently implemented in Python. This design decision was influenced by the need to rapidly produce working prototypes. It would be interesting to investigate the suitability of standard MDE tools such as the Java Emitter Template language² and the Atlas transformation language³ for providing the model transformation

²<http://www.eclipse.org/emft/projects/jet/>

³<http://www.eclipse.org/m2m/atl>

functionality.

The domain-model transformation algorithm could be extended to provide components that simplify the task of specifying state inference functions. The evaluation of the CCTV Selection scenario showed that the domain modeller was required to specify complex logic in the `enter_sensor_data()` function to map layer 2 data objects onto a Bayesian network used to infer values for the `JunctionInterest` system-state. An examination of this function, shown in Listing 4.12, reveals that it contains no scenario-specific code and provides a generic function to map layer 2 data objects to Bayesian network node types used in the scenario. This functionality could be extended to provide a reusable planning-model component for mapping sensor data to Bayesian networks.

The planning-model transformation currently has a limited concept of execution models. The control units generated by the planning-model transformation assume a simple agent model of execution. Stronger models of agency could be provided if the planning-model transformation could be integrated with agent modelling frameworks such as Jack (Papasimeon & Heinze 2001) and Jade (Bellifemine et al. 2007).

The planning-model transformation must be aware of the middleware API used to access sensor and actuator infrastructure at run-time. This dependency could be removed by providing descriptors for well-known pervasive computing middlewares, which could be used by the planning-model transformation to generate control units with middleware hooks appropriate to the deployment environment. This would effectively split the planning transformation into two steps. The initial step would perform algorithm selection and the second would generate the control units complete with middleware hooks and ready for deployment.

Algorithm selection is performed by the planning-model transformation using design-time information. This means that the domain modeller must implicitly provide the information required to select an algorithm. It would be interesting to investigate deployment-time or run-time algorithm selection. State-variable objects could be deployed in advance of algorithm selection and used to test properties of the deployment environment such as dynamism and observability. This information would then be used to select an algorithm for use. Such an approach would be more appropriate for richer sequential planning algorithms based on the POMDP model.

Autonomic computing is a research field focused on the provision of self-managing systems (Kephart & Chess 2003). Run-time algorithm selection would add a self-managing aspect to the methodology. The ability of an autonomic system to self-manage is based on several self-* (self-star) properties: self-optimisation, self-configuration, self-healing, and self-protection (Ganek & Corbi 2003). Tailoring algorithm selection to run-time environment properties would provide a mechanism for delivering the self-optimisation of application execution through the automated monitoring and deployment of appropriate planning and optimisation algorithms.

Finally, one area not addressed by this work is how the control-unit functionality would be integrated into other application code. An interface could be developed to provide a

standard method of integrating control-unit functionality into other applications.

7.3 Conclusions

We conclude the thesis with a high-level discussion of whether the methodology simplifies the task of applying planning and optimisation algorithms in pervasive computing applications. To frame the discussion we refer to the famous "No Silver Bullet." article published by Fred Brooks (Brooks 1987). This is a seminal publication whose impact and validity still informs software engineering research (OOPSLA 2007). Brooks, as quoted in Chapter 4, referred to the essential complexity of software models and states:

The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.

Inter-locking concepts are a fundamental cause of complexity, however, pervasive computing applications possess an inherent modularity in that relevant concepts often map to physical entities such as sensors and actuators and application states are often grounded in real-world processes and activities. If domain-model abstractions can be provided to allow a problem specification to map to real-world entities then there is a potential to exploit the natural modularity of the real-world when designing pervasive computing applications, which may not be present in other areas of software development. Brooks continues:

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.

The policy and domain model specifications constitute the main activities performed by the domain modeller when using the methodology. The methodology positions these activities at the centre of the application development effort and motivates a policy or behaviour-centric approach to application design, i.e., where the domain modeller creates applications primarily through high-level specifications of what they want the application to do. A development model that allowed a domain expert to create systems primarily through specifying desired behaviour using a high-level language composed of domain relevant constructs would go some way to alleviating the essential complexity identified by Brooks.

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems.

7.3. CONCLUSIONS

For almost 40 years, people have been anticipating and writing about "automatic programming," or the generation of a program for solving a problem from a statement of the problem specifications.

If the policy specification allows application behaviour to be specified, then the detailed technical requirements to achieve the desired application behaviour are the plans and system-state configurations supplied by planning and optimisation algorithms. The question that arises is what is desirable and what is possible to automate?

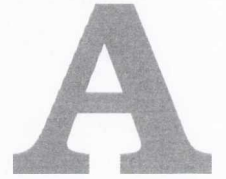
State inference is essentially a difficult task often requiring domain-specific knowledge. The case studies presented in the thesis have used an inference framework based on Bayesian networks. While support can be provided to automate secondary inference tasks such as competitive fusion and mapping sensor data to Bayesian network nodes, the primary task of specifying network structures and conditional probabilities cannot be easily automated and will remain a challenging task. Classical planning and optimisation algorithms often use root-finding and loss function gradient techniques also requiring domain-specific knowledge. However, as discussed in Chapter 2, the very challenging nature of pervasive computing environments motivates the use of approximate planning and heuristic optimisation techniques. Such techniques do not require as much information about problem structure and hence can be more easily automated.

Brooks does offer some hope for a silver bullet to tackling the essential complexity of software development:

..one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements.

The automated transformations provided by the methodology allow the rapid generation of control-units and facilitates the testing of multiple algorithms to determine which produces the most useful application behaviour. However as evidenced by the analysis of algorithm performance in the CCTV Selection scenario, control tuning of algorithm parameters impacts significantly on performance. The domain model allows algorithm parameters to be tuned however this requires experimentation by the domain modeller to determine the most useful values and presents an inherent limitation on the benefits of automation.

Finally, with reference to Dijkstra's admonition for the humble programmer quoted in Chapter 5, we conclude that our methodology addresses some of the circumstantial cumbersomeness involved in applying planning and optimisation algorithms to pervasive computing environments and provides tools to the programmer to attend to the more difficult challenges.



Complexity

Complexity in pervasive computing applications may arise due to issues such as state space size, the degree of interconnectedness among system elements and partially-understood or stochastic state-transition systems.

The complexity of a problem can act as an input to the selection of an appropriate algorithm for the system. If the complexity of a problem is such that a deterministic or brute force solution is be infeasible given computational or time constraints, then a heuristic approach may provide useful though not optimal results.

This section introduces some complexity measures that may be roughly divided into those that measure the complexity of an algorithm versus those that measure the complexity of a problem.

Algorithmic Complexity

The complexity of an algorithm can be expressed as a function of its running time or space requirement. Consider a procedure $p(s)$ that takes a single argument s and let $T(p, s)$ and $S(p, s)$ respectively be the running time and the space requirement for $p(s)$. Consider the input s as a string of symbols, binary or ASCII. Then the procedure p 's worst-case running time and space requirement are defined, respectively, as follows:

$$T_{\max}(p, n) = \max\{T(p, s) \mid |s| = n\}$$

$$S_{\max}(p, n) = \max\{S(p, s) \mid |s| = n\}$$

Complexity analyses of $T_{\max}(p, n)$ and $S_{\max}(p, n)$ normally focus not on the exact values of $T_{\max}(p, n)$ and $S_{\max}(p, n)$ but instead on their *big O* values.

