



Perception as Stochastic Grammar-based Sampling on Dynamic Graph Spaces

Luis J. Manso

Cáceres, 2013

University of Extremadura

Doctoral Dissertation

This work is licensed under license:

[Creative Commons Attribution-Noncommercial-No Derivative Works 3.0.](https://creativecommons.org/licenses/by-nc-nd/3.0/)



Abstract:

One of the current main goals of robotics is to develop robots able to robustly achieve useful tasks in non-trivial environments. Most of these tasks do not only require robots to be able to model and interact with the objects in the environment but also to interact with humans (communicating and accepting commands from them). To make robots able to accomplish these tasks they must be capable of perceiving the environment autonomously and efficiently, so that the representations generated are useful to achieve the missions they are required to perform.

Since some tasks can not be achieved in a pure reactive manner it is necessary that the robots generate some sort of environment representation. Some moderately simple tasks such as local and global navigation can be achieved using relatively simple structures such as two-dimensional occupancy grids. However, when the robot is supposed to perform complex tasks it is necessary to use considerably richer and structured world models, with a higher semantic load than a two-dimensional grid.

As robotics advances, it becomes more plausible to find robots working in real, non-engineered environments, where shadows, textureless areas, clutter and obstacles represent real challenges not only to perceive the world but also to perform actions. Despite that these environments are more complex and less predictable than engineered ones, they are not composed of random shapes nor change without a reason. To build actual intelligent robots it is necessary to seize all the *ecological* information available. This information is even more important in indoor environments, where the rooms and objects within them are mostly composed of small sets of geometric shapes and, in many cases, have strong structural relationships —*e.g.*, of composition, where an element of the environment is always composed of others; of relative position, where an object is always located within or besides other specific type of object. Thus, it would be desirable to have a technology that allows to build robots able to model and interact with their environment using all the a priori information available about the environment in an efficient way, so that robots can interact with it appropriately.

The goal of this work is the study and development of novel techniques that can be used to cover these needs. The contributions of the thesis can be classified in three blocks: **a)** software engineering for robotics; **b)** active perception and modeling; and **c)** a set of experiments regarding indoor modeling. Specifically, most of the thesis focuses on the second and third of these blocks.

Regarding software engineering for robotics, all the contributions have been integrated in RoboComp, a robotics framework. RoboComp stands out for being component-oriented, for the set of tools and libraries it provides and for making extensive use of domain-specific languages (DSL) that facilitate and accelerate the development of software for robots. Among the developed tools it is worth mentioning RCManager (a graphical manager of component networks), RCMonitor (a monitoring and visualizing tool for components), RCControlPanel (a tool to simultaneously access and control several components) and RCInnerModelSimulator (a robotics simulator). InnerModel and InnerModelViewer are also of particular interest. They are a pair of classes designed to represent, manage and visualize kinematic trees. The kinematic trees used by these classes are described using IMDSL, a domain-specific language used to describe this kind of structures.

In the field of active perception it is presented *Active Grammar-based Modeling* (AGM),

the main contribution of the thesis. Active Grammar-based Modeling is based on the fact that the changes that take place in the world models that robots generate and use can be described using formal grammars. In AGM, grammar rules are decorated with the configuration of the different physical and perceptual behaviors that a specific robot might have —being the latter formulated as stochastic sampling processes. These grammars can then be translated to classic AI planning languages and provide solutions to different perceptual phenomena: bottom-up parsing, action selection (both for perception and mission accomplishment), covert perception and the inclusion of context-dependent perceptual restrictions.

AGM proposes a distributed approach where several agents interact according to a grammar that describes how world models can be created and modified. Such grammar is used to guide robot control and perception, making it interact with the environment and modify the world model as a result of the interaction between all agents. A central node is in charge of coordinating the agents and managing the changes that they propose (allowing only those made according to the grammar). Both tasks are based on the use of a classic AI planner, which is the element that takes the grammar as input (after being translated to a planning language). It generate plans and verify that the changes are the result of applying the rules of the grammar to the current model.

The rest of the contributions of the thesis are specific techniques related to indoor environment modeling and are presented as experiments in real environments. While the first experiments use simple representations with a fixed structure, the latest ones use the graph-based representations of unknown size and structure that are supported in this thesis. Specifically, the last two experiments show how AGM can be used to actively perceive the environment, not only the rooms but also the objects within them.

Resumen:

Uno de los principales retos de la robótica en la actualidad es conseguir que los robots realicen robustamente tareas útiles en entornos no triviales. Para la gran mayoría de tareas es necesario no sólo que sean capaces de modelar los objetos que haya en el entorno e interactuar con ellos sino también con humanos (comunicándose con ellos y aceptando órdenes). Todo esto implica que han de ser capaces de percibir el entorno autónoma y eficientemente, de tal forma que les sirva para poder completar exitosamente las misiones que se les encomienden.

Para poder realizar tareas que no puedan ser completadas de una forma meramente reactiva es necesario que los robots generen algún tipo de representación del entorno. Para tareas simples como la navegación local o global estas estructuras pueden ser extremadamente simples, como mapas de rejillas bidimensionales. Sin embargo, cuando se espera que los robots desarrollen tareas complejas se hace necesario que estos modelos sean considerablemente ricos y estructurados, con una carga semántica mayor que una simple rejilla de ocupación.

Con los avances de la robótica, es cada vez más viable que los robots habiten entornos reales no controlados, donde las sombras, zonas de baja textura, oclusiones y obstáculos representan importantes dificultades tanto para percibir el mundo como para realizar acciones sobre él. A pesar de que estos entornos son más complejos y menos predecibles que un entorno controlado, ni están compuestos de formas aleatorias ni cambian sin razón. Para construir robots verdaderamente inteligentes, es necesario que estos usen toda la información *ecológica* disponible. Esta información es aun más importante en entornos de interior, donde las habitaciones y los objetos que en ellas se pueden encontrar están en su mayor parte sujetos a pequeños conjuntos de formas geométricas y, en muchos casos, mantienen relaciones estructurales fuertes —*e.g.*, de composición, donde un elemento del entorno está siempre compuesto por otros; de localización, donde un elemento siempre se encuentra dentro o al lado de otro. Por tanto, es preciso disponer de la tecnología que nos permita crear robots que modelen e interaccionen con su entorno usando la información a priori de la que se disponga sobre el entorno de una forma eficiente, que permita a los robots actuar con él de una forma apropiada.

El objetivo de este trabajo es el estudio y desarrollo de técnicas novedosas que puedan ser usadas para suplir estas necesidades. Las aportaciones de la tesis se pueden clasificar en tres bloques: **a)** ingeniería de software orientada a la robótica; **b)** modelado y percepción activa; y **c)** experimentos. En concreto, el grueso de la tesis se centra en el segundo y tercero de los bloques.

Dentro de la ingeniería de software orientada a la robótica destaca una parte importante del desarrollo de RoboComp, un framework de robótica. RoboComp destaca por ser orientado a componentes, por el conjunto de herramientas y librerías que lo forman, y por hacer un uso extensivo de lenguajes específicos de dominio (DSL) que facilitan y aceleran el desarrollo de software para robots. Entre las herramientas desarrolladas destacan RCManager (un gestor gráfico de redes de componentes), RCMonitor (un monitorizador y un replicador de la salida de componentes), RCControlPanel (una herramienta que sirve para acceder y controlar simultáneamente varios componentes de robots) y RCInnerModelSimulator (un simulador de robótica). También son dignos de mención InnerModel e InnerModelViewer, un par de clases diseñadas para representar, manejar y visualizar árboles cinemáticos; y IMDSL, un lenguaje específico de dominio usado para describir estos árboles.

En el campo de la percepción activa se presenta *Active Grammar-based Modeling* (AGM), una arquitectura que representa la principal contribución de la tesis. Active Grammar-based Modeling se basa en el hecho de que los cambios que tienen lugar en los modelos del mundo que generan y usan los robots pueden ser descritos dentro de una gramática formal con ciertas características especiales. Estas gramáticas, junto con la asociación de las reglas que las componen con diferentes configuraciones de sus comportamientos físicos y perceptivos — siendo estos últimos planteados como procesos de muestreo estocástico— aportan soluciones a distintos problemas o fenómenos perceptivos: la generación de modelos, la selección de acción (tanto para la percepción como para cumplir un objetivo determinado), la percepción encubierta y la inclusión de restricciones perceptivas dependientes del contexto.

AGM propone un planteamiento distribuido donde varios agentes interactúan de acuerdo a la gramática que rige el control del robot, haciendo que éste interactúe con el entorno y modificando entre todos el modelo del mundo. Un nodo central se encarga de gestionar los cambios (permitiendo únicamente aquellos que no violan la gramática) y coordinar al resto de agentes. Ambas tareas se basan en el uso de un planificador, que es el que realmente trabaja con la gramática, que genera planes y verifica que las modificaciones son el resultado de aplicar las reglas de la gramática al modelo previo.

El resto de las aportaciones de la tesis, técnicas específicas relacionadas con el modelado de entornos de interior, se presentan en forma de experimentos en entornos reales, inicialmente usando representaciones simples de características conocidas y finalmente usando las representaciones en grafos de tamaño variable que se proponen dentro de Active Grammar-based Modeling. En particular, se presentan dos experimentos que muestran cómo AGM puede ser usado para percibir activamente el entorno, tanto la habitación en sí como los objetos que están dentro de ella.

Resumin:

Unu delos prencipalis afanis dela robótica ogañu es precural que los robonis hagan con reziúra tareas aparentis en ambientis no insussus. Pala gran huerça de tareas es precisu de que no sean escapás de modelal i entreactual conos ojetus namás, sino tamién con umanus (comunicandu-si i acetandu mandaus). Tó estu emprica que tien de sel escapás de percebil el entornu autónoma i eficientimenti de má que le valga pa poel de cumpril con déssitu las missionis que se l'encomiendin.

Pa poel de hazel tareas que no puean sel cumprías duna manera meramenti reativa, es mestel que los robonis generin alguna crassi de representación del entornu. Pa tareas simpris comu el navegaeru local o general estas estruturas puein sel tó pol tó simpris, comu mapas de hiendas bidimensionalis. Sin embargo, de ná que s'espera que los robonis cumbran tareas complexas, es precisu que estus modelus sean considerablimenti ricus i estructuraus, con una carga semántica mayol que una hiendina d'ocupamientu.

Conos avancis dela robótica, ca ves es más fetivu los robonis abital ambientis realis no barajaus, ondi las sombras, partis de testura baxa, atacuñonis i estáculos presentan atravancamientus importantis lo mesmu pa percebil el mundu comu pa obral en él. Enque estus entornus son más complexus i menus prediziblis que unu barajau, ni están compuestus de hormas aleatorias ni múan sin razón. Pa componel robonis verdaeramenti enteligenis, es precisu d'ellus gastal de tola información ambiental disponibli. Esta información velaquí es entovía más importanti en entornus d'interiol, andi las abitacionis i los ojetus que en ellas se puein encontral están pendienteis de congullinus de hormas geométricas i, muchas vezis, tienin relacionis estruturalis huertis —p.e., de composición, ondi un elementu del entornu está siempre compuestu por otrus; de sitiu, ondi un elementu siempre s'encuentra endrentu o la vera d'otru. De mó que es mestel disponel dela tecnología que mos permita crial robonis que modelin i entreactúin con el entornu usando dela enformación a priori dela que se disponga duna manera eficienti, que permita alos robonis portal-si propiamenti.

L'objetivu desti trebaju es el estudiaeru i desenroamientu de técnicas novedosas que puean gastal-si pa supril estas necessiais. Las aportacionis dela tesi puein acolocal-si en tres apartijus: a) ingeniería de pogramagi canteá ala robótica; b) modelau i apercebimientu ativu; i c) esprimentus. En concretu, la huerça dela tesis se centra nel segundu i tercel apartijus.

Endrentu dela ingeniería de pogramagi canteá ala robótica se destaca una parti importanti del desenroamientu de RoboComp, un framework de robótica. RoboComp se destaca por estal canteau a componentis, pol herramienteu i librerías que lo componin, i por gastal destensamenti de lenguagis específicus de domiñu (DSL) que facilitan i aceleran el desenroamientu del pogramagi pa robonis.

Entri las herramientas desenroás se destacan RCManager (un gestol gráficu de redis de componentis), RCMonitor (un monitoreaol i un repricaol dela salida de componentis), RCControlPanel (una herramienta que val pa acedel i manejar ala par varius componentis robóticos) i RCInnerModelSimulator (un simulaol de robótica). Tamién son dinus de mentación InnerModel i InnerModelViewer, un par de crassis pensás pa repressental, manejar i visoreal arvus cinemáticus; i IMDSL, un lenguagi específicu de domiñu usau pa descrevil estus arvus.

Nel campu del apercebimientu ativu se presenta Active Grammar-based Modeling (AGM), una arquitetura que es la prencipal contribución dela tesi. Active Grammar-based Modeling se basa en que las muacionis que acontecin enos modelus del mundu que generan i gastan

los robonis puein sel descritos endrentu duna gramática horma con ciertas señas especialis. Estas gramáticas, junta l'associación delas regras que las componin con deferentis ahormacionis delos sus comportamientus físicus i percetivus —en huendu estus acabarus pranteaus comu processus de muestreu estocásticu— aportan solucionis a destintus pobremas o fenómenos percetivus: la generación de modelus, la selección d'ación (lo mesmu pal apercibimientu que pa cumpril un ojetivu determinau), el apercibimientu encubierto i la enclusión d'arrayus percetivus dependientis del contestu.

AGM propón un pranteamientu destribuúu ondi varius agentis entreactúan conformi ala gramática que rigi el mandu del robón, hiziendu que esti entreactúi con el entornu i muandu entre tous el modelu del mundu. Un nodu central s'acupa de gestional las muacionis (premitiendu namás aquellus que no enviolan la gramática) i cordinal el restu d'agentis. Dambas a dos tareas se sostriban en el gastu dun pranificaol, que es el que realmenti trabaja cona gramática, genera pranis i verifica que las muacionis son la resultancia d'aprical las regras dela gramática al modelu previu.