Using this notation $O(f(n))$ is the set of all functions that grow no faster than propor-

tional to $f(n)$. Stated formally, a function $f(n)$ is in the set $O(g(n))$ if there are numbers c and n_0 such that:

$$\forall n > n_0, 0 \leq f(n) \leq cg(n)$$

A function $f(n)$ is said to be logarithmically bounded if $f(n) = O(\log(n))$; polynomially bounded if there is a constant c such that $f(n) = O(n^c)$; and exponentially bounded if there is a constant c such that $f(n) = O(c^n)$ (Ghallab et al. 2004).

This notation provides an asymptotic analysis because as the input size n asymptotically approaches infinity an $O(n)$ algorithm is better than an $O(n^2)$ algorithm.

Problem Complexity

When faced with a pervasive computing task it may be more relevant to classify the complexity of the given problem rather than that of an algorithm. Computational complexity theory analyses problems rather than algorithms and is concerned with the question for which decision problems do efficient algorithms exist.

Traditional computational complexity analyses focus on decision problems in general and on language recognition problems in particular (Ghallab et al. 2004). There are standard procedures for restating any planning problem as a language recognition problem (Ghallab et al. 2004) hence it is possible to perform a complexity analysis on pervasive computing problems.

Pervasive computing problems can be treated as planning problems or decision problems where a decision problem is any problem that takes as input a finite length binary string and returns as output a 0 or 1 answer.

Complexity theory is based on the standard Turing model of computation and measures the efficiency of decision problems by measuring the time and space resources of a Turing machine required to solve a decision problem of finite length. A key definition of complexity theory is that *feasible* or *tractable* means polynomial time (Rudich Steven 2004).

The class of decision problems that have a worst-case running time of $O(n^k)$ for some k is called **P**. These problems can be solved deterministically in polynomial time and are hence considered to be the 'easiest' or most tractable problems.

Another important class of problems is **NP**, the class of non-deterministic polynomial problems. This is the class of problems for which a solution can be verified but not generated quickly, i.e. polynomially. The hardest set of problems in this class are **NP Complete** problems. Establishing that a problem is **NP Complete** is useful as heuristic approaches should then be adopted in place of any further analysis of the problem.

An excerpt from the complexity class hierarchy in (Ghallab et al. 2004) is:

$$\text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXSPACE}$$

However in practice both the complexity and the size of a problem need to be taken into consideration. For example a decision problem which runs in $O(n^{100})$ is in **P** but is intractable for practical purposes.

Combinatorial Complexity

A closely related but more intuitive measure of complexity is that provided by Kauffman (Stuart 1995). Imagine a gene sequence as a number in binary format, each digit is a gene whose value is 0 or 1. Each gene sequence has a particular fitness. The fitness of each sequence depends on the values of the individual genes. At its simplest, the fitness of each gene could depend solely on its own state. At its most complex, the fitness of a gene could depend on the state of all other genes. If there are N genes and each genes fitness depends on the state of K others, the gene sequence can be modelled as an NK landscape or network with N nodes and K edges between nodes.

Kauffman's NK model describes such networks of coupled or interacting nodes. It states that for a set of N binary nodes (with each node only capable of 2 values) with K (epistatic) coupled nodes, the dimensionality of the system increases by $N2^{K+1}$. It has been shown that when $K \geq 2$ the problem is **NP Complete** (Weinberger 1996).

To establish the complexity of an application domain, an NK complexity model is used to characterise the difficulty of a problem according to its size and the interconnectedness of the elements that make it up.

Complexity in Pervasive Computing Environments

The use of complexity analysis in pervasive computing is complicated by the fact that such analysis is applied to stationary and finite decision problems. In order to discuss the computational complexity of planing problems, the problems must be reformulated as language-recognition problems.

This approach may not be useful when dealing with many of the problems faced by application developers in pervasive computing environments, many of of which are dynamic and continuous in nature.

An NK model could provide a more robust measure of the complexity of a pervasive computing system as follows: Consider a UTC system to be deployed in a city region that contains 50 traffic-light controlled junctions. Suppose there are 2 actions that can be taken to alter the state at each controller: alter the cycle time; and switch the current phase. The size of the state space (of decision variables) is 2^N where N is the number of junctions (or binary variables). The state space of decision variables is then 2^{50} .

This is a huge number but if the decisions taken at each junction were independent of each other then the task could be parallellised. However the degree of connectedness or interference between junction controllers further complicates the decision making and evaluation process. In a UTC scenario, the state of each junction is somehow affected by the state of the immediately adjacent junctions. If a junction has 4 neighbours whose state directly affect it, ie $K = 4$, the NK model gives a complexity landscape of $2^{50 \times 2^4}$.

A naive brute force enumeration and search through the state space would be inappropriate here. Algorithm selection must take account of the complexity and size of a problem.

More fundamentally though pervasive computing applications even if formulated as planning problems will typically operate in continuous noisy environments. Complexity

theory deals with finite or bounded decision problems with observable and enumerable inputs. (Spall 2003) recognises this difficulty and states that any static measure of problem difficulty will always be severely limited.

B

Fuzzy Logic

The third technique that we examine for representing and reasoning about uncertain information is fuzzy logic and set theory. Fuzzy set theory can reason about variables whose values are not well defined. Consider a traffic control system in which the time of day is classified as morning rush-hour, daytime off-peak, evening rush-hour or night-time off-peak. It may not make sense to define crisp thresholds for the transition from morning rush-hour to daytime off-peak or from off-peak to evening rush-hour. In this case, the value that a time-of-day variable takes on is fuzzy and there is no clear boundary between the end of one value and the beginning of another.

Fuzzy set theory (FST) is a means of specifying how well an object satisfies a vague description. Consider the proposition *The current traffic flow context is morning rush-hour*. Is this true if the time is 10:30 ? The issue here is not a question of uncertainty about the value of the time variable. The issue is that the term *morning rush-hour* does not refer to a sharp demarcation of objects into two classes. The traffic flow associated with morning rush-hour does not abruptly end at any particular time. FST treats *Morning rush-hour* as a fuzzy predicate and says that the truth value of *Morning rush-hour(10:30)* is a number between 0 and 1.