El restu delas aportacionis dela tesi, técnicas específicas atillás al modelau d'entornus d'interiol, se presentan comu esprimentus en ambientis realis, alo primeru gastandu de representacionis simpris de características conocías i alo últimu gastandu delas representacionis en grafus de grandol variabli que se proponin endrentu de Active Grammar-based Modeling. En particular, se presentan dos esprimentus que muestran cómo AGM puei sel gastau pa percebil ativamenti el ambienti, lo mesmu la abitación en sí que los ojetus que están endrentu d'ella.

Contents

| | | |
|----------|--------------------------------------------------------------|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivations | 1 |
| 1.2 | Contributions | 2 |
| 1.3 | Document Structure | 4 |
| I | RoboComp | 7 |
| 2 | The specific needs of software development for robots | 9 |
| 3 | Component-oriented programming | 11 |
| 3.1 | Advantages | 11 |
| 3.2 | A component-oriented programming example | 13 |
| 3.3 | Enhancing component-oriented programming | 16 |
| 3.4 | Model-driven Engineering | 18 |
| 4 | The RoboComp framework | 21 |
| 4.1 | Introduction | 21 |
| 4.2 | RoboComp main characteristics | 22 |
| 4.2.1 | Middleware | 22 |
| 4.2.2 | Component model | 24 |
| 4.2.3 | Tools and classes | 26 |
| 4.2.4 | HAL characteristics | 27 |
| 4.3 | RoboComp repository structure | 28 |
| 4.4 | Framework comparison | 28 |
| 5 | RoboComp DSLs | 33 |
| 5.1 | CDSL | 33 |
| 5.2 | IDSL | 37 |
| 5.3 | DDSL | 37 |
| 5.4 | PDSL | 39 |
| 5.5 | InnerModelDSL / IMDSL | 40 |
| 6 | RoboComp tools | 45 |
| 6.1 | RCManager | 45 |
| 6.2 | RCReplay | 46 |
| 6.3 | RCMonitor and RCControlPanel | 47 |

| | | |
|-----|---------------------------------|----|
| 6.4 | RCLogger | 47 |
| 6.5 | RCDSLEditor | 47 |
| 6.6 | RCInnerModelEditor | 48 |
| 6.7 | RCInnerModelSimulator | 48 |
| 6.8 | RoboComp components | 49 |

II Active Perception 51

7 Environment modeling 53

| | | |
|-------|---------------------------------------------------------|----|
| 7.1 | Introduction | 53 |
| 7.2 | World model structures | 54 |
| 7.3 | Deterministic versus probabilistic modeling | 58 |
| 7.4 | Probabilistic modeling and control | 59 |
| 7.5 | Bayes filter | 62 |
| 7.5.1 | Complementary concepts | 62 |
| 7.5.2 | The algorithm and its mathematical derivation | 63 |
| 7.6 | Histogram filter | 65 |

8 Stochastic sampling techniques 71

| | | |
|---------|----------------------------------------------------------------------------------|----|
| 8.1 | Numerically representing probability distributions: the PDF - sample set duality | 71 |
| 8.2 | Monte Carlo methods | 73 |
| 8.2.1 | Non-iterative Monte Carlo methods | 73 |
| 8.2.1.1 | Rejection sampling | 73 |
| 8.2.1.2 | Importance sampling | 75 |
| 8.2.1.3 | Sampling/Importance Resampling | 77 |
| 8.2.1.4 | Particle filtering: Sequential Importance Resampling | 78 |
| 8.2.2 | Iterative Monte Carlo methods | 80 |
| 8.2.2.1 | Metropolis sampling | 81 |
| 8.2.2.2 | Metropolis-Hastings sampling | 82 |
| 8.2.2.3 | Gibbs samplers | 83 |
| 8.2.3 | Simulated Annealing in MCMC | 85 |
| 8.2.3.1 | Bayesian filters using MCMC | 87 |
| 8.2.3.2 | Reversible jump MCMC | 87 |
| 8.3 | Improving Particle Filtering | 88 |
| 8.3.1 | Rao-Blackwellized particle filters | 88 |
| 8.3.2 | Annealed particle filters | 88 |
| 8.3.3 | Auxiliary particle filters | 89 |
| 8.3.4 | Partitioned Sampling | 89 |
| 8.3.5 | Subspace hierarchical particle filters | 90 |
| 8.3.6 | Branched Iterative Hierarchical Sampling | 91 |
| 8.4 | Discussion | 92 |

9 Active Grammar-based Modeling 95

| | | |
|-----|---------------------------------------------------|----|
| 9.1 | Autonomous robots and active perception | 95 |
| 9.2 | Grammars in robotics | 98 |

| | | |
|------------|-------------------------------------------------------------------------|------------|
| 9.3 | Active grammar-based modeling system description | 100 |
| 9.4 | Describing grammars | 103 |
| 9.4.1 | Associating active-perceptual configurations to rules | 104 |
| 9.4.2 | Dealing with qualitative metric information | 105 |
| 9.5 | A simple example | 106 |
| 9.5.1 | Grammar | 107 |
| 9.5.2 | Agents | 107 |
| 9.5.3 | Configuration | 108 |
| 9.5.4 | Execution | 108 |
| 9.6 | Using Active grammar-based modeling | 108 |
| 9.6.1 | Bottom-up parsing | 109 |
| 9.6.2 | Context-aware perception restrictions | 109 |
| 9.6.3 | Model verification | 109 |
| 9.6.4 | Covert perception | 110 |
| 9.6.5 | Planning and execution | 110 |
| 9.6.6 | Intersecting filters | 110 |
| 9.7 | Implementation issues | 111 |
| 9.7.1 | Tools | 111 |
| 9.7.2 | AGG to PDDL translation | 111 |
| 9.7.3 | Component-oriented implementation | 111 |
| 9.8 | Discussion | 112 |
| 10 | Grammar-based Cognitive Subtraction | 115 |
| 10.1 | Cognitive Subtraction | 117 |
| 10.1.1 | Generation of expected measurement data from the model | 117 |
| 10.1.2 | Adjustment between input and expected data | 118 |
| 10.1.3 | Detection of outliers | 120 |
| 10.1.4 | Implementation considerations of cognitive subtraction for point clouds | 121 |
| 10.2 | Results and conclusions | 122 |
| III | Experiments | 125 |
| 11 | Experiments map | 127 |
| 12 | RoboComp DSLs | 129 |
| 12.1 | Goal | 129 |
| 12.2 | CDSL | 129 |
| 12.3 | PDSL | 130 |
| 12.4 | DDSL | 130 |
| 12.5 | IDSL | 131 |
| 12.6 | Conclusions | 132 |
| 13 | Deterministic indoor room modeling | 137 |
| 13.1 | Goal | 137 |
| 13.2 | Solution | 137 |
| 13.3 | Room modeling | 138 |

| | |
|----------------------------------------------------------|------------|
| 13.4 Results | 141 |
| 13.5 Conclusions | 145 |
| 14 A graph grammar for indoor environment mapping | 147 |
| 14.1 Experiment description | 147 |
| 14.2 Symbols | 147 |
| 14.3 Grammar rules | 148 |
| 14.4 Experiment results | 149 |
| 14.5 Conclusions | 149 |
| 15 Yaw estimation | 153 |
| 15.1 Goal | 153 |
| 15.2 Solution | 153 |
| 15.2.1 State space and transition model | 154 |
| 15.2.2 Oriented point extraction | 155 |
| 15.2.3 Histogram generation | 156 |
| 15.2.4 Observation model | 157 |
| 15.3 Results and conclusions | 158 |
| 16 Distance estimation | 161 |
| 16.1 Goal | 161 |
| 16.2 Solution | 161 |
| 16.2.1 State space and transition model | 162 |
| 16.2.2 Observation model | 162 |
| 16.2.3 Implementation considerations | 163 |
| 16.3 Results and conclusions | 163 |
| 17 Room modeling | 167 |
| 17.1 Experiment description | 167 |
| 17.2 Symbols | 167 |
| 17.3 Grammar rules | 168 |
| 17.4 Agents and behaviors | 170 |
| 17.5 Implementation and software components | 171 |
| 17.6 Experiment results and conclusions | 172 |
| 18 A graph grammar for scene understanding | 175 |
| 18.1 Experiment description | 175 |
| 18.2 Symbols | 175 |
| 18.3 Grammar rules | 176 |
| 18.4 Agents and behaviors | 177 |
| 18.5 Experiment results and conclusions | 178 |
| IV Conclusions | 181 |
| 19 Review and contributions | 183 |

| | |
|------------------------|------------|
| <i>CONTENTS</i> | xiii |
| 20 Future works | 185 |
| 21 Publications | 187 |

Chapter 1

Introduction

1.1 Motivations

Developing robots able to robustly achieve useful tasks in non-trivial environments is one of the current main goals of robotics. Most of these tasks do not only require robots to be able to model and interact with the objects in the environment but also to interact with humans (communicating and accepting commands from them) and other complex elements. To enable robots for these tasks they must be able to perceive the environment autonomously and efficiently, so that the representations generated are useful to successfully achieve the missions they are required to perform.

Since some tasks can not be achieved in a pure reactive manner it is necessary that the robots generate some sort of environment representation. Some moderately simple tasks such as local and global navigation can be achieved using relatively simple structures such as two-dimensional occupancy grids. However, when robots are supposed to perform complex tasks it is necessary to use considerably richer and structured world models, with a higher semantic load than two-dimensional grids.

It is not necessary to suggest extremely complicated scenarios to justify this kind of models, instead it can be as simple as achieving human commands such as “pick up the blue mug and bring it to me”. In order to obey commands, robots must have symbolic information that they can use to parse the commands and generate a valid plan. Even if the plan could be hard-coded in the software, case in which it would be possible to use a pseudo-reactive control combined with a grid map, robots would not perform well (rich information would still be necessary to efficiently look for the mug) and it would be limited to a very specific task.

Despite the concept of hybrid modeling —combining symbolic and metric information— might be broadly accepted, lots of questions remain:

- How can these models be built?
- How can they be efficiently used?
- How can robot perception and action be bound so they complement each other?
- How can a whole robot software system be properly designed and implemented?

As robotics advances, it becomes more plausible to find robots working in real, non-engineered environments, where shadows, textureless areas, clutter and obstacles represent real

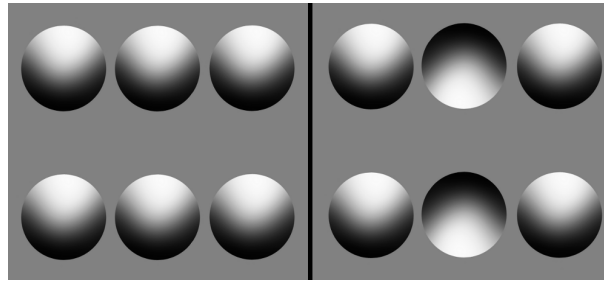


Figure 1.1: The *domino illusion*: humans —and probably other animals— are so influenced by the patterns they usually see in nature that brightness gradients sometimes makes them perceive depth changes. Despite this behavior might lead to optical illusions, this should not be thought as a handicap, since it also enables them to cope with situations where the sensors provide poor input. *Illustration by E. Martinena (<http://emartinena.es>), licensed under the Creative Commons Attribution-Share Alike 3.0 license.*

challenges not only to perceive the world but also to perform actions. Despite that these environments are more complex and less predictable than engineered ones, they are not composed of random shapes nor change without a reason. To build actual intelligent robots it is necessary to seize all the *ecological* information available.

As pointed by several psychologists such as Gibson [Gibson, 1986] or Frith [Frith, 2009], the brains of the animals do not process senses in an objective way. As Gibson points out, animal brains are adapted to life in the Earth, and this adaption led to the prejudices and assumptions that made animals subjective, but also effective and efficient as long as living in the Earth. An example of this subjectivity is the so called "domino illusion" (see figure 1.1). The light that can be seen on Earth usually comes from the Sun, so objects are rarely illuminated by light sources located under seen objects (which includes obstacles). As can be seen in figure 1.1, this make people infer 3-dimensional properties from particular shadowing patterns. So to speak, it is better being an efficient subjective robot/animal than a objective one that is unable to accomplish any task because it tries to rationalize every single outcome.

Robots are located in environments even more predictable than those of animals (with few exceptions) and should probably be adapted to perceive their environment in the most likely way —dismissing unlikely possibilities, and so, making them able to perform in a faster and usually safer way. Thus, it would be desirable to have a technology that allows to build robots able to model and interact with their environment using all the a priori information available about the environment in an efficient way, so that robots can interact with it appropriately.

A priori information is even more important in indoor environments, where the rooms and objects within them are mostly composed of small sets of geometric shapes and, in many cases, have strong structural relationships. Thus, it would be desirable to have a technology to build robots capable of modeling and interacting with their environment using all the a priori information available in an efficient way. The goal of this work is the study and development of novel techniques that can be used to cover these needs.

1.2 Contributions

I Active Grammar-based Modeling:

Active Grammar-based Modeling (AGM) is the main contribution of the thesis. Active Grammar-based Modeling is based on the fact that the changes that take place in the world models robots generate and use can be described using formal grammars. In AGM, grammar rules are decorated with the configuration of the different physical and perceptual behaviors that a specific robot might have —being the latter formulated as stochastic sampling processes. These grammars can then be translated to classic AI planning languages (PDDL) and provide solutions to different perceptual phenomena: bottom-up parsing, action selection (both for perception and mission accomplishment), covert perception and the inclusion of context-dependent perceptual restrictions.