Fuzzy Membership

Membership in classical set theory is binary. A membership query returns true if an object is a member of a set and false otherwise. In contrast, fuzzy set membership queries return values between a range between 0 (not a member) and 1 (a member). A fuzzy membership function maps every object x (from the universe of discourse, i.e. the set of all possible objects X) to a number between 0 and 1. This number gives a degree of membership of a particular fuzzy set. This is expressed formally as: $\mu_A : X \rightarrow [0, 1]$

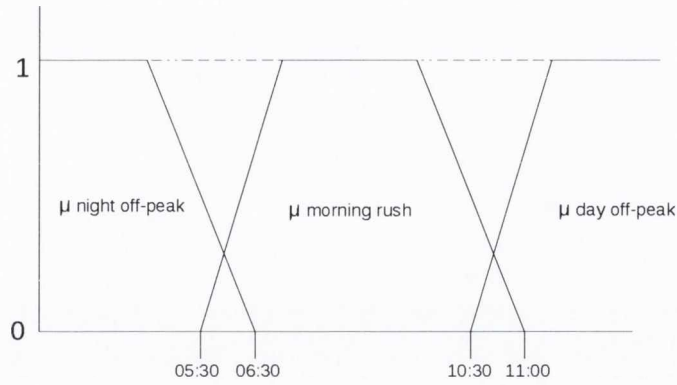


Figure B.1: Fuzzy Membership Time Context Function

The membership function for a set A , denoted μ_A , takes an object (or measurement) x and returns a number between 0 and 1. A membership function to describe traffic contexts could be denoted, $\mu_{\text{morning_rush_hour}}$, and would take values such as 4:30, 7:00, 8:00 and return a number in the interval $[0, 1]$. $\mu_{\text{morning_rush_hour}}(4 : 30) = 0$, $\mu_{\text{morning_rush_hour}}(7 : 00) = 0.5$, $\mu_{\text{morning_rush_hour}}(8 : 00) = 1$. Other membership functions could be defined for $\mu_{\text{daytime_off_peak}}$ and $\mu_{\text{evening_rush_hour}}$.

Fuzzy Logic

Fuzzy logic is a method for reasoning with logical expressions describing membership in fuzzy sets. For example, the complex sentence:

$$\text{morningrush} - \text{hour}(8 : 30) \cup \text{heavy} - \text{traffic}(5\text{v/s})$$

has a fuzzy truth value that is a function of the truth values of its components. This sentence queries two fuzzy membership functions and evaluates whether 8:30 a.m is morning rush hour or if a traffic flow of 5 vehicles per second is heavy traffic. The standard rules for evaluating the fuzzy truth T of a complex sentence containing membership functions A and B are:

$$T(A \cap B) = \min(T(A), T(B)) \quad (\text{B.1})$$

$$T(A \cup B) = \max(T(A), T(B)) \quad (\text{B.2})$$

$$T(\neg A) = 1 - T(A) \quad (\text{B.3})$$

These rules do not take correlations or co-dependence between components into account.

Fuzzy Controller

Many applications of fuzzy control are in engineering control applications such as cruise control or automatic transmission systems in vehicles, auto focus of a camera, etc. Each of these applications have continuous variables such as speed, light intensity, etc. Fuzzy logic takes such input variables such as sensor readings and measurements and maps

them using rules to an output variable or action. (Callan 2003) provides a cruise control example where the output variable is the adjustment to be applied to the accelerator or brake to maintain a desired speed. A fuzzy controller consists of a set of rules (i.e. if/then statements) and an inferencing procedure. The “if” part of a rule is matched against the fuzzy values and the “then” part of a rule is associated with the output variable of interest. The fuzzy values have an associated degree of membership which are combined to indicate the strength of the conclusion.

Two variables in the system are: SpeedError(se) = Actual–Desired and Acceleration(acc). Assume the possible values for these two variables are: Zero (Z), Positive (P), Negative (N).

The membership function, μ_{se} , for the Speed Error (se) variable is composed of three separate functions μ_{Neg} , μ_{Zero} and μ_{Pos} .

Similarly, the membership function, μ_{acc} , for the Acceleration variable is composed of three separate functions μ_{Neg} , μ_{Zero} and μ_{Pos} .

The possible outputs are to increase/reduce speed by a small or large amount using the accelerator or brake or to continue with the status quo (i.e. constant speed). Assume also that the fuzzy controller has a set of rules defined in the form of fuzzy logic expression : action. A set of rules for the scenario might look as follows:

- If se = Z and acc = Z then Constant
- If se = Z and acc = P then Reduce Small
- If se = Z and acc = N then Increase Small
- If se = P and acc = Z then Reduce Small
- If se = P and acc = P then Reduce Large
- If se = P and acc = N then Constant
- If se = N and acc = Z then Increase Small
- If se = N and acc = P then Constant
- If se = N and acc = N then Increase Large

Combining Membership Functions

Given the fuzzy logic rules defined previously, it is possible to construct a new membership function $\mu_{se} \cap \mu_{acc}$ where: $\mu_{se} \cap \mu_{acc}(x) = \min(\mu_{se}(x), \mu_{acc}(x))$.

If the rule has “and”, then the new membership function represents the intersection of the sets se and acc. The new membership function is obtained by taking the minimum of the existing membership functions.

If the rule has “or”, then the new membership function represents the union of the sets se and acc. The new membership function is obtained by taking the maximum of the existing membership functions. $\mu_{se \cup acc}(x) = \max(\mu_{se}(x), \mu_{acc}(x))$.

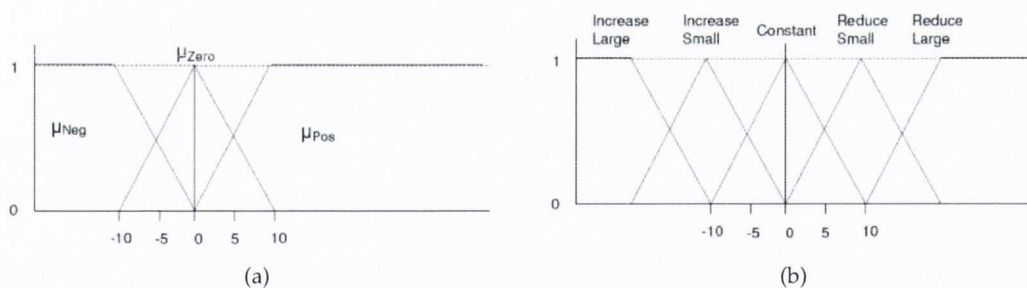


Figure B.2: Fuzzy Membership Functions: (a) Speed/Acceleration membership function; (b) Fuzzy Controller for cruise control example.