AGM proposes a distributed approach where several agents interact according to a grammar that describes how world models can be created and modified. Such grammar is used to guide robot control and perception, making it interact with the environment and modify the world model as a result of the interaction between all agents. A central node is in charge of coordinating the agents and managing the changes that they propose (allowing only those made according to the grammar). Both tasks are based on the use of classic AI planners, which is the element that takes the grammar as input (after being translated to a planning language). It generate plans and verify that the changes are the result of applying the rules of the grammar to the current model. AGGL (Active Graph Grammar Language), a new domain-specific language has been designed in order to describe and translate the grammars to PDDL automatically.

Grammar-based Cognitive Substraction, other of the most important contributions of the thesis, is proposed as a solution to drive the attention of the robot towards those areas of the world for which the robot perceives unexpected sensorial inputs. Along the use of Active Grammar-based Modeling, it allows the robot to autonomously build complex representations where the elements of the environment and their positions are unknown.

II RoboComp:

Regarding software engineering for robotics, all the contributions have been integrated in RoboComp, an open-source (GPL) robotics framework. RoboComp stands out for being component-oriented, for the set of tools and libraries it provides and for making extensive use of domain-specific languages (DSL) that facilitate and accelerate the development of software for robots. Among the developed tools it is worth mentioning RCManager (a graphical manager of component networks), RCMonitor (a monitoring and visualizing tool for components), RCControlPanel (a tool to simultaneously access and control several components) and RCInnerModelSimulator (a robotics simulator). InnerModel and InnerModelViewer are also of particular interest. They are a pair of classes designed to represent, manage and visualize kinematic trees. The kinematic trees used by these classes are described using IMDSL, a domain-specific language used to describe this kind of structures.

III Experiments:

The rest of the contributions of the thesis are specific techniques related to indoor environment modeling and are presented as experiments in real environments. The first experiments present novel particle filters which are used to estimate different variables of interest in indoor environments. They use simple representations with a fixed structure. The rest of the experiments use the graph-based representations of unknown size

and structure that are supported in this thesis. Specifically, the last two experiments show how AGM can be used to actively perceive the environment, not only the rooms but also the objects within them.

1.3 Document Structure

This thesis is organized in the following parts:

I RoboComp:

Developing complex autonomous robots involves not only designing and implementing their hardware and algorithms. In fact, programming is an extremely important issue because it is one of the most time-consuming tasks. Part I describes RoboComp—a robotics-oriented software framework—the tools and technologies which enabled the development of the algorithms proposed in later parts.

Chapter 2 discusses the peculiarities that software development in robotics has and why a robotics-specific framework is needed. Chapters 3 and 3.4 describe two of the technologies used in RoboComp: component-oriented programming and model-driven engineering. If the reader is familiar with this concepts chapters 2 to 3.4 can be safely skipped. Finally, chapters 4 to 6.8 introduce the RoboComp framework (chapter 4), the Domain-Specific Languages involved (chapter 5), the tools provided by RoboComp (chapter 6) and some of the most relevant components (chapter 6.8).

II Active Perception:

In the active perception part it is introduced the state of the art in chapters 7 and 8:

- the need for active perception
- the different types of world models and their characteristics
- the most relevant types of algorithms used to build world models

Active Grammar-based Modeling perceptive architecture (in chapter 9) and finally, Grammar-based Cognitive Subtraction (chapter 10).

III Experiments:

Finally, part III describe the development of different experiments carried out during this work in the field of indoor modeling: first, using simple representations of known characteristics, and then, using the graph-based representations that are supported within this work. Chapters 12, 13, 14, 15, 16 and 10 belong to the first set. They present different experiments related to each of the secondary contributions of the thesis. On the other hand, chapter 17 y 18, show how AGM can be used in order to perceive the environment. First, chapter 17 approaches indoor room modeling using AGM, where the most relevant concepts and implications of the use of AGM are highlighted. Finally, chapter 18 considers a more complex experiment, where the robot does not only have to model the room but also the elements within it.

IV Conclusions:

Part **IV** reviews the work done and provides a discussion on the contributions, the results of the thesis and future works.

V Publications:

Chapter **21** lists the papers published during the development of this work.

VI Index and Bibliography:

The dissertation concludes with the index and the bibliography. To facilitate finding the terms included in the index they are highlighted with a ✓ symbol in the margins.

✓

Part I

RoboComp

Chapter 2

The specific needs of software development for robots

Developing complex autonomous robots involves not only designing their hardware and algorithms. It also involves a great deal of day-to-day configuration, programming, bug solving, and dealing with issues such as efficiency, complexity, code reuse, scalability, distribution, hardware independence, concurrency, or language and platform support. These issues should be handled with the proper tools and software engineering techniques in order to meet the specific needs of robotics software systems and their developers.

Robot control programs generally share most of the modules they are composed of. Linear algebra, machine learning or image and point cloud processing algorithms are examples of software modules that can easily be reused by packaging them as dynamic libraries. However, developing monolithic software systems that use large amounts of libraries is not generally a good idea. Libraries are difficult to integrate when they have an active role on the robot's hardware or require to have their own execution loop. In some cases the software, or the libraries they use, have dependencies on different versions of a specific library, which leads to symbol collisions. This is more common in robotics than in other areas due to the fast-paced changes that the related libraries experiment. Decomposing the system in task-driven highly decoupled modules would help solving these problems. Moreover, running these modules in different computers is an easy way to distribute the computational load of the system. Additionally, it makes easier understanding the structure of robotics software (both for the developers and their users) and bounds bugs to the context of each module, which in turn makes easier solving them.

Hardware independence is also a very desirable feature. In the context of robotics, **hardware independence** means that a robotic system is able to continue working after modifying some of its hardware parts with equivalent ones by performing minor configuration changes only. It enables us to change the platform or the cameras of a robot as long as the new ones have the same capabilities that the old ones (*e.g.*, two differential bases, two RGBD cameras, two linear actuators) without changing a single line of code. Despite it is now a little outdated, the developers of the Player/Stage project [Gerkey et al., 2003] were perfectly aware of the impact of hardware independence in the reusability of the code and proposed the “Player Abstract Device Interface” (PADI) in [Vaughan et al., 2003], which consisted on a specification to define how robotics software should communicate with their hardware drivers using standard APIs. ✓

The main requisite to achieve hardware independence is that the modules accessing the hardware must have the same programming interface. However, this is a rather problematic

issue, since the interface to access similar robotic parts, as provided by the manufacturer, might considerably differ even though they might have almost the same features. For example, a robot actuator might be able to change its PID configuration on-line, while other similar actuators might not. Deciding what to include in the interface of a module is a tricky and delicate task: if the interface includes a high number of seldom available features, the interface will be complex and hard to understand; however, if it is unrealistically simple, it will be useless for a considerable number of robotic applications.

A wide and transparent platform and language support is also desirable. First, because autonomous robots are seldom programmed by a single person but by multidisciplinary teams of people with different preferences and requirements. Moreover, the software of the robots might necessarily be distributed over heterogeneous platforms (*e.g.*, when some software modules are only available for a specific language or operating system, or when working among different research groups with different policies or operating systems). Depending on the robot, it might be controlled from a hand-held device such as Android tablets or a remote web interface, so sometimes it is essential that different modules can interact with each other regardless of the operating system they run or their programming language. Other reasons to support multiple languages are that it makes possible reusing much more libraries and that some tasks are better approached using specific programming languages.

Maintaining the previous features (platform and language independence) while distributing the computational load of the robots is frequently a must. However, distributed systems might have efficiency and synchronization issues, specially in complex and heterogeneous distributed systems. The synchronization problems are common to most distributed systems, since each node generally has to perform some task while interacting with its peers. The efficiency problems are due to message passing and are heavier if the communication is platform and language independent. In that case messages must be serialized (*i.e.*, convert complex data structures to streams of data which can later be used to reconstruct the original structure, also known as *marshalling*) and translated to a platform and language independent representation before its transmission and the opposite process must be executed upon reception (potentially to a different representation with a different endian-ness).

All the problems pointed out in this chapter should be addressed with appropriate software engineering techniques in order to make development (programming, but also other tasks such as testing, bug solving, and configuring the software) as efficient as possible. The fact that in the field of robotics teams are sometimes heterogeneous groups of experts should also be taken into account and it should not be assumed that they have a Computer Science or Software Engineering degree. Developing robots should be made accessible enough for all these experts to participate in the development process. More importantly, since the development process is a time-consuming task, it would be extremely interesting to boost it by providing tools to create and connect modules, to monitor how each of the modules work, or to find bugs and test the software. The remaining of the first part of the thesis introduces some useful technologies regarding these topics and describe RoboComp, a robotics framework that makes use of them, which has been partially developed in the course of this thesis.

Chapter 3

Component-oriented programming

This chapter describes the basic concepts of component-oriented programming (COP), a paradigm that, despite being relatively old, has gained great importance during the last years in the robotics field (see [McIlroy, 1969, He et al., 2005]). Currently, almost all software for robots is designed and developed using this paradigm, from the most advanced robots (autonomous manipulators, social or assistive robots) to those used in introductory courses in robotics. Despite component-oriented programming solves or mitigates most of the issues described in chapter 2, its main advantage is that it makes much easier to build scalable and reusable robotics software.

The idea behind *component-oriented programming* (*COP*) is designing and developing complex systems as networks of standalone modules, so that the goal of the whole system is achieved through the interaction among the elements of the network. These independent programs are called components, and the software layer they use to communicate is usually called *middleware* . Each component has a *component interface* which other components can use to interact with it. Component interfaces can be seen as the API that other components can use to communicate with each other, much like C++ class definitions. In fact, as virtual classes, component interfaces can be implemented by more than one component (*i.e.*, the same interface can be provided by several components).

3.1 Advantages

Being able to define standardized interfaces for specific tasks enables two of the most useful capabilities of component-oriented robotics systems: **a)** hardware abstraction; **b)** easy replacement of software components. A module using a component interface to access the images of a camera will generally be unable to tell which camera is it using. It will only fetch images from another component with the interface of a camera, regardless of the type of bus the camera uses, the vendor, or if the images come from a real camera, a simulated one or a recorded video file. The fact that components only have to know about the interface they are going to connect to makes easy to try new algorithms —by substituting the components in which they are implemented— as long as all of them are encapsulated in the same interface.

Figure 3.1 shows a small dependency graph (*i.e.*, a graph where edges indicate that the components at the origin use the components at the end) that can be used a simple example of the advantages of component-oriented programming. It shows a network of five components containing an ASR component (Automatic Speech Recognition), a TTS component (Text To

Speech), a human detector, a camera image grabber and a conversational module. The overall goal of the network is to monitor, listen to humans and talk back to them depending on what they say. To start with, the functionality and the computation is distributed over the five components, which could potentially be run in different computers. Besides distributing the computation, this setup makes easy to find bugs because they are isolated within the context of the processes in which components are run. It also makes much easier to understand how the whole system works, which is not a hard endeavor in this case, but things can get much more complicated. Moreover, since components are standalone programs, they are easier to update or replace when needed, as long as the two components (the one to be replaced and the replacement) provide the same interface. Code reuse is also improved, since components are generally easier to reuse than libraries: unlike regular classes, components can be pre-compiled programs, so users do not necessarily have to compile them, not even to link against them, which is a cumbersome topic sometimes —specially when two libraries depend on different specific versions of another library (quite frequent if using state-of-the-art software). In fact, in order to avoid awkward library or operating system dependencies, components could even be executed in different virtual machines on the same or different computers. Components only need to have information about how to locate the component they are supposed to connect to and their interface, which is usually an IDL (Interface Description Language) file, where the interface of the component is described using a high-level language.

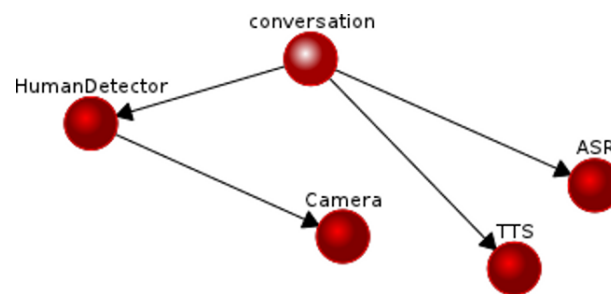


Figure 3.1: A small component dependency network. Edges represent dependencies of the node of origin on the node of the end of the arrow. This particular network is used to interact with humans using voice and gestures.

Components are generally thought as independent processes that perform a specific task and can be used in different systems (networks of components) by communicating with them. However, it is possible to find technologies that are referred to as “component-oriented” while they are implemented as networks of communicating threads, not processes. These approaches, despite being slightly more memory-efficient and a tiny bit more time-efficient, provide reduced benefits on the most important strengths of COP: reusability, hardware abstraction, bug isolation or scalability. From now on, we will assume the use of a component-oriented programming at a process level.

Other of the important aspects that conditions a COP technology is how components communicate. Depending on the middleware used, components might or might not be able to communicate with component outside the same operating system. Also, the variety of patterns available for components to communicate can be very different. In the works presented in [Schlegel, 2004, Chapter 5] and [Schlegel, 2006], six communication patterns are highlighted: send, query, push-newest, push-timed, event, and wiring (see table 3.1).

Table 3.1: Communication patterns detected in [Schlegel, 2004] and [Schlegel, 2006].

| | |
|-------------------------------------------|------------------------------------|
| send | One-way message |
| query (a.k.a RPC) | Two-way request/response |
| push newest (a.k.a. pub/sub) ^a | Message broadcast |
| push timed | Periodic message broadcast |
| event | Asynchronous event broadcast |
| wiring | Component connection/disconnection |

^aThe patterns *push timed* and *event* can be seen as special cases of the *push newest* (push) pattern.

The rest of the chapter provides a simple example of how are components generally programmed and discusses how can the experience and efficiency of robotic software developers and users be improved.

3.2 A component-oriented programming example

In this section we will grasp the idea of component-oriented programming using a simple example composed of two components. The first one will read strings from the standard input and will ask the other component to compute the number of vowels in the strings using RPC-like calls. The resulting network is depicted in figure 3.2. Despite this example is too simple to use component-oriented programming it would not be that way if the first component, instead of reading strings and printing the number of vowels computed by the second, fetched laser readings from the other component in order to make grid maps. In fact, in this case one of the components is client-only and the other is server-only; in the general case components act as both clients and servers. The remaining of the section will describe the interfaces of the components and will provide the necessary code and instructions to reproduce the example. For this example, we will be using the Python programming language and the ZeroC Ice middleware: Python because it is very easy to understand and Ice because it is the main middleware of the RoboComp framework. All files are supposed to be placed in the same directory.

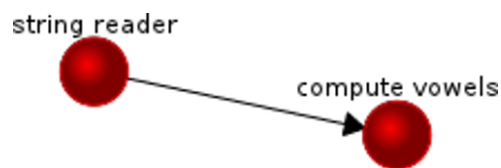


Figure 3.2: Graphical representation of the network resulting from the example.

Interface files

Since only one of the components will act as server (this is not the general case, in which the majority of the components act as servers and clients at the same time), it is only necessary to write a single interface file.

Listing 3.1: *VowelCounter.ice*

```

1 module VowelCounterModule
2 {
3     interface VowelCounter
4     {
5         int compute(string text);
6     };
7 };

```

- ✓ Generally, the interface description is written in a middleware-dependent **Interface Definition Language** (*IDL*). The IDL of Ice is named Slice [Henning and Spruiell, 2005]. Basically, the IDL file shown in listing 3.1 describes an interface named “VowelCounter” which is composed by a single method named “compute” which receives a string and returns a number. The interface is included in a namespace named “VowelCounterModule” in order to avoid redefining types or interfaces.

Vowel counter

The code of the component computing the number of vowels is shown in listing 3.2. The component has two classes: **a)** “VowelCounterI”, which will be the class attending the remote calls, implements all the methods defined in the IDL file (this kind of class is generally referred to as servant); and **b)** “Server”, which apart from instantiating a VowelCounterI instance, only contains middleware-related code.

Lines 6-7 load the IDL file shown in listing 3.1 and import its contents as a Python module. Lines 10-15 implement the interface defined in the IDL file. Since it only contains a single call, the class only needs to implement one method. Lines 17-35 perform all the middleware-related tasks necessary to make an Ice component: create an object adapter for the VowelCounter interface, associate it to a new instance of the VowelCounterI class, activate it, and wait for shutdown. Line 37 instantiates the server class.

Listing 3.2: *vowelcounter.py*

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sys, traceback, Ice, subprocess, threading, time, Queue, os
5
6  Ice.loadSlice("VowelCounter.ice")
7  import VowelCounterModule
8
9
10 class VowelCounterI(VowelCounterModule.VowelCounter):
11     def compute(self, text, current = None):
12         print "We've got a new command:", text
13         vowels = len([x for x in text if x in 'aeiou'])
14         print "We will return", vowels
15         return vowels
16
17 class Server(Ice.Application):
18     def run(self, argv):
19         status = 0
20         try:
21             self.shutdownOnInterrupt()
22             adapter = self.communicator().createObjectAdapter('VowelCounter')
23             adapter.add(VowelCounterI(), self.communicator().stringToIdentity('vowelcounter'))
24             adapter.activate()

```

```

25         self.communicator().waitForShutdown()
26     except:
27         traceback.print_exc()
28         status = 1
29
30     if self.communicator():
31         try:
32             self.communicator().destroy()
33         except:
34             traceback.print_exc()
35             status = 1
36
37 Server( ).main(sys.argv)

```

Listing 3.3 shows the configuration file for the “vowelcounter” component. Since it only acts as a server it just needs to read the port it should listen to and the associated protocol. This information is generally known as its *endpoint* .

✓

Listing 3.3: *vowelcounter.cfg*

```

1 VowelCounter.Endpoints=tcp -p 10000

```

String reader

The component using the previous one is shown in listing 3.4 and is also very simple. In this case it is only composed of a single class named “Client”. Lines 15-19 ensure the configuration information necessary to connect to the vowelcounter component is properly read. Lines 21-27 use that information to connect to the remote component and instantiate a proxy to the other component. Such proxy will be used as if the proxy was the remote component and it is accessed as a regular class. Lines 29-34 act as the main code of the component: ask the user to type a message, store the message in a variable, retrieve the result given the message from the vowelcounter component (line 33), and print it.

Listing 3.4: *stringreader.py*

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sys, traceback, Ice, os
5
6  Ice.loadSlice("VowelCounter.ice")
7  import VowelCounterModule
8
9  class Client (Ice.Application):
10     def run (self, argv):
11         status = 0
12         self.shutdownOnInterrupt()
13
14         # Get connection config
15         try:
16             proxyString = self.communicator().getProperties().getProperty('VowelCounterProxy')
17         except:
18             print 'Cannot get VowelCounterProxy property.'
19             return
20
21         # Remote object connection
22         try:
23             basePrx = self.communicator().stringToProxy(proxyString)
24             self.vowelcounter_proxy = VowelCounterModule.VowelCounterPrx.checkedCast(basePrx)

```

```

25     except:
26         print 'Cannot connect to the remote object.'
27         return
28
29     while True:
30         print "Type some input:",
31         text_read = raw_input()
32         print "We've read:", text_read
33         result = self.vowelcounter_proxy.compute(text_read)
34         print "According to the other component what we've read has", result, "vowels."
35
36 c = Client()
37 c.main(sys.argv)

```

In this case, the configuration needed by the component (shown in listing 3.5) is not which port should it listen to remote calls but where to locate vowelcounter, the component it uses. Since more than one interface might be using the same port, it needs the identity of the component ('vowelcounter'), the port (as in listing 3.3, '10000') and the host ('localhost').

Listing 3.5: *stringreader.cfg*

```

1 VowelCounterProxy = vowelcounter:tcp -p 10000 -h localhost

```

Running the example

In order to run the example it is necessary to install python and its module python-zero-ice, both available in most modern GNU/Linux distributions and other operating systems. Assuming all the files are stored in the same directory and it is the current working directory, the vowelcounter component is run by executing:

```
python vowelcounter.py --Ice.Config=vowelcounter.cfg
```

The stringreader component is similarly run by executing:

```
python stringreader.py --Ice.Config=stringreader.cfg
```

Of course, it is possible to run both components in different machines as long as the configuration of the client component is properly updated. It would be necessary to substitute 'localhost' for the host in which the vowelcounter component is to be run.

3.3 Enhancing component-oriented programming

Despite the use of component-oriented programming (COP) is extremely helpful in order to mitigate many of the problems described in section 2, there still are many aspects which could be improved. COP only enhances bug solving passively in the sense that, since each component is run as an independent process, some types of programming errors are isolated within a single component (e.g., buffer overflows, deadlocks). The rest of semantic errors are generally spread over the network to other components, so it would be interesting to have specific tools to verify each component works as expected.

Additionally, using COP instead of libraries has two rather important drawbacks: **a)** unlike monolithic programs using libraries, components need to include a considerable number of extra lines of code to connect to the components they will be interacting with and to check for

connection errors; and **b**) components must be properly configured. These disadvantages are the price to pay in order to get an actually modular, flexible distributed system. Regarding the extra lines of code, the example shown in section 3.2 provides a sense of the amount of code not directly related to the problem to solve. Only lines 12-15 of listing 3.2 and lines 29-34 of listing 3.4 are strictly written for that purpose. Although the number of middleware-related lines is not very big in absolute terms, it can grow depending on the number of interfaces to implement and the number of components to connect to. Since these tasks are quite systematic, error-prone and will unlikely be of the interest of roboticists, it would be desirable to automate them and abstract developers from middleware-related code. These kind of issues should be automated by providing the appropriate tools. Derived from the distributed nature of component-oriented programming there is one last little nuisance: components must be properly deployed—in potentially different computers—and monitored. These tasks could be manually done in as many command lines as components in the system, but it is annoying and inefficient when using more than a few components. Consequently, deployment and monitorization should also be made easier.

Middleware-independence is another desirable feature. Despite there is no known middleware-independent robotics frameworks yet, there are two robotics frameworks working towards it: SmartSoft (which provides support for two different middlewares [ScGutierrezhlegel et al., 2010]) and RoboComp (see [Gutiérrez et al., 2013]). The idea is that the middleware to use can be chosen, whether at compile-time, or even at run-time using multiple compatible middlewares concurrently. In order to achieve this, components have to use a middleware-agnostic API whose calls are transformed in middleware-specific calls (whether at compile time or at execution-time).

There are also other robotics related features that general COP frameworks do not provide. For example, COP provides the basic features needed in order to implement hardware independence, not the implementation itself. Of course, it is to some extent a robotics-specific issue and is not covered by any middleware.

In order to debug components it is also interesting to have logging facilities within the framework. Despite some component programming frameworks enable us to log messages, the fact that robots usually work with very specific data types could be used to provide specific visualization tools. In fact, this logs could also be used to replicate—replay—the behavior of the components in the logs.

Robot simulation is an interesting feature to offer on the top of the middleware. Simulation is useful in many different contexts, *e.g.*: **a**) modeling new robots; **b**) developing robotics software without the need of using an actual robot—it is faster and much safer sometimes—; **c**) developing robotics software when it is hard to gain access to the actual robot—such as in academia, where robots are shared by several students—; or **d**) to identify whether the software of a robot is not working properly due to noise and uncertainty or other problems—if the simulation can be set to work in ideal conditions.

The robotics community is aware of the benefits of using component-oriented programming and how can it be used to deal with complexity, and has already adopted it to some extent. Despite the most relevant frameworks use COP, they do not usually enforce making a component-oriented design, so their users still tend to use some components and then implement all their ideas in a single component (RoboComp, SmartSoft and OpenRTM are the only known exceptions). Despite the ostensible consensus regarding the use of COP, many different approaches have been developed in order to deal with all

the robotics specific issues presented in chapter 2. The works in [Montemerlo et al., 2003, Brooks et al., 2005, Gerkey et al., 2005, Newman, 2006, Brooks et al., 2007, Utz et al., 2007, Cote et al., 2007, Cañas et al., 2007, NAIST, 2010, Manso et al., 2010] are some examples of the research in component-oriented programming for the field of robotics. In chapter 4 it is presented the robotics framework RoboComp [Manso et al., 2010]. It is an open-source (*libre*) robotics component-oriented framework that aims to provide solutions for the previously mentioned problems, focusing on ease of use and rapid development without losing technical features. It provides hardware independence, mechanisms to manage and monitor components, and a full-featured set of tools to make easier and as transparent as possible the development of robotics software components. RoboComp is based on Ice [Henning and Spruiell, 2007], the lightweight industrial-quality middleware used for the example on section 3.2. Instead of spending large amounts of time developing an ad-hoc middleware, it was found preferable to reuse an existing one. In order to fully understand the benefits of RoboComp it is advisable to have some basic notions of Model-Driven Engineering (MDE). If the reader is already familiar with the concept of MDE, section 3.4 can be safely skipped.

3.4 Model-driven Engineering

Much effort has been placed in developing tools that provide better software reusability and scalability. When it comes to software for robotics, the need for these tools is even higher due to the multidisciplinary and heterogeneous nature of the programs that are built. Component-oriented programming, which to some point can be seen as an extension of object-oriented programming, is a promising direction towards achieving these goals. As previously introduced, components are standalone programs providing an interface for interaction with other components, so that any whole network of interacting components serves a particular purpose. This design makes them a more autonomous and reusable concept than classes. On the other hand, robotic software components have a life-cycle that can become quite complex. It includes all common activities that are present during the life-cycle of any program: requirements analysis, design, implementation, unit testing, system integration, verification and validation, operation support, maintenance and disposal. However, due to the ever-changing requirements typical of robotics software, the life-cycle of robotics components is extremely active and it is very common to modify structural properties of the components being developed (see [Brugali and Scandurra, 2009, Brugali and Shakhimardanov, 2010]). In these cases, it generally entails performing changes in middleware-related code continuously. To help developers in these tasks, specific tools are required.

- ✓ **Model-driven engineering** (*MDE*) is a generic methodology —not a specific approach— that makes use of the existing knowledge on a specific domain to create generic models of the problems to solve —known as meta-models— and provide tools for describing and solving concrete problems, enabling users to work in a higher level of abstraction and increase their efficiency. Using MDE entails designing a new **Domain-Specific Language** (DSL) that fits the described meta-model and can be interpreted or compiled into a (generally) general-purpose language. In some cases, MDE not only improves the way developers work but also makes possible that users who are not familiar with this general-purpose language can work in the domain of interest.

In the case of RoboComp, there were found several tasks that could be enhanced by provid-

ing a domain-specific language:

Code generation Besides the middleware-related code that has already been discussed we want to avoid, it would also be interesting to automate other tedious tasks such as modifying structural properties of the program (*e.g.*, threads, state machines), including a graphical user interface, or managing the libraries the program should link against.

Interface description Most middleware alternatives already provide interface description languages (IDLs). Ironically, to achieve middleware-independence was necessary a new IDL: one that selects the most useful features that robotics software components will use.

Component deployment Each time a component network must be deployed all its components must be executed. This makes manual component deployment, by far, the most tedious task when working with robotic software components. A specific tool should parse component network descriptions and provide a means to easily manage the execution of components.

Component configuration Despite configuring components is not especially time-consuming, it is extremely error-prone. Here, we propose to use two languages: a) one to describe the elements that can or need to be configured and the restrictions of their values; and b) a language to describe specific component configurations according to the model described in the previous language.

Kinematic trees description Kinematic trees are acyclic graphs of geometric relationships that replicate the kinematic structure of robots. They are extremely useful for a wide variety of tasks in robotics. However, hard-coding them in a general-purpose language is error-prone and very time consuming. Thus, it is also an interesting problem to solve using a DSL.

As demonstrated in [Romero-Garces et al., 2011], providing domain-specific languages for these tasks led to remarkable efficiency improvements. All these DSLs are detailed in chapter 5.