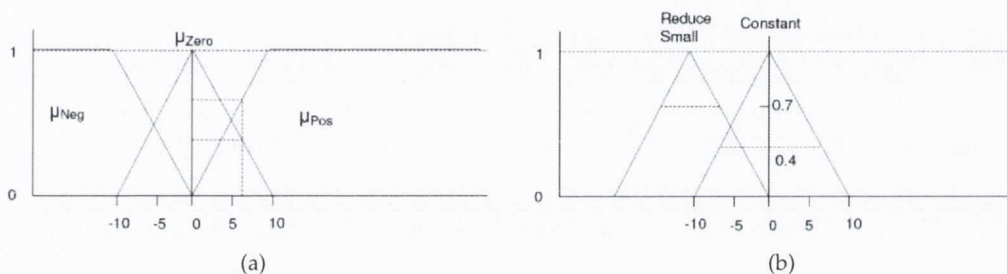


Figure B.3: Fuzzy Membership Functions: (a) Acceleration membership function with value of 6/mps ; (b) Fuzzy logic controller

Combining the member functions using the given rules yields a number of output member functions. Assume that sensor data is received which indicates a value of 0/mps for the speed error variable and also that the acceleration is 6/mps. In this case there are only two possible rules that are relevant to determining what action the fuzzy controller should take:

- If $se = Z$ and $acc = Z$ then Constant
- If $se = Z$ and $acc = P$ then Reduce Small

Thus, there are two possible outcomes or action selections:

1. Keep the system constant
2. Reduce the speed by a small amount

The degree of membership for the se variable of value 0 is equal to 0, 1 and 0 for the states Negative, Zero and Positive respectively. The degree of membership for the acc variable of value of 6 is equal to 0, 0.4 and 0.7 for the fuzzy values Negative, Zero and Positive respectively.

Defuzzification

If the degrees of membership of the scenario sets are measured and the relevant rules invoked then:

-
- the degree of membership for the Constant state is $0.4 = \min(1, 0.4)$.
 - the degree of membership for the Reduce Small state is $0.7 = \min(1, 0.7)$.

Thus there is stronger support for reducing speed by a small amount. The controller uses these two degrees of membership to produce a precise output value which will be the amount by which the speed will be reduced. The process of producing a precise quantifiable result is called defuzzification. There are numerous methods for performing defuzzification. One method of defuzzification is to calculate the centre of gravity of the areas that each output member function makes with its degree of membership (Callan 2003).

Criticism of Fuzzy Logic

Fuzzy logic has been used successfully in a number of application domains such as automatic transmissions, video cameras, electric shavers. Critics argue that the reason that these applications have been successful is because each application has a small number of rules and that these rules do not need to be chained (i.e., inferencing is not used). Arguments persist as to whether fuzzy logic and probability theory are complementary or not (Ross et al. 2002). However it is unclear how to combine fuzzy logic with decision making algorithms.



Bayesian Network for Junction Interest

```
net
{
    node_size = (80 40);
    HR_Desc = "";
    HR_OOBN_CheckConsistency = "1";
    HR_OOBN_FitExpand = "1";
    HR_Zoom_ZoomFactor = "100";
    HR_Groups_GroupColors = "";
    HR_Groups_GroupNames = "";
    HR_Groups_UserGroupsNo = "0";
    HR_Color_Interface = "21";
    HR_Color_Instance = "0";
    HR_Color_Utility = "36";
    HR_Color_Decision = "17";
    HR_Color_ContinuousChance = "48";
    HR_Color_DiscreteChance = "16";
    HR_Monitor_InitSD = "2";
    HR_Monitor_InitStates = "5";
    HR_Monitor_OpenGraph = "0";
    HR_Monitor_GraphPrecision = "100";
    HR_Monitor_AutoUpdGraph = "0";
    HR_Compile_MaxMinimalSeparators = "100000";
}
```

```

    HR.Compile_ApproxEpsilon = "1.0E-5";
    HR.Compile_Approximate = "0";
    HR.Compile_SaveToMemory = "0";
    HR.Compile_Compress = "0";
    HR.Compile_TriangMethod = "0";
    HR.Propagate_AutoNormal = "1";
    HR.Propagate_AutoSum = "1";
    HR.Propagate_Auto = "1";
    HR.Font_Italic = "0";
    HR.Font_Weight = "400";
    HR.Font_Size = "-12";
    HR.Font_Name = "javax.swing.plaf.FontUIResource[family=DejaVu
        Sans,name=DejaVu Sans,style=plain,size=12]";
    HR.Grid_GridShow = "0";
    HR.Grid_GridSnap = "0";
    HR.Grid_Y = "40";
    HR.Grid_X = "40";
}

continuous node SurfaceWater
{
    label = "SurfaceWater";
    position = (152 134);
    HR.NodeSize = "(102, 52)";
    HR.LinkMode_DegreeOfInterest = "0";
    HR.Group = "0";
    HR.Desc = "";
}

node Pedestrian
{
    label = "PedestrianPresent";
    position = (247 188);
    states = ("false" "true");
    subtype = boolean;
    HR.NodeSize = "(94, 63)";
    HR.LinkMode_DegreeOfInterest = "0";
    HR.Group = "0";
    HR.Desc = "PedestrianPresent";
    HR.State_1 = "";
    HR.State_0 = "";
}

```

```
}
```

```
node TrafficVolume
```

```
{  
    label = "TrafficVolume";  
    position = (354 148);  
    states = ("Low" "Medium" "High");  
    HR_NodeSize = "(102, 63)";  
    HR_LinkMode_DegreeOfInterest = "0";  
    HR_State_2 = "";  
    HR_Group = "0";  
    HR_Desc = "TrafficVolume";  
    HR_State_1 = "";  
    HR_State_0 = "";  
}
```

```
continuous node RainLevel
```

```
{  
    label = "RainLevel";  
    position = (91 214);  
    HR_NodeSize = "(88, 51)";  
    HR_LinkMode_SurfaceWater = "0";  
    HR_Group = "0";  
    HR_Desc = "RainLevel";  
}
```

```
continuous node JunctionInterest
```

```
{  
    label = "JunctionInterest";  
    position = (247 53);  
    HR_LinkMode_C1 = "0";  
    HR_NodeSize = "(119, 62)";  
    HR_Group = "0";  
    HR_Desc = "DegreeOfInterest";  
}
```

```
potential (SurfaceWater | RainLevel)
```

```
{  
    data = normal ( 1.2 * RainLevel, 0.25 );  
}
```

```

potential (Pedestrian)
{
    data = ( 0.5 0.5 );
}

potential (TrafficVolume)
{
    data = ( 0.2 0.4 0.4 );
}

potential (RainLevel)
{
    data = normal ( 0, 0.25 );
}

potential (JunctionInterest | Pedestrian TrafficVolume SurfaceWater
)
{
    data
    = (( normal ( 1 + 5 * SurfaceWater, 2 )    % Pedestrian=false
        TrafficVolume=Low
        normal ( 3 + 5 * SurfaceWater, 2 )    % Pedestrian=false
        TrafficVolume=Medium
        normal ( 5 + 5 * SurfaceWater, 2 ) ) % Pedestrian=false
        TrafficVolume=High
    ( normal ( 3 + 5 * SurfaceWater, 2 )    % Pedestrian=true
        TrafficVolume=Low
        normal ( 5 + 5 * SurfaceWater, 2 )    % Pedestrian=true
        TrafficVolume=Medium
        normal ( 7 + 5 * SurfaceWater, 2 ) )); %
        Pedestrian=true TrafficVolume=High
}