The SmartSoft [ScGutierrezhlegel et al., 2010] robotics framework also provides an MDE tool based on a UML profile implementation. First, developers have to create a platform-independent model (PIM) where specific information about middleware, operating system, programming languages and other properties are ignored. Then the PIM is transformed to a platform-specific model, introducing details about middleware or operating systems. This PIM is also transformed to a platform-specific implementation where developers can add their code and libraries. Regarding component deployment, SmartSoft uses the platform description model to define the target platform properties. The model is extended with this platform information and finally the system can be run following the specified deployment model. During the development process users guide these model transformations in order to obtain a specific component by selecting the desired real-time and QoS properties of the component and the communication middleware it will use.

SmartSoft provides an interesting set of well-defined middleware-independent communication patterns that provides the necessary abstraction from the final communication model and its reference implementation. Currently, it supports the ACE [Schmidt and Huston, 2002] (SmartSoft/ACE) and ACE/TAO [Schmidt et al., 1997] (SmartSoft/CORBA) communication middlewares and provides other interesting features, such as a mechanism to guarantee real-time properties using an external scheduler analyzer or dynamic wiring for components.

Chapter 4

The RoboComp framework

4.1 Introduction

As explained in chapter 2, developing software for robotics has several particularities that should be taken into account. Especially with modularity and reusability in mind, the development of the RoboComp framework started in late 2005 in the Robotics and Artificial Vision Laboratory of the University of Extremadura. By the time, the most remarkable options were the Player/Stage project [Gerkey et al., 2003], Orca [Brooks et al., 2005] and Carmen [Montemerlo et al., 2003]. Carmen had very interesting components for localization, mapping and navigation, but its components used IPC¹, which implied that all components had to be instantiated within the same operating system. It was not considered an option for our purpose because such approach does not support distributing the computation. Despite Player/Stage supported distribution, it still required users to manually program the (un)marshaling of the data structures to transfer every time they wanted to create a new data-type/interface or modify an existing one. This was acceptable for the main purpose of the Player/Stage project, to act as a hardware abstraction layer, but not for being used as a component-oriented framework, where interfaces are created and modified on a daily basis. In turn, Orca supported decentralized distributed communication, and once Orca2 [Brooks et al., 2007] appeared —based on the Ice [Henning and Spruiell, 2007] middleware— users were freed from programming marshaling code. After experimenting with Orca for some time, the development team chose to create a new framework from scratch, to a certain degree because of the way Orca used the Ice middleware (*e.g.*, Orca components could only publish a single topic and all topics had to go through the IceStorm node). Moreover, the objective of RoboComp went further: it aimed to provide not just a set of components, but also a component model and a component generator tool which could automatically generate working components with the selected interface and proxies (the classes used to connect to other interfaces). That is, the purpose was to free developers from writing middleware-related code and let them focus on writing robotics-related code. In order to achieve this it is necessary to develop a component model, a structure that all components have when they are generated². One of the main reasons why a new framework was created, instead

¹IPC stands for Inter-Process Communication.

²When a component is created all middleware-related code is automatically generated in order to free the user from writing it. Despite users are free to modify their components' structure after its creation, it would have, however, have some implications: if users want to maintain these changes they have to avoid re-generating the code, since this would rewrite the initial component structure.

of contributing to Orca2, is that it would involve imposing our component structure to previous users. Nevertheless, although they are independent projects, using the same middleware facilitates the inter-operation between both frameworks.

Nowadays, RoboComp is a state-of-the-art robotics framework. The set of tools it is accompanied with has grown dramatically and, along its component model, which has evolved over the years, is the most remarkable strength of RoboComp. These tools allow to perform daily tasks such as visually managing networks of components, monitoring, testing or creating and modifying components, make programming easier and more agile. RoboComp also provides a wide variety of ready-to-use components, ranging from hardware interface or data processing to robot behavior. The web of the project gives access to an extensive on-line documentation covering all the information users may need. New components can be easily integrated with existing ones just by placing their interface description file (IDL file, see chapter 3) in an accessible path by the components which are supposed to use it. Moreover, RoboComp components can communicate with other widely used frameworks. In particular, successful integration has been achieved with Orca2, ROS and Player/Stage.

4.2 RoboComp main characteristics

In addition to the technologies presented in the previous section, many different robotics frameworks have been proposed over the years. The most remarkable ones are: Carmen [Montemerlo et al., 2003], JDE [Cañas et al., 2007], Marie [Cote et al., 2007], Miro [Utz et al., 2007], MOOS [Newman, 2006], OpenRTM-aist [NAIST, 2010], Orca2 [Brooks et al., 2007], OROCOS [Bruyninckx, 2001], Player [Gerkey et al., 2005], ROS [Quigley et al., 2009], YARP [Fitzpatrick et al., 2008]). Each of them appeared in different moments and are designed with different objectives and so each framework has its own strengths and weaknesses.

One of the main reasons why a new framework was created, instead of contributing to Orca2, is that it would involve imposing our component structure to previous users. Nevertheless, although they are independent projects, using the same middleware facilitates the inter-operation between both frameworks. Therefore, RoboComp and Orca2 users can share their components and benefit from the advantages of both projects.

The rest of this section describes the main characteristics of RoboComp and, for a better understanding of it, a discussion of the most important design decisions involved: why the Ice middleware was selected, the component model, the tools it is accompanied with, how the hardware abstraction layer is implemented, how the components are structured and how new ones can be integrated.

4.2.1 Middleware

The middleware is one of the most important parts of any distributed component-oriented robotics framework. It allows to make components communicate without dealing with low level networking issues such as opening and closing sockets, instantiating multiple threads and so forth. Therefore, all known distributed component-oriented robotics frameworks rely on a middleware.

Despite work to be middleware-independent is underway (see [Gutiérrez et al., 2013]),

communications in RoboComp are currently handled using the ZeroC Ice middleware [Henning and Spruiell, 2005]. Creating an industrial-grade middleware such as Ice [ZeroC, 2010] is an extremely hard task that would have required similar manpower than creating a robotics framework itself, especially if it was planned to support many different platforms and programming languages. Ice works on many platforms and languages, is free (GPL licensed, so it will always be free) and tremendously stable. In addition to the cost of the initial development, the middleware is also maintained by ZeroC, so all the modifications needed to correct bugs and keep up with the updates of the APIs of the supported platforms are performed by ZeroC as well. Despite all these reasons, other projects have decided to develop their own middleware; this is the case of MOOS [Newman, 2006] and ROS [Quigley et al., 2009]. To the date, the only small drawback the RoboComp team has detected using Ice is that the GPL license Ice uses—as opposed to the LGPL or BSD licenses—do not allow to create proprietary components without paying for a special license.

In particular, besides the stability—Ice has been used in various critical projects [ZeroC, 2010]—and effort saved up, the RoboComp development team chose Ice because it meets all the requirements experimentally identified for a robotics framework:

- Support for different optional communication methods (*e.g.*, RPC, pub/sub, AMI, AMD).
- IDL-based, strongly typed interfaces.
- Good performance with low overhead.
- Efficient communication.
- Multiple language support (*e.g.*, C++, C#, Java, Python).
- Multiple platform support (*e.g.*, Linux, Windows, Mac OS X).

Even though users could potentially create higher communication layers and simulate other communication methods using the existing ones—making all of them equivalent to some extent—that is an extremely time-consuming and error-prone task; hence not a very good idea. Therefore, selecting a middleware that covers the communication patterns users are going to need is an important requirement. As introduced in 3.1, the works presented in [Schlegel, 2004, Chapter 5] and [Schlegel, 2006] presented an interesting communication pattern set proposal for the common communication patterns used in robotics. The following is a list with the five patterns included:

1. *send*: one-way RPC-like method invocation (a data structure can be sent);
2. *query*: two-way RPC-like method invocation (two data structures can be sent, one input-only, the other output-only);
3. *push newest*: generally known as publish/subscribe (a data structure is propagated simultaneously to multiple subscribers);
4. *push timed*: similar to 'push newest', but with a fixed frequency;
5. *event*: similar to 'push newest' but data-less (a signal).

The first two patterns (*send* and *query*) in the previous enumeration can be seen as special cases of RPC-like calls, which are supported by Ice, so they can be implemented easily. The third pattern, *push newest*, namely publish/subscribe, is also supported by Ice. Finally, the last

two patterns are special cases of *push newest*. In particular, *event* is equivalent to publishing a data-less topic, something that can be easily implemented using Ice by publishing a void structure. The only pattern that can not be directly implemented in the Ice middleware is the *push timed* pattern. However, it only requires 2-6 lines of code approximately (depending on the language) to implement. The *send*, *query* and *push newest* patterns are especially interesting: *send* and *query* because RPC-like calls are close to what programmers are used to face in object-oriented programming; *push newest* (or publish/subscribe) because it is very appropriate to fetch data while avoiding polling all the topics a component is subscribed to (*e.g.*, to fetch the images of a camera without constantly asking the component interfacing with the camera of a robot).

Being IDL-based is an important middleware feature —DDL-based³ at least—. As any other domain specific language, interfaces in this case, they are designed to ease the job of developers. In particular, IDL-based middlewares are able to automatically generate the necessary target language code to implement the skeleton of the described interface (data structures in the case of DDL-based middlewares).

Of course, efficiency is a very important topic. Component-oriented programming is supposed to improve the programming experience of complex distributed systems in many ways; however, slowing down system should not be a consequence. This involves not just computational but also network-use efficiency, so data marshaling should be as optimized as possible in size and speed. It is worth noting that efficiency influences the granularity of the modules robot software can be composed of: the more efficient the middleware is, the finer-grained modules can potentially be.

As introduced in chapter 2, a wide platform support, and more importantly, a wide language support are also very important features. There are many reasons why a developer would choose a specific programming language: some problems are better solved with specific programming languages; sometimes the component to build would benefit from an already-existing library which only have bindings for a specific language; or it might just be that the developer likes the language. This is one of the strengths of Ice.

4.2.2 Component model

Establishing a specific component model is a controversial topic. On one hand it restricts the internal structure of each component (of each component individually, not of the overall system), making some developers feel uncomfortable as the amount of imposed structure grows. On the other hand, it has several advantages that improve the efficiency of the developers once they become familiar with it. The first and most important, it is the basis for code auto-generation; of course, if there was not predefined structure there would be almost no code to generate. This is especially important because part of what is auto-generated is middleware-related code, which is generally of no interest to roboticists. Additionally, if the structure is reasonably useful — *i.e.*, it coincides with what developers would write to some extent — and is known by them, they know where to look and what to modify without having to write everything from scratch, enhancing the efficiency of the developers. Moreover, despite there might be a *default structure*, it is of no harm if it can be later modified (even if the advantages of code auto-generation are lost).

In RoboComp it was decided to find a trade-off between a freedom and the advantages

³DDL stands for Data Definition Language.

of a fixed structure. The solution is to provide a default structure that can be used for code auto-generation but that the user can change later in the event that it is not appropriate for the purposes of a component. Of course, if the structure is changed, and the code is regenerated it might not be compatible with the modified version. It is worth noting that after eight years and hundreds of components have been created, to our knowledge, only five of them which are tools with very special characteristics have required a different structure: RCManager, RCReplay, RCMonitor, RCControlPanel and RCInnerModelSimulator (see section 6).

The skeleton of RoboComp components (figure 4.1) was composed of three main elements up to early 2012: **a)** the server interfaces (generally one, but as many as the number of interfaces components should implement); **b)** the *Worker* and **c)** the proxies for communicating with other components. The *Worker* class is the responsible of implementing the core functionality of each component. The server interfaces are classes derived from Slice (Ice IDL) definition interfaces, which implement the services of the component. These classes generally work by interacting with the *Worker* class. The core class and the interfaces are implemented as threads in order to reduce as much as possible the time components performing remote calls wait⁴. Proxies are also of remarkable importance within a component. These, which are attributes of the *Worker*, are instances of auto-generated classes by the ICE Slice to CPP compiler that provide access to other components.

Components also use a configuration file where all the operational and communication parameters are specified. Using this configuration, the main program makes the necessary initializations to start the component and the worker class.

Additionally, components may implement a common interface called *CommonBehavior*. This interface provides access to the parameters and status of the component and allows changing some aspects of its behavior at run time (for example, the frequency of its processing loop). This interface can be used to dynamically analyze the status of a component, providing a means of determining wrong configurations or potential execution errors.

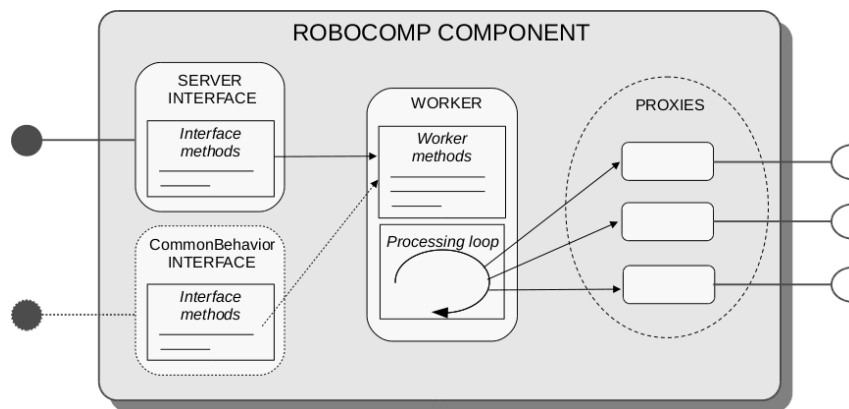


Figure 4.1: General structure of a RoboComp component.

After RoboComp embraced MDE (see section 3.4), a new challenge arose: if the developer needed to regenerate the code after modifying the component description, the changes made between its first and second code generation were lost. The solution adopted was to split the *Worker* class in two classes: the *GenericWorker* class, which is supposed to be modified only

⁴Otherwise, if a component is called whilst performing a considerable computation, the caller would have to wait until it finishes. This way, the caller can just receive the last results and it does not have to wait.

by the code generation tool, and *SpecificWorker*, which inherits from *GenericWorker* and is supposed to include manually written code. The code generation tool always modifies the *GenericWorker* class files depending on the component description. Nevertheless, the files for the *SpecificWorker* class are only created if they do not exist and are never modified if they exist. This way, RoboComp is able to regenerate the code of the components without deleting what developers have written.

4.2.3 Tools and classes

Besides of the component structure and the wide set of off-the-shelf working components, two of the aspects in which RoboComp extends Ice are the tools and the numerous set of classes it is equipped with. The set of tools is one of the main factors that influence user experience, making easier to develop and manage complex robotic systems.

The set of classes comprises different issues related to robotics and computer vision such as matrix computation, hardware access, Kalman filtering, graphical widgets, fuzzy logic or robot proprioception.

Among the different available classes, the robot proprioception class, which we call *InnerModel*, plays an especially important role in robotics software. It deals with robot body representation and geometric transformations between different reference frames. *InnerModel* is based on an XML description of the robot kinematics read from file. In this file, the transformations nodes (joints and links) are identified and described. *InnerModel* also provides different methods to estimate projections and frame transformations. Unlike other frameworks such as ROS [Quigley et al., 2009], RoboComp does not enforce the use of a central component for such functionality. The *InnerModel* class is a lightweight kinematics representation which can be replicated in different components (even more than once if necessary) according to the needs of each component. This decision was taken in order to reduce both, software complexity and latency. Indeed, the memory overhead of replicating *InnerModel* objects is negligible. Additionally, RoboComp components can also maintain a centralized representation. This is done using the *RCInnerModelSimulator* robotics simulator (RCIS for short, see section 6.7) using its *InnerModelManager* interface. Besides of the general features of a robotics simulator, *RCInnerModelSimulator* provides an easy-to-use API that allows to include, remove and modify nodes on-line and perform queries.

The tools, which are further explained in chapter 6, can be summarized as:

- *RCManager*, an easy-to-use tool to deploy and inspect the status of multiple components (see 6.1).
- *RCInnerModelEditor*, an editor XML-based kinematic trees (see 6.6).
- *RCInnerModelSimulator*, a robotics-oriented simulator (see 6.7).
- *RCReplay*, a tool to replay the output of RoboComp components (see 6.2).
- *RCLogger*, logs output and debug messages (see 6.4).
- *RCMonitor*, used to monitor the information that a component is working with (see 6.3).
- *RCDSLEditor*, the RoboComp's DSL-based code generation tool (see 6.5).

4.2.4 HAL characteristics

Component oriented programming frameworks for robotics provide an effective way to keep software complexity under control. However, some of them have ignored one of the most important contributions of Player [Gerkey et al., 2005]: the idea of a robotics hardware abstraction layer (HAL). In RoboComp this idea is considered extremely important for code reuse and portability. All sensors and actuators of the same type can provide a common interface as long as they do not have very specific features. There are three main reasons that make preferable —when possible— to use the same interface: **a)** different users, researchers, or companies can share their components regardless of the underlying hardware, **b)** it reduces the impact of hardware updates, **c)** it prevents software deprecation [Fitzpatrick et al., 2008].

Since it is possible to find hardware that does not fit the standard interface, RoboComp does not force the use of the proposed interfaces, just recommends it and emphasizes its benefits. Currently, the RoboComp hardware abstraction layer comprises the following interfaces: Bumper, Camera, DifferentialRobot, Fork, GazeControl, GPS, HeadNTP, HeadT2P, IMU, JointMotor, Joystick, Kinect, Laser, Laser3D, SoundCapture and RGBD. Table 4.1 provides a description of the previous interfaces. Additional interfaces can be easily added creating a new Slice IDL file.

Table 4.1: Middleware general aspects

| Interface | Description of what the interface provides |
|-------------------|----------------------------------------------------------------------------------------------|
| Bumper | State of a bumper |
| Camera | Stream of RGB and gray images |
| DifferentialRobot | Control a differential robot platform and access to its odometry |
| Fork | Control and access to the state of a robotic fork lift |
| GazeControl | Generic gaze control, independent of the particular head |
| HeadNTP | Interface for robotic heads with a neck, a common tilt and two independent pan movements |
| HeadT2P | Interface for robotic heads a common tilt and two independent pan movements |
| IMU | Information of accelerometers, gyroscopes and magnetometers |
| JointMotor | Control and access to joint motors |
| Joystick | Access to the state of joysticks |
| Kinect | Control and access to the motor and leds of the kinect sensor, as well as to its RGBD sensor |
| Laser | Access to laser range-finders |
| Laser3D | Access to rotating 3D lasers |
| SoundCapture | Capture audio signals |
| RGBD | Access to RGBD sensors |

4.3 RoboComp repository structure

In order to make RoboComp easy to understand it has been hierarchically organized in directories. The basic organization is shown in figure 4.2. The *Classes* and *Tools* directories contain several subdirectories with different classes and tools, respectively. *CMake* contains CMake⁵ feature files that can be used by components to easily include and link third-party libraries (e.g., OpenCV, IPP, OpenNI, OpenRave). *Interfaces* contains IDL files that define the different component interfaces that have been included in RoboComp. *Components* itself contains three directories:

- *HAL*, where the components of the hardware abstraction layer are located,
- *Essential*, where non-HAL components which are of general interest are located, such as *gmappingComp*, *joystickComp*, *gazeComp* or *robotTrajectoryComp*),
- additional subdirectories in which the components of different research groups —that might or might not be of interest to other users— are stored.

On the other hand, in order to make RoboComp easy to extend, it has been designed so that users do not necessarily have to follow this structure. New components can be compiled and distributed separately. This is possible thanks to two environment variables: *\$ROBOCOMP*, that must be configured to contain the path of the source code of RoboComp; and *\$SLICE-
PATH*, that should contain the different paths where interface files can be found.

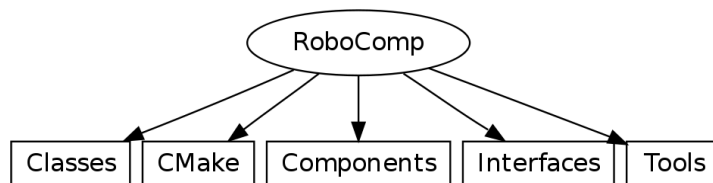


Figure 4.2: Directory hierarchy of RoboComp.

4.4 Framework comparison

Throughout this chapter, different robotics framework features have been presented and their benefits have been discussed. In order to provide a global view of the features and benefits of RoboComp, this section presents a comparison with the most relevant open-source frameworks to the date.

The middleware is one of the most important parts of a robotics framework. Different communication methods may be very interesting features in order to fit to the different inter-operational requirements and communication patterns. Depending on the task, robots may have to use a particular platform or programming language. Table 4.2 shows the license, supported platforms and programming languages of the different frameworks. Note that both RoboComp and Orca2 support a wide variety of possibilities derived from the use of the Ice middleware.

⁵CMake is an open-source software system used to generate build-projects and compilation scripts for multiple compilers and platforms.

Table 4.2: Middleware general aspects

| Framework | License | Supported Platforms | Programming Languages |
|-----------|----------|------------------------------------------|--------------------------------------------|
| Carmen | GPL | Linux Windows | C C++ Java Python |
| Marie | LGPL | Linux | C C++ |
| Miro | GPL | Linux | C C++ |
| MOOS | GPL | Linux Mac OS X | C++ Matlab |
| OpenRTM | EPL | Linux Windows FreeBSD | C++ Python Java |
| Orca2 | GPL/LGPL | Linux Windows Mac OS X Android iPhone | C++ C# Python PHP Ruby Java Objective-C |
| OROCOS | GPL/LGPL | Linux Windows | C++ |
| SmartSoft | GPL/LGPL | Linux Windows | C++ |
| RoboComp | GPL | Linux Windows Mac OS X Android iPhone | C++ C# Python PHP Ruby Java Objective-C |
| ROS | BSD | Linux Windows | C++ Python Octave Lisp Java |
| YARP | GPL | Linux Windows QNX | C++ Java Ruby Python Lisp |

Table 4.3 shows the middleware used by the different frameworks and some of its associated features. Those frameworks providing an IDL-based strongly typed interface are usually preferable: they are easier to understand and, therefore, reduce the possibility of programming errors. In RoboComp and Orca2, the Ice compilation tools are used in order to convert IDL interfaces into code of specific programming languages. Frameworks using CORBA IDL are also equipped with similar tools. The remaining frameworks do not include any IDL description facilities, although ROS does provide data and service description language support with its corresponding compilation tools. Regarding the communication methods, Ice-based frameworks provide a good variety of both synchronous and asynchronous communication mechanisms. This feature allows the programmer to select the most suitable component interaction method regarding to their operation requirements.

Due to the complexity of the problems to solve, the toolset is also one of the most remarkable characteristics of a robotics framework. Table 4.4 shows the tools provided by the reviewed frameworks. In the table, *Text* means that the framework provides a console-based tool, *gui* corresponds to a graphical user interface, and *no* to the unavailability of a tool. The columns of table 4.4 correspond to:

- '*Code gen.*': represents availability of a DSL-based code generation tool that abstracts users from middleware-related code and does not interfere with the code written by users.
- '*Simulator*': lists the simulators the different frameworks provide connection to.
- '*Manager*': represents the existence of an easy to use and specific tool to deploy and inspect the status of the deployed components (Orca2 uses IceGrid to deploy components but it is not framework-specific or user-friendly).
- '*Replay*': represent the availability of tools for component replaying.

Table 4.3: Networking and API

| Framework | Middleware | Comm. Methods | IDL |
|-----------|------------|---------------------------------------------------|-----------|
| Carmen | IPC | pub/sub query/response | No IDL |
| Marie | ACE | data-flows (socket-based) | No IDL |
| Miro | ACE+TAO | sync/async client/server | CORBA IDL |
| MOOS | custom | pub/sub | No IDL |
| OpenRTM | ACE | pub/sub query/response | CORBA IDL |
| Orca2 | Ice | RPC, AMI, AMD, pub/sub | Ice IDL |
| OROCOS | ACE | commands, events, methods, properties, data-ports | CORBA IDL |
| SmartSoft | ACE/CORBA | send, query, push newest, push timed, event | CORBA IDL |
| RoboComp | Ice DDS | RPC, AMI, AMD, pub/sub | Ice IDL |
| ROS | custom | pub/sub query/response | No IDL |
| YARP | ACE | asynchronous data-flows | No IDL |

- 'Log': corresponds to the availability of tools for logging messages.
- 'Monitor': existence of tools to monitor the information that a component is working with, not just its status (*i.e.*, images of a camera or the pose of a robot platform). Both ROS and RoboComp have such tool. However, *monitorComp* is more advanced than the tools ROS is equipped with: *rxbag* has a plugin system that allows users to display new types of data but it can only work with off-line data; *rxplot* works with on-line data but only with a few data types. *monitorComp* provides both features.

On the basis of this comparison, we think that RoboComp has a remarkable set of technical and user-oriented features that make it an outstanding robotics framework. However, we are aware of the big effort that would involve for new users to migrate their software from a framework to another. Thus, the on-line documentation of RoboComp describes the procedure to achieve inter-operation with Orca2 and ROS, two of the most widely used frameworks in robotics.

Table 4.4: Tools

| Framework | Code Gen. | Manager | Replay | Simulator | Log | Monitor |
|-----------|----------------|---------|--------|----------------------|------|---------|
| Carmen | no | no | gui | custom 2d | gui | no |
| Marie | no | no | no | stage gazebo | no | no |
| Miro | no | no | no | no | gui | no |
| MOOS | no | text | gui | uMVS | gui | no |
| OpenRTM | gui | gui | no | stage gazebo | gui | no |
| Orca2 | no | no | gui | stage gazebo | no | no |
| OROCOS | no | text | no | no | no | no |
| SmartSoft | gui | gui | no | stage | no | no |
| RoboComp | gui | gui | gui | stage gazebo rcis | gui | gui |
| ROS | partial (text) | text | gui | stage gazebo | gui | gui |
| YARP | no | gui | no | icubSim | text | no |

Chapter 5

RoboComp DSLs

As seen in chapter 4, RoboComp provides a wide set of tools that aim to reduce the errors and the required effort for performing everyday tasks such as component creation, modification, execution or monitoring. Some of the tools require a sophisticated configuration. For example, the tool used to automatically generate code, requires the description of the component to generate. In order to appropriately handle this and other issues we created several Domain-Specific Languages (DSLs) that are associated with some of the tools of the framework.

We identified six tasks as the most error-prone or time-consuming ones and have provided DSL-based tools to handle the issues:

1. Creation of new components.
2. Addition or removing proxies to existing components.
3. Addition or changing existing data types or methods in interface declarations.
4. Solving errors in configuration parameters that determine the run-time behavior of the components.
5. Deploying and monitoring networks of components.
6. Implementing and working with kinematic trees.

The remaining of the chapter describes the different DSLs that have been implemented. In particular, we propose four DSLs: CDSL, IDSL, PDSL, DDSL and InnerModelDSL. These DSLs help developers to quickly understand the structure of a components, design them or even modify and deploy them at any time in its life-cycle. The experiment described in chapter 12 provides a case study covering all of the proposed DSLs and quantifies the benefits of DSLs when developing components.

5.1 CDSL

Although there were some tools and scripts written to decrease the amount of time spent in writing and modifying RoboComp components, they lacked of the necessary generality to adapt to the continuous evolution of the component model. For instance, at some point, the model was modified so new components implemented a generic introspection interface. This interface is

served by an internal thread that, when started, reads and checks the configuration parameters and, afterwards, monitors the execution of the working thread. These changes in the model are difficult to incorporate in the component creation scripts if they are designed as simple template transformation processes. Moreover, once a component was created, introducing changes was even more difficult —specially when components with different versions of the component model coexisted in the system. All these issues can be easily tackled by separating the code that can be potentially generated automatically from the code produced by developers. This design choice makes possible to modify the generic properties of the components without interfering with what was manually changed since the creation of a component. Moreover, this isolation would facilitate the inclusion of new features that were previously not taken into account in already-generated component. The optional use of graphical interfaces, internal state-machines, auxiliary classes or third-party libraries are some of these features, but code separation also enables to easily include features that did not exist when the component was generated.