```

D

Decision Network for Extended Junction Controller Scenario

```
net
{
    node_size = (80 40);
}
decision Switch
{
    label = "Switch";
    position = (378 42);
    states = ("Change");
    HR_Desc = "Switch";
}
utility Reward
{
    label = "Reward";
    position = (178 44);
    HR_Desc = "Reward";
}
node TrafficDemand
{
    label = "TrafficDemand";
    position = (214 236);
```

```

    states = ("heavy" "medium" "light");
    HR_Desc = "TrafficDemand";
}
node PedestrianPresent
{
    label = "PedestrianPresent";
    position = (210 222);
    states = ("heavy" "light");
    HR_Desc = "PedestrianPresent";
}
node EmergencyVehiclePresent
{
    label = "EmergencyVehiclePresent";
    position = (300 238);
    states = ("false" "true");
    subtype = boolean;
    HR_Desc = "EmergencyVehiclePresent";
}
potential (Switch | TrafficDemand PedestrianPresent
EmergencyVehiclePresent) {}
potential (Reward | Switch TrafficDemand PedestrianPresent
EmergencyVehiclePresent)
{
    data
    = ((( (20 % Switch=Change TrafficDemand=heavy
PedestrianPresent=heavy EmergencyVehiclePresent=false
50 ) % Switch=Change TrafficDemand=heavy
PedestrianPresent=heavy EmergencyVehiclePresent=
true
( 15 % Switch=Change TrafficDemand=heavy
PedestrianPresent=light EmergencyVehiclePresent=false
45 )) % Switch=Change TrafficDemand=heavy
PedestrianPresent=light EmergencyVehiclePresent=
true
(( 15 % Switch=Change TrafficDemand=medium
PedestrianPresent=heavy EmergencyVehiclePresent=false
45 ) % Switch=Change TrafficDemand=medium
PedestrianPresent=heavy EmergencyVehiclePresent=
true
( 10 % Switch=Change TrafficDemand=medium
PedestrianPresent=light EmergencyVehiclePresent=false

```

```

40 ))          % Switch=Change  TrafficDemand=medium
    PedestrianPresent=light  EmergencyVehiclePresent=
    true
(( 11 % Switch=Change  TrafficDemand=light
    PedestrianPresent=heavy  EmergencyVehiclePresent=false
41 )          % Switch=Change  TrafficDemand=light
    PedestrianPresent=heavy  EmergencyVehiclePresent=
    true
( 6  % Switch=Change  TrafficDemand=light
    PedestrianPresent=light  EmergencyVehiclePresent=false
36 )))); % Switch=Change  TrafficDemand=light
    PedestrianPresent=light  EmergencyVehiclePresent=
    true
}
potential (TrafficDemand)
{
    data = ( 0.4 0.4 0.2 );
}
potential (PedestrianPresent)
{
    data = ( 0.5 0.5 );
}
potential (EmergencyVehiclePresent)
{
    data = ( 0.1 0.9 );
}

```



System State XSD Schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="systemState">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:integer" />
        <xsd:element name="name" type="xsd:string" />
        <xsd:element name="description" type="xsd:string" />
        <xsd:element name="properties">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="complexity" type="xsd:boolean" />
              <xsd:element name="dynamism" type="xsd:boolean" />
              <xsd:element name="observability" type="xsd:string" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="scope">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="type">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="element"/>
              <xsd:enumeration value="multi-element"/>
              <xsd:enumeration value="region"/>
              <xsd:enumeration value="null" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="number" type="xsd:integer" minOccurs="0" maxOccurs="
    unbounded" />
  <xsd:element name="value" type="xsd:string" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="location" type="xsd:string" minOccurs="0" maxOccurs="
  unbounded" />
<!-- Note the inputs etc are defined in scope -->
<xsd:element name="inputs">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="layer1" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
              "unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="layer2" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
              "unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="layer3" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs=
              "unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="implementation">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="code" type="xsd:string" />
      <xsd:element name="sourceRef" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:sequence>

```

```
</xsd:complexType>  
</xsd:element>  
</xsd:schema
```

Listing E.1: XML Schema for System-States specification

Bibliography

- Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Sterling, König, D., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P. & Yiu, A. (2006), 'Web services business process execution language version 2.0', OASIS Committee Draft.
- Andersen, S. K., Olesen, K. G. & Jensen, F. V. (1990), *HUGIN - a shell for building Bayesian belief universes for expert systems*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 332–337.
- Anneke, K., Jos, W. & Wim, B. (2003), *MDA Explained - The Model Driven Architecture: Practice and Promise*, Addison-Wesley.
- Atkinson, C. & Kuhne, T. (2003), 'Model-driven development: a metamodeling foundation', *Software, IEEE* **20**(5), 36–41.
- Bacchus, F. (2001), 'The aips'00 planning competition', *AI Magazine*, 22(1):47-56.
- Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J. & Zukowski, D. (2000), Challenges: an application model for pervasive computing, in 'MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking', ACM, New York, NY, USA, pp. 266–274.
- Banker, R. D., Datar, S. M. & Zweig, D. (1989), Software complexity and maintainability, in 'ICIS '89: Proceedings of the tenth international conference on Information Systems', ACM, New York, NY, USA, pp. 247–255.
- Baresi, L. & et al (2006), Towards a model-driven approach to develop applications based on physical active objects, in 'APSEC '06', IEEE Computer Society.
- Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T. V. & Reifenrath, K. (2009), 'State chart xml: State machine notation for control abstraction', W3C Working Draft.
URL: <http://www.w3.org/TR/scxml/>
- Bauer, M., Becker, C. & Rothermel, K. (2002), 'Location models from the perspective of context-aware applications and mobile ad hoc networks', *Personal Ubiquitous Comput.* **6**(5-6), 322–328.

BIBLIOGRAPHY

- Beazley, D. M. (1996), Swig: an easy to use tool for integrating scripting languages with c and c++, in 'Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4', USENIX Association, Berkeley, CA, USA, pp. 15-15.
URL: <http://portal.acm.org/citation.cfm?id=1267498.1267513>
- Beazley, D. M. & Lomdahl, P. S. (1997), Feeding a large-scale physics application to python, in 'In 6th International Python Conference', pp. 21-28.
- Behnel, S. (2009), 'Lxml- version 2.0', <http://codespeak.net/lxml>.
- Bellifemine, F. L., Caire, G. & Greenwood, D. (2007), *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*, Wiley.
- Bellman, R. (1961), *Adaptive Control Processes: A Guided Tour*, Princeton University Press.
- Bertsekas, D. P. & Tsitsiklis, J. N. (2002), *Introduction to Probability*, Athena Scientific.
- Bonet, B. & Geffner, H. (2001), GPT: a tool for planning with uncertainty and partial information, in A. Cimatti, H. Geffner, E. Giunchiglia & J. Rintanen, eds, 'Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information', Seattle, WA, pp. 82-87.
- Bonet, B. & Geffner, H. (2005), 'mgpt: A probabilistic planner based on heuristic search', *Journal of Artificial Intelligence Research* **24**, 24-933.
- Bratman, M. (1987), *Intentions, Plans, and Practical Reason*, Harvard University Press.
- Brooks, F. (1987), 'No silver bullet', *IEEE Computer* **20**(4), 10-19.
- Callan, R. (2003), *"Artificial Intelligence"*, Palgrave Macmillan.
- Cassandra, A. R., Kaelbling, L. P. & Littman, M. L. (1994), Acting optimally in partially observable stochastic domains, in 'AAAI- 1994'.
URL: citeseer.ist.psu.edu/cassandra94acting.html
- Chan, H. & Darwiche, A. (2005), 'On the revision of probabilistic beliefs using uncertain evidence', *Artif. Intell.* **163**(1), 67-90.
- Chen, H., Finin, T. & Joshi, A. (2003), 'An ontology for context-aware pervasive computing environments', *Knowl. Eng. Rev.* **18**(3), 197-207.
- Cowan, G. (1998), *Statistical Data Analysis*, Clarendon (Oxford), Oxford.
- Dargie, W. (2007), The role of probabilistic schemes in multisensor context-awareness, in 'PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops', IEEE Computer Society, Washington, DC, USA, pp. 27-32.

BIBLIOGRAPHY

- Dasarathy, B. V. (1994), *Decision Fusion*, IEEE Computer Society Press, Los Alamitos, CA, USA.
- Davis, J. S. & LeBlanc, R. J. (1988), 'A study of the applicability of complexity measures', *IEEE Trans. Softw. Eng.* **14**(9), 1366–1372.
- de Jong, I. (2009), 'Pyro – version 3.10', <http://www.pyro.sourceforge.net>.
- Dean, M. & Schreiber, G. (2004), OWL web ontology language reference, W3C recommendation, W3C.
- Dey, A. K. (2001), 'Understanding and using context', *Personal Ubiquitous Comput.* **5**(1), 4–7.
- Dey, A. K., Abowd, G. D. & Salber, D. (2001), 'A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications', *Hum.-Comput. Interact.* **16**(2), 97–166.
- Dowling, J., Cunningham, R., Curran, E., Harrington, A. & Cahill, V. (2005), Emergent consensus in decentralised systems using collaborative reinforcement learning, in 'Self-Star Workshop'.
- D'Souza, D. F. & Wills, A. C. (1999), *Objects, components, and frameworks with UML: the catalysis approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Durrant-Whyte, H. F. (1988), 'Sensor models and multisensor integration', *Journal of Robotics Research* **7**(6).
- Fellendorf, M. (1997), 'Public transport priority within scats - a simulation case study in dublin', Presented at the 67th Annual Meeting of the Institute of Transport Engineers.
- Fenton, N. E. & Neil, M. (2000), Software metrics: roadmap, in 'ICSE '00: Proceedings of the Conference on The Future of Software Engineering', ACM, New York, NY, USA, pp. 357–370.
- Fernandes, J. E., Machado, R. J. & Carvalho, J. A. (2008), Model-driven software development for pervasive information systems, in 'Advances in Ubiquitous Computing: Future Paradigms and Directions', IGI Publishing, pp. 44–82.
- Fox, M. & Long, D. (2002), 'International planning competition'.
URL: <http://www.dur.ac.uk/d.p.long/competition.html>
- France, R. & Rumpe, B. (2007), Model-driven development of complex software: A research roadmap, in 'FOSE '07: 2007 Future of Software Engineering', IEEE Computer Society, Washington, DC, USA, pp. 37–54.
- Franken, D. & Hupper, A. (2005), Improved fast covariance intersection for distributed data fusion, in '8th International Conference on Information Fusion', Vol. 1.

BIBLIOGRAPHY

- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns*, Addison-Wesley Professional.
- Ganek, A. G. & Corbi, T. A. (2003), 'The dawning of the autonomic computing era', *IBM Syst. J.* **42**(1), 5–18.
- Gellersen, H. W., Schmidt, A. & Beigl, M. (2002), 'Multi-sensor context-awareness in mobile devices and smart artifacts', *Mob. Netw. Appl.* **7**(5), 341–351.
- Ghallab, M., Nationale, E., Aeronautiques, C., Isi, C. K., Golden, K., Penberthy, S., Smith, D. E., Sun, Y., Weld, D. & Mcdermott, C. D. (1998), Pddl - the planning domain definition language, version 1.2, Technical report.
- Ghallab, M., Nau, D. & Traverso, P. (2004), *Automated Planning*, Morgan Kaufmann.
- Gill, G. K. & Kemerer, C. F. (1991), 'Cyclomatic complexity density and software maintenance productivity', *IEEE Trans. Softw. Eng.* **17**(12), 1284–1288.
- Gu, T., Pung, H. K. & Zhang, D. Q. (2004), A bayesian approach for dealing with uncertain contexts, in 'Proceedings of the 2nd international conference on Pervasive computing', Austrian Computer Society.