Regarding middleware, RoboComp provides both client/server and publish/subscribe communication models. In order to establish communication, components must perform different operations. For example, if a component is required to perform remote calls to other components, its code needs to: **a)** include the definition of the proxy classes corresponding to the interfaces it is going to connect to; **b)** read from the configuration file how to reach the remote component; **c)** create the proxy object using the previously read configuration; **d)** provide the proxy object to the classes that will be using it. Similar scenarios exist when providing new interfaces, subscribing or publishing new topics (see [Manso et al., 2010] for more details). Until RoboComp adopted model-driven engineering, components were automatically generated using a Python script with a small form. However, if the requirements of a component changed after its creation (a considerably common scenario), the changes in the code had to be manually performed.

✓ All these situations made it difficult to maintain more than a few components. To solve these problems, we have developed a DSL to create and modify component properties. This DSL is called **Component Description Specific Language** (CDSL) and allows users to create and maintain their component descriptions from a textual model. CDSL files contain information about communication parameters such as proxies, interfaces and topics used by the components, their dependencies with external classes and libraries, the optional support of Qt graphical interfaces, the programming language of the component, and an optional SCXML file path for embedding a state machine in the component.

✓ Figure 5.1 shows the development process to obtain the CDSL. First, a meta-model defining the CDSL entities and their relations is created using the Eclipse Modeling Framework (EMF). Then it is automatically translated to an Xtext grammar that is used to automatically generate a parser for the language (see [Romero-Garces et al., 2011] for technical details). Once the Xtext grammar is created, users can create their own CDSL models using the **RoboComp DSL Editor**, which performs code generation using M2T transformations. Because components need interfaces or topics to communicate with other components, CDSL files can import data types, topics and interfaces defined in IDSL models (as shown in section 5.2).

The source code of the components generated from CDSL can be divided in two parts: the specific and the generic (see Figure 5.3). The generic part contains the logic of interprocess communication, the general structure of the components (e.g., main program and threads, source directory structure, documentation rules or configuration parameters) and some introspection and self-monitoring capabilities. This generic functionality is implemented with ab-

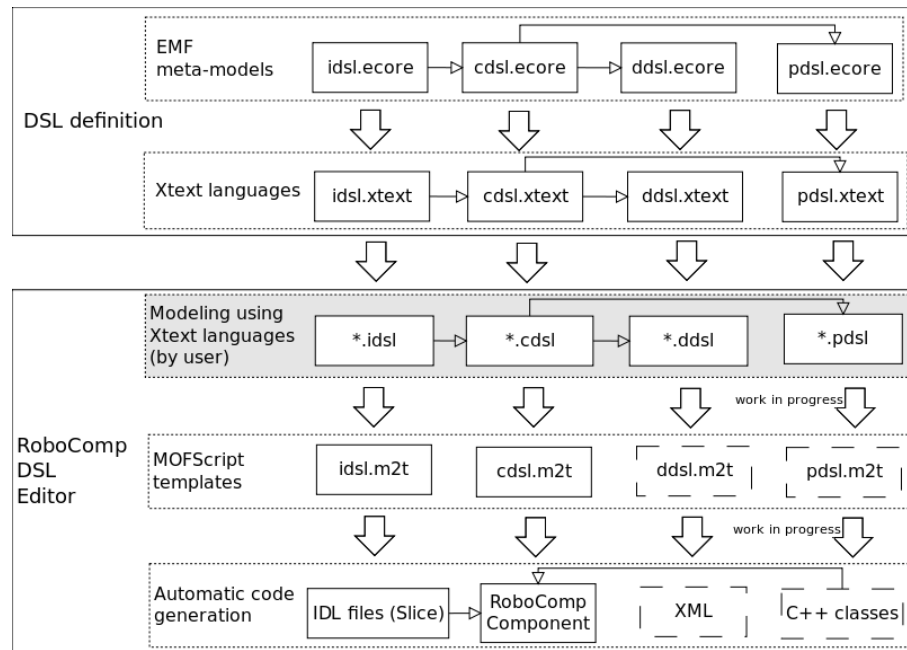


Figure 5.1: Development process of CDSL, DDSL, IDSL and PDSL

struct classes that are inherited and extended by the user-specific code to achieve the final working component. Thus, a component can be divided in two parts by a line separating the generic from the specific. The specific component code is generated by the RoboComp DSL Editor only the first time, but the generic code is always generated when regeneration from CDSL models occur. This way, users can be sure that their specific code will never be deleted and, at the same time, they are able to modify any component property. This organization is one of the most important design decisions. Figure 5.2 provides an example of the RoboComp DSL Editor while modifying a CDSL file.

The properties that are contemplated in CDSL are the following:

- Component name.
- Interfaces and data types defined in external IDSL files.
- Client/Server communication model: required and provided interfaces.
- Publish/Subscribe communication model: topics that the component will publish or subscribe to.
- Graphical interface support.
- State machine support.
- Dependences with external classes and libraries.
- Programming language of the component.

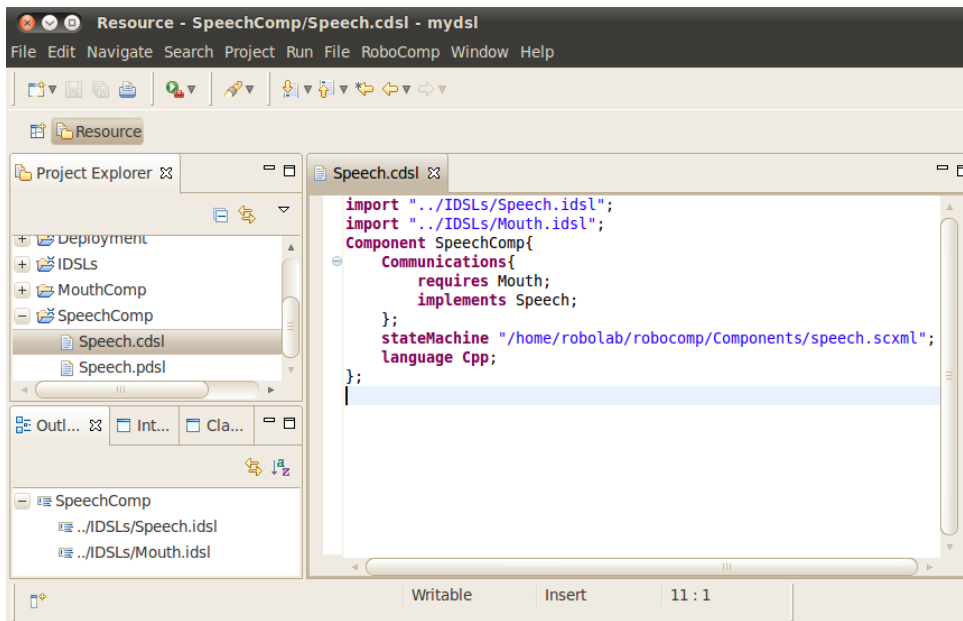


Figure 5.2: Screenshot of the RoboComp DSL Editor modifying a CDSL file.

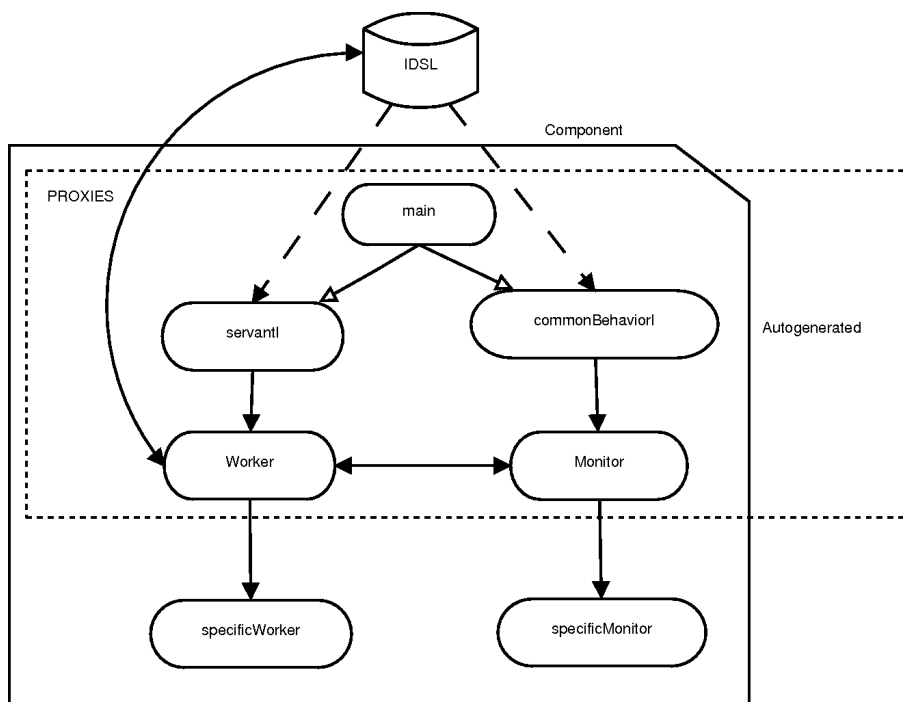


Figure 5.3: Structure of RoboComp components.

5.2 IDSL

Despite middleware-independence is underway [Manso et al., 2010], RoboComp currently uses Ice [Henning and Spruiell, 2007] as its main communication middleware¹. Our goal is to evolve RoboComp towards a middleware-independent framework, with several reference implementations. The first step in this direction is the definition of an IDL that can act as a bridge between the IDLs of the final middlewares and RoboComp. This new IDL provides a syntax for designing data types and procedures that can be safely used inside RoboComp code without relying on external dependencies of third-party providers.

Until the work towards middleware independence was started, RoboComp used Slice (the Ice Interface Definition Language, by ZeroC, see chapter 3) to define component interfaces. Unfortunately, this language is —of course— middleware-dependent (Ice), so, if RoboComp had to be integrated with another middlewares, all Slice files had to be adapted to them (or at least, a bridge should had to be implemented). Moreover, any change in the Slice language would require modifying the RoboComp interfaces already defined.

To solve these problems, we developed the *Interface Description Specific Language* (IDSL). IDSL has been initially designed as a subset of the Slice features, mainly data types and procedures definition, and avoiding interface inheritance, to facilitate future transformations among third party IDLs. Figure 5.1 shows the development process to obtain the IDSL, which is similar to CDSL. First the IDSL is defined using an EMF-based meta-model. Then the meta-model is imported to Xtext to generate the IDSL language. Users can define their own interfaces using the IDSL language, which is integrated in the RoboComp DSL Editor. Finally, they can generate the specific IDL using the code generator which is also integrated in the RoboComp DSL Editor. Figure 5.4 provides a screenshot of the RoboComp DSL Editor while modifying an IDSL file.

Currently, IDSL supports the following features:

- Interfaces and topics definition.
- Basic data types, such as integer and real numbers or strings.
- Enumerated types.
- Custom structures and data types such as sequences and maps.
- Exceptions.

5.3 DDSL

The last source of repetitive tasks and errors is related to the deployment of components. The two main reasons are wrong or inadequate configuration values (which are covered by PDSL in section 5.4) and unresolved dependencies found when deploying connected components. Despite component-oriented robotic systems can be seen as a whole, it is important to remember that they are always composed of independently executed programs that interact with each

¹This is because Ice is extremely robust, supports a rich variety of communication resources, including push/pull data-oriented mechanisms and has a nice license (GPL). The decision was made in 2005 and, fortunately, Ice has never represented a problem.

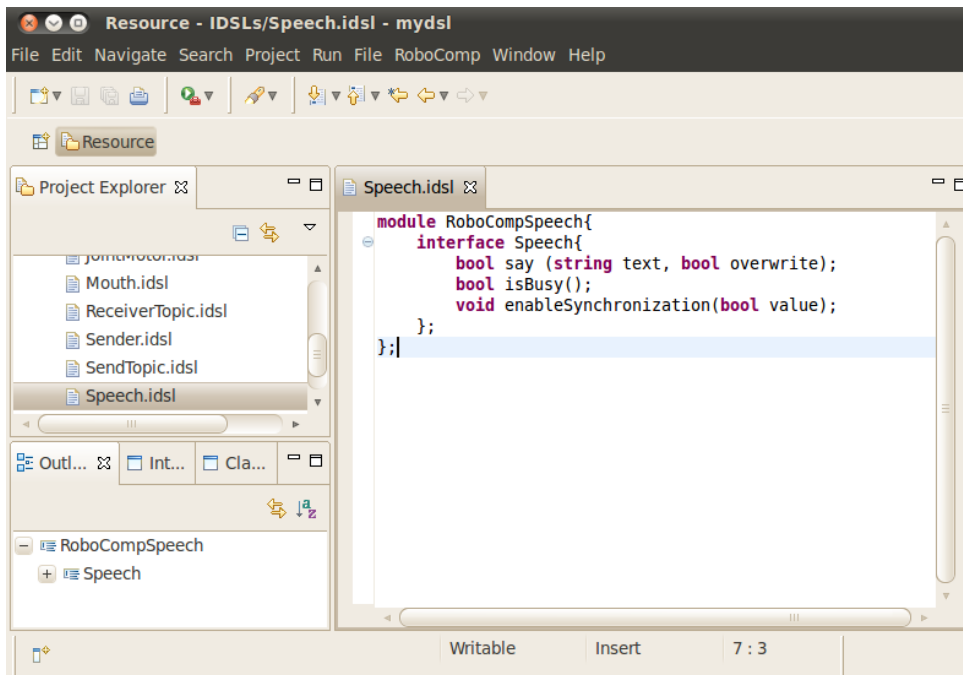


Figure 5.4: Screenshot of the RoboComp DSL Editor modifying a IDSL file.

other. These components can be executed manually, but as the number of components grows, it becomes increasingly difficult to manage them appropriately. A correct and safe deployment can be implemented in the appropriate application if it is provided a structured description language along the necessary tools to orderly execute the involved processes.