- Guestrin, C., Koller, D., Parr, R. & Venkataraman, S. (2003), 'Efficient solution algorithms for factored mdps', *J. Artif. Intell. Res. (JAIR)* **19**, 399–468.
- Harrington, A. & Cahill, V. (2004), Route profiling: putting context to work, in 'Proceedings of the 2004 ACM symposium on Applied computing', SAC '04, ACM, New York, NY, USA, pp. 1567–1573.
URL: <http://doi.acm.org/10.1145/967900.968214>
- Harrington, A. & Cahill, V. (2007), 'Domain modelling for ubiquitous computing applications', *Advanced Information Networking and Applications Workshops, International Conference on* **2**, 326–333.
- Henderson, T. & Shilcrat, E. (1984), 'Logical sensor systems', *Journal of Robotic Systems* **1**(2), 169–193.
- Hromkovič, J. (2001), *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics*, Springer-Verlag New York, Inc., New York, NY, USA.
- IEEE (2007), 'Ieee standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (teds) formats', *IEEE Std 1451.5-2007* pp. C1 –236.
- Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P. & Venkatramani, C. (2006), Design, implementation, and evaluation of the linear road bnchmark on the stream

BIBLIOGRAPHY

- processing core, in 'SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data', ACM, New York, NY, USA, pp. 431–442.
- Jensen, F. (2007), 'Hugin api reference manual'.
URL: <http://www.hugin.com/developer/documentation/api-manuals>
- Jensen, F. & Nielsen, T. (2007), *"Bayesian Networks and Decision Graphs"*, Springer Verlag.
- Jensen, K. (1992), *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Springer.
- Jordan, M. I. (1999), *Learning in graphical models*, The MIT Press.
- Julier, S. J. & Uhlmann, J. (2001), General decentralized data fusion with covariance intersection(ci), in D. L. Hall & J. Llinas, eds, 'Handbook of Multisensor Data Fusion', CRC Press.
- Julier, S. J. & Uhlmann, J. K. (2007), 'Using covariance intersection for slam', *Robot. Auton. Syst.* **55**(1), 3–20.
- Kautz, H. A. & Selman, B. (1999), Unifying sat-based and graph-based planning, in 'IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 318–325.
- Kemerer, C. F. (1995), 'Software complexity and software maintenance: A survey of empirical research', *Ann. Software Eng.* **1**, 1–22.
- Kephart, J. O. & Chess, D. M. (2003), 'The vision of autonomic computing', *Computer* **36**(1), 41–50.
- Khoshgoftaar, T. & Munson, J. (1995), 'Predicting software development errors using software complexity metrics', *Selected Areas In Communications* **2**(8), 253–261.
- Korb, K. & Nicholson, A. (2004), *"Bayesian Artificial Intelligence"*, Chapman and Hall.
- Li, J., Wenkai, L., Karimi, I. & Srinivasan, R. (2006), 'A robust and efficient algorithm for optimizing crude oil operations', *AIChE Chemistry Journal* .
- Luo, R. & Kay, M. (1989), "'Multisensor Integration and Fusion in Intelligent Systems'", *IEEE Transactions on Systems, Man, and Cybernetics* **19**(5).
- Lutz, M. (2007), *Learning Python*, O'Reilly & Associates, Inc., Sebastopol, CA, USA.
- Ma, R., O'Hare, G. & O'Grady, M. (2006), 'Agent factory micro edition: A framework for ambient applications', In *Proceedings of: Intelligent Agents in Computing Systems - the workshop in frames of the ICCS 2006 (International Conference on Computational Science)* .

BIBLIOGRAPHY

- Machado, R. J., Lassen, K. B., Oliveira, S., Couto, M. & Pinto, P. (2007), 'Requirements validation: Execution of uml models with cpn tools', *Int. J. Softw. Tools Technol. Transf.* **9**(3), 353–369.
- Madani, O., Hanks, S. & Condon, A. (1999), On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems, in 'AAAI/IAAI', pp. 541–548.
URL: citeseer.ist.psu.edu/madani99undecidability.html
- Mathiassen, L. & Stage, J. (1990), Complexity and uncertainty in software design, in 'In CompEuro'90, Proceedings of the 1990 International Conference on Computer Systems and Software Engineering', pp. 482–489.
- Maybeck, P. (1979), *"Stochastic Models, Estimation and Control"*, Academic Press.
- McCabe, T. J. (1976), 'A complexity measure', *IEEE Trans. Software Eng.* **2**(4), 308–320.
- McGuinness, D. L., Fikes, R., Hendler, J. & Stein, L. A. (2002), 'Daml+oil: An ontology language for the semantic web', *IEEE Intelligent Systems* **17**(5), 72–80.
- Meier, R., Harrington, A. & Cahill, V. (2005), A framework for integrating existing and novel intelligent transportation systems, in 'IEEE ITSC'.
URL: <http://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-45.pdf>
- Meier, R., Harrington, A., Termin, T. & Cahill, V. (2006), A spatial programming model for real global smart space applications, in 'IFIP DAIS', LNCS.
URL: <http://www.cs.tcd.ie/publications/tech-reports/reports.06/TCD-CS-2006-18.pdf>
- Mellor, S. J., Clark, A. N. & Futagami, T. (2003), 'Guest editors introduction: Model-driven development', *IEEE Software* **20**(5), 14–18.
- Mernik, M., Heering, J. & Sloane, A. M. (2005), 'When and how to develop domain-specific languages', *ACM Comput. Surv.* **37**(4), 316–344.
- Misra, S. & Cafer, F. (2010), A software metric for python language, in 'Computational Science and Its Applications', LNCS.
- Mitchell, H. (2007), *Multi-Sensor Data Fusion*, Springer.
- Munnelly, J. (2010), ALPH: A DSAL-based Programming Model for Complexity Management in Pervasive Healthcare Applications, PhD thesis, School of Computer Science and Statistics.
- Murphy, K. (2005), 'A survey of pomdp solution techniques - technical report university of british columbia'.

BIBLIOGRAPHY

- Muñoz, J. & Pelechano, V. (2006), Applying software factories to pervasive systems: A platform specific framework, in '8th International Conference on Enterprise Information Systems (ICEIS 2006), Paphos (Cyprus) 23 - 27, May 2006. pags: 337', p. 342.
- Nasr, E. (2002), A technique for managing complexity of use cases for large complex embedded systems, in 'ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing', IEEE Computer Society, Washington, DC, USA, p. 225.
- Nau, D. & Ghallab, M. (2004), 'Measuring the performance of automated planning systems'.
- Newmarch, J. (2000), *A programmer's guide to Jini technology*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Nicklas, D., Grossmann, M. & Schwarz, T. (2003), Nexusscout: an advanced location-based application on a distributed, open mediation platform, in 'VLDB '2003: Proceedings of the 29th international conference on Very large data bases', VLDB Endowment, pp. 1089–1092.
- Obe, R. & Hsu, L. (2010), *PostGIS in Action*, Manning Publications.
- O'Hare, G. M., O'Grady, M. J., Tynan, R., Muldoon, C., Kolar, H. R., Ruzzelli, A. G., Diamond, D. & Sweeney, E. (2007), 'Embedding intelligent decision making within complex dynamic environments', *Artif. Intell. Rev.* 27(2-3), 189–201.
- OOPSLA (2007), "'no silver bullet reloaded retrospective oopsla panel summary'".
URL: <http://www.infoq.com/articles/No-Silver-Bullet-Summary>
- OpenGeoSpatialConsortium (1999), 'Opengis simple features specification for sql'.
URL: <http://www.opengespatial.org>
- Pan, R., Peng, Y. & Ding, Z. (2006), Belief update in bayesian networks using uncertain evidence, in 'ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence', IEEE Computer Society, Washington, DC, USA, pp. 441–444.
- Papasimeon, M. & Heinze, C. (2001), Extending the uml for designing jack agents, in 'ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering', IEEE Computer Society, Washington, DC, USA, p. 89.
- Paquet, S. (2006), Distributed Decision-Making and Task Coordination in Dynamic, Uncertain and Real-Time Multiagent Environments, PhD thesis, Universite Laval Quebec.
- Paquet, S., Tobin, L. & Chaib-draa, B. (2005), An online pomdp algorithm for complex multiagent environments, in 'AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems', ACM, New York, NY, USA, pp. 970–977.

BIBLIOGRAPHY

- Pearl, J. (1986), Jeffrey's rule and the problem of autonomous inference agents- technical report ucla csd-860099.
- Pearl, J. (1990), Jeffrey's rule, passage of experience, and *neo*-Bayesianism, in H. e. a. Kyburg, Jr., ed., 'Knowledge Representation and Defeasible Reasoning', Kluwer Academic Publishers, pp. 245–265.
- Ranganathan, A. (2004), Autonomic pervasive computing based on planning, in 'ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)', IEEE Computer Society, Washington, DC, USA, pp. 80–87.
- Ranganathan, A. & Campbell, R. H. (2003), A middleware for context-aware agents in ubiquitous computing environments, in 'Middleware '03: Proceedings of the ACM/I-FIP/USENIX 2003 International Conference on Middleware', Springer-Verlag New York, Inc., New York, NY, USA, pp. 143–161.
- Ranganathan, A. & Campbell, R. H. (2005a), Olympus: A high-level programming model for pervasive computing environments, in 'IEEE Pervasive Computing'.
- Ranganathan, A. & Campbell, R. H. (2005b), Supporting tasks in a programmable smart home, in 'ICOST 2005 : 3rd International Conference On Smart homes and health Telematic', University of Illinois at Urbana-Champaign.
- Robin, A. & Botts, M. (2006), 'White paper - creation of specific sensorml process models'.
- Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H. & Nahrstedt, K. (2002), 'A middleware infrastructure for active spaces', *IEEE Pervasive Computing* 1(4), 74–83.
- Ross, S., Pineau, J., Paquet, S. & Chaib-Draa, B. (2008), 'Online planning algorithms for pomdps', *Journal of Artificial Intelligence Research* 32, 663–704.
- Ross, T. J., Sellers, K. F. & Booker, J. M. (2002), 'Considerations for using fuzzy set theory and probability theory', pp. 87–104.
- Rudich Steven, Wigderson Avi, e. (2004), *Computational Complexity Theory*, IAS Park City Mathematics Institute.
- Russell, S. & Norvig, P. (2003), *Artificial Intelligence - A Modern Approach*, 2 edn, Prentice Hall.
- Selic, B. (2003), 'The pragmatics of model-driven development', *IEEE Softw.* 20(5), 19–25.
- Senart, A., Cunningham, R., Bouroche, M., O'Connor, N., Reynolds, V. & Cahill, V. (2006), MoCoA: Customisable middleware for context-aware mobile applications, in '8th International Symposium on Distributed Objects and Applications (DOA 2006)'.

BIBLIOGRAPHY

- Senart, A., Harrington, A., Nedos, A. & Cahill, V. (2009), 'Design environment for sensor network environments', <http://www.dsg.cs.tcd.ie/local/desna>.
- Sentz, K. & Ferson, S. (2006), Combination of evidence in dempster-shafer, Technical reports, Los Alamos National Laboratory.
URL: <http://www.sandia.gov/epistemic/Reports/SAND2002-0835.pdf>
- Serral, E., Valderas, P. & Pelechano, V. (2008), A model driven development method for developing context-aware pervasive systems, in 'UIC '08: Proceedings of the 5th international conference on Ubiquitous Intelligence and Computing', Springer-Verlag, Berlin, Heidelberg, pp. 662–676.
- Shepperd, M. (1988), 'A critique of cyclomatic complexity as a software metric', *Softw. Eng. J.* 3(2), 30–36.
- Sondik, E. J. (1971), the Optimal Control of Partially Observable Markov Decision Processes, PhD thesis, University of Stanford.
- Sooriamurthi, R. (2009), Introducing abstraction and decomposition to novice programmers, in 'ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education', ACM, New York, NY, USA, pp. 196–200.
- Spall, J. (2003), *Introduction to Stochastic Search and Optimization*, Wiley InterScience.
- Stuart, K. (1995), *"At home in the Universe- The search for the Laws of Complexity"*, Oxford University Press.
- Suman, B. & Kumar, P. (2006), 'A survey of simulated annealing as a tool for single and multiobjective optimization', *Journal of the Operational Research Society* (57), 1143–1160.
- Suppakitnarm, A., Seffen, K., Parks, G. & Clarkson, P. (2000), 'A simulated annealing algorithm for multiobjective optimization', *Engineering Optimization* 33(1), 59–85.
- Tarr, P., Ossher, H., Harrison, W. & Sutton, Jr., S. M. (1999), N degrees of separation: multi-dimensional separation of concerns, in 'ICSE '99: Proceedings of the 21st international conference on Software engineering', ACM, New York, NY, USA, pp. 107–119.
- Tegarden, D. P., Sheetz, S. D. & Monarchi, D. E. (1995), 'A software complexity model of object-oriented systems', *Decis. Support Syst.* 13(3-4), 241–262.
- Tian, J. & Zelkowitz, M. V. (1995), 'Complexity measure evaluation and selection', *IEEE Trans. Softw. Eng.* 21(8), 641–650.
- Uhlmann, J. (2003), 'Covariance consistency methods for fault-tolerant distributed data fusion', *Information Fusion* 4(3), 201–215.
URL: [http://dx.doi.org/10.1016/S1566-2535\(03\)00036-8](http://dx.doi.org/10.1016/S1566-2535(03)00036-8)

BIBLIOGRAPHY

- van Deursen, A., Klint, P. & Visser, J. (2000), 'Domain-specific languages: an annotated bibliography', *SIGPLAN Not.* **35**(6), 26–36.
- Vaquero, T. S., Romero, V., Tonidandel, F. & Silva, J. R. (2007), itsimple 2.0: An integrated tool for designing planning domains, in 'ICAPS', pp. 336–343.
- Vaquero, T. S., Sette, F., dos Santos, E. A. & Silva, J. R. (2008), 'Integrated tools environment for modelling and analysis of automated planning systems', *ABCM Symposium Series in Mechatronics* **3**, 434–443.
- Vaquero, T. S., Tonidandel, F., de Barros, L. N. & Silva, J. R. (2006), On the use of uml.p for modeling a real application as a planning problem, in 'ICAPS', pp. 434–437.
- Vickers, P. (2003), 'An introduction to function point analysis by dr paul vickers'.
- Wang, P. (2004), 'The limitation of bayesianism', *Artif. Intell.* **158**(1), 97–106.
- Weinberger, E. D. (1996), Np completeness of kauffman's n-k model, a tuneable rugged fitness landscape, Working Papers 96-02-003, Santa Fe Institute.
- Weld, D. S., Anderson, C. R. & Smith, D. E. (1998), Extending graphplan to handle uncertainty and sensing actions, in 'AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence', American Association for Artificial Intelligence, Menlo Park, CA, USA, pp. 897–904.
- Williams, T. & Kelley, C. (2010), 'gnuplot – version 4.4', <http://www.gnuplot.info>.
- Wolpert, D. H. & Macready, W. G. (1997), 'No free lunch theorems for optimization', *IEEE Transactions on Evolutionary Computation* **1**(1), 67–82.
- Wu, H. (2003), Sensor data Fusion for Context-Aware Computing Using Dempster-Shafer Theory, PhD thesis, The Robotics Institute, Carnegie Mellon University.
- Wu, H., Siegel, M., (contact, M. S., Yang, J. & Stiefelwagen, R. (2002), Sensor fusion using dempster-shafer theory, in 'in Proceedings of IEEE Instrumentation and Measurement Technology Conference', pp. 21–23.
- Xiao, T. G., Wang, X. H., Pung, H. K. & Zhang, D. Q. (2004), An ontology-based context model in intelligent environments, in 'In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference', pp. 270–275.
- Ye, J., McKeever, S., Coyle, L., Neely, S. & Dobson, S. (2008), Resolving uncertainty in context integration and abstraction: context integration and abstraction, in 'ICPS '08: Proceedings of the 5th international conference on Pervasive services', ACM, New York, NY, USA, pp. 131–140.