Robots of middle complexity (e.g., mobile robots equipped with stereo heads) and high complexity robots (e.g., mobile manipulators with expressive heads) are controlled by graphs containing dozens of components running on several computers. The configuration and management of these networks of processes suggest the combination of a graphical tool and a representation language. The Deployment Description Specific Language was developed as the underlying language to make this management task easier.

✓ With DDSL (***Deployment Domain-Specific Language***), users are able to describe which components will be used, where they should be executed and which configuration to use. This makes it possible to automatically deploy RoboComp components in a certain computer or computer network. DDSL has been designed to simplify the system deployment and integration. Figure 5.1 shows the development process of the DDSL. It is defined using an EMF-based meta-model which is translated to an Xtext grammar. Once the Xtext DDSL grammar is created, developers can create and manage their DDSL models using the RoboComp DSL Editor.

In order to define a component network, the following parameters have to be specified for each component:

- Component: the CDSL file path of the component to execute.
- Path to the executable file of the component.
- IP address and port.
- Path to the configuration file.

With the information in this file, all component dependences can be precomputed. This way, the DSL editor can warn users of basic configuration errors while editing the file and prior to the actual deployment.

5.4 PDSL

As introduced in the previous section, wrong or inadequate configuration values is one of the most important sources of errors when deploying components. The **Parameter Definition Specific Language** (PDSL) provides a generic structure for the configuration parameters that define the run-time behavior of the components. This DSL guides developers in writing the necessary configuration files in a standardized way within the framework. The file format that was previously used lacked of hierarchical structure, being just a list of *<attribute,value> pairs*. When a component required a nested relation of parameters, such as a list of lists, the component had to parse the corresponding configuration string. Figure 5.5 —particularly its last two lines— provides an example of this situation.

```

1 # Endpoint
2 JointMotorComp.Endpoints=tcp -p 10067
3 # Parameters
4 JointMotor.NumMotors=2
5 JointMotor.Handler = Dunkermotoren
6 JointMotor.Device = /dev/ttyUSB0
7 JointMotor.BaudRate = 115200
8 JointMotor.BasicPeriod = 220
9 # Motor: name,id,invertedSign,min,max,zero,vel
10 JointMotor.M0 = dunker0,A,true,-3.14,3.14,0,.9
11 JointMotor.M1 = dunker1,B,true,-1.7,1.7,0,.9

```

Figure 5.5: Example of the structureless configuration file format used previously by Robo-Comp.

With PDSL, the configuration parameters can be organized in nested lists and the editor can check that the introduced values are within the predefined ranges. After the code generation process, the component can access the configuration information by using methods defined in the generic classes. This way, the programmer can safely access and modify the configuration values, according to their predefined value ranges. Figure 5.1 shows the development process of the PDSL. The dotted lines show that it is work in progress.

Currently, PDSL supports the following parameter types:

- Component: the CDSL file path of the component to execute.
- Basic data types, such as integer, booleans and real numbers or strings.
- Enumerated types.
- List types and nested lists.
- Default values.
- Optional variables.

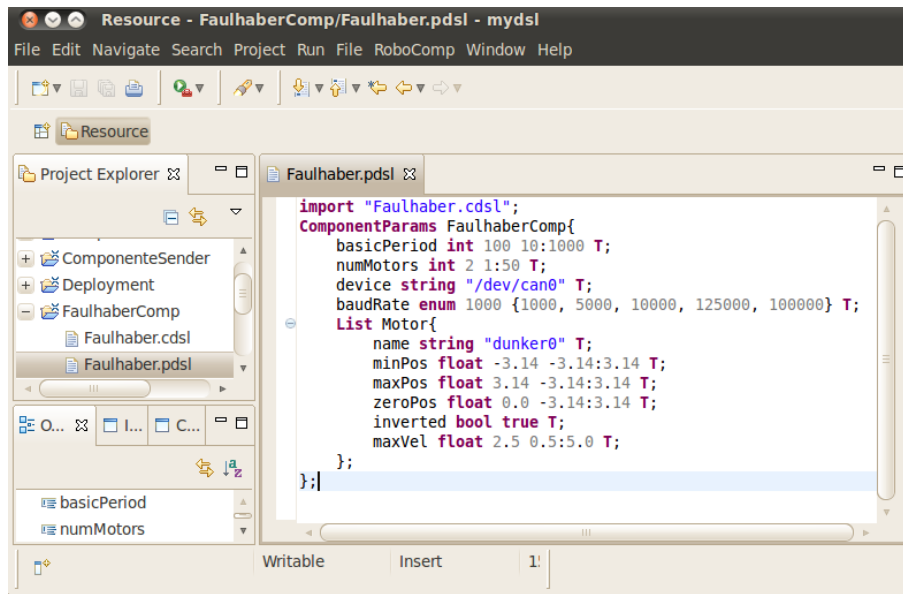


Figure 5.6: Definition of the parameters using the PDSL

- Range of possible values of basic and enumerated types.

5.5 InnerModelDSL / IMDSL

InnerModel is a tool designed to help solving some of the most common algebraic tasks in robotics such as frame of reference changes, computation of transformation matrices between arbitrary joints in the kinematic chain; also other not so common such as 3D point-to-image projection and back-projection. In order to perform these tasks it is necessary to describe the kinematic tree of any particular scene. Because programming this kind of algorithms is error-prone and inflexible we created **InnerModelDSL**, an XML-based language used to describe kinematic trees.

Each InnerModelDSL tag represents a node (InnerModelNode) in the kinematic tree that specifies the transformation that takes from its parent to the pose where the node is located. Specialized nodes provide specific features that make certain tasks easier (e.g., InnerModelLaserNode, InnerModelCameraNode). For example, the laser node allows to transform from laser polar coordinates to Cartesian coordinates and vice versa. Figure 5.7 shows a simple example of a kinematic tree.

InnerModel can then be used to answer the following: I know where the point x is from the point of view of the head but, where is it from the point of view of the hand?

```
innerModel->transform("hand", initialPosition, "head")
```

Of course, this kind of InnerModel call can be used with any pair of nodes.

The following is an IMDSL file with the example used in figure 5.7:

```
<innerModel>
  <transform id="world">
```

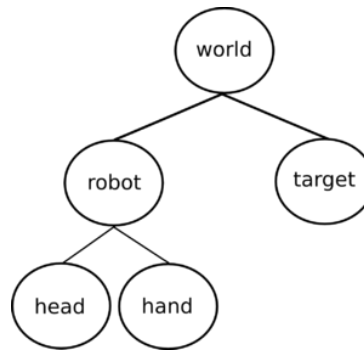


Figure 5.7: Two transformations are specified within the "world" reference frame (robot and target), each of those can also contain frame references.

```

<transform id="robot">
  <transform id="head" />
  <transform id="hand" />
</transform>
<transform id="target" />
</transform>
</innerModel>

```

Besides transformations tags and their simpler versions, translation and transforms, InnerModelDSL supports many additional tags. The following list describes the use and attributes of all the tags that InnerModelDSL currently supports (excluding the *id* field):

transform : Used to define three-dimensional transformations. Attributes:

- rx, ry, rz: Rotation.
- tx, ty, tz: Translation.

translation : Used to define three-dimensional translations. Attributes:

- tx, ty, tz: Translation.

rotation : Used to define three-dimensional rotations. Attributes:

- rx, ry, rz: Rotation.

differentialrobot : For differential robots.

- tx, ty, tz: Translation offset.
- rx, ry, rz: Rotation offset.
- port: Used by RCIS, the port where the simulated DifferentialRobot interface will be created.

laser : For two-dimensional lasers.

- min, max: Minimum and maximum distances.

- **measures:** Number of measures per reading.
- **angle:** Angle covered by the laser readings.
- **port:** Used by RCIS, the port where the simulated Laser interface will be created.
- **ifconfig:** Additional configuration parameters that RCIS needs to emulate the Laser interface.

camera : For regular cameras. It can be used with simulation purposes but also in order to perform queries to InnerModel.

- **width, height:** Width and height of the image that captures the corresponding camera.
- **focal:** Focal distance measured in pixels.

joint : For 1-axis rotation actuators. It can be used with simulation purposes but also in order to perform queries to InnerModel.

- **min, max:** Maximum and minimum angular positions of the actuator.
- **axis:** Axis in which the motor rotates.
- **port:** Used by RCIS, the port where the simulated JointMotor interface will be created.

imu : For inertial sensors. Useful in simulation.

- **port:** Used by RCIS, the port where the simulated IMU interface will be created.

rgbd : For RGBD cameras. It can be used with simulation purposes but also in order to perform queries to InnerModel.

- **width, height:** Width and height of the image that captures the corresponding camera.
- **focal:** Focal distance measured in pixels.
- **port:** Used by RCIS, the port where the simulated RGBD interface will be created.
- **ifconfig:** Additional configuration parameters that RCIS needs to emulate the RGBD interface.

mesh : Used along the InnerModelViewer class in order to visualize nodes.

- **file:** Path of the mesh file.
- **scale:** Scale of the mesh. The attribute must be a string containing one or three comma-separated floats (one if the scale is the same for all axes, three if it is desired to specify a different scale for each dimension).
- **render:** Whether “normal” for normal rendering or “wireframe” for wireframe rendering.
- **tx, ty, tz:** Optional translation offset.
- **rx, ry, rz:** Optional rotation offset.

pointcloud : Used along the InnerModelViewer class in order to visualize point clouds.

plane : Includes a plane in the kinematic tree. It can be used with simulation purposes but also in order to perform queries to InnerModel.

- **nx, ny, nz**: Normal vector of the plane.
- **px, py, pz**: A point that belong to the plane. In case the plane is rendered using the InnerModelViewer class, the plane is approximated to a box-shaped object.
- **texture**: Texture that the plane will be assigned if it is rendered using InnerModelViewer.
- **repeat**: The number of times the texture is repeated in the plane.
- **size**: Size of the box used when the plane is rendered using InnerModelViewer.

All translation, rotation, and offset attributes are optional and are assigned zero by default. For a wider explanation of InnerModel, InnerModelDSL and more examples, see the RoboComp's tutorial on InnerModel [[Manso, 2012](#)].

Chapter 6

RoboComp tools

A good robotics framework should not be just a bag of middleware features. Programming software for autonomous robots also involves other issues such as ease of use or readiness for an agile software development lifecycle. RoboComp is equipped with some of the necessary tools to complement the middleware layer and facilitate the work of the developers.

6.1 RCManger

Section 5.3 presented the main issue regarding component deployment: when the number of components in a network grows, manually deploying the whole network becomes time-consuming and frustrating. Along with DDSL, *RCManager* was developed aiming to minimize the problem (see screenshot in figure 6.1). ✓

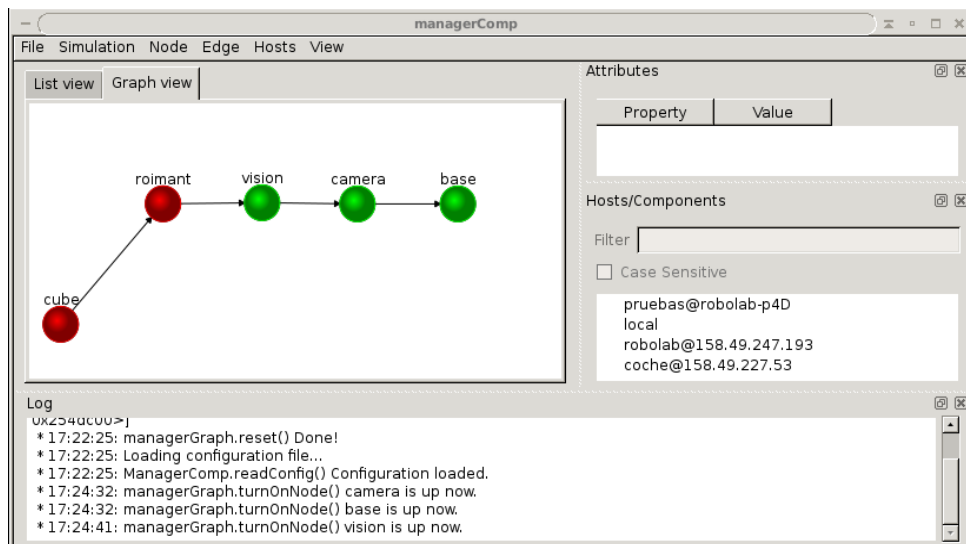


Figure 6.1: Graphical interface of the *RCManager* tool.

This tool allows users to graphically generate a deployment file (using DDSL) and run a system in an intuitive way: **a)** components located within a host list are automatically found; **b)** additional components are included in the system by using a simple drag&drop operation; **c)** component dependencies are specified drawing arrows connecting nodes of the system graph

(see Graph View tab of figure 6.1); **d**) components can be started/stopped by simply clicking the corresponding node of the graph view. This tool also facilitates system execution: when the user starts a component, all its dependencies are automatically satisfied. All these operations can be performed for components in both, local and remote hosts.

Moreover, *RCManager* can be used not only to manage local or remote components, but also to achieve certain level of introspection. As it was explained in section 4.2.2, components may optionally implement the common interface *CommonBehavior*. The manager tool uses this interface in order to give visual access to the parameters of the components. This is useful to detect wrong configurations or even to change operational attributes. Thanks to this feature, the manager tool can be used to help users diagnosing potential errors which would otherwise be harder to detect.

6.2 RCReplay

- ✓ The **RCReplay** tool (figure 6.2) records the output of a set of components in order to subsequently emulate their roles. During emulation, it can run at the desired speed or manually step by step. This feature is extremely useful for debugging purposes: if the inputs are the same, the behavior of programs should also be the same. Thus, it is very helpful when trying to reproduce errors.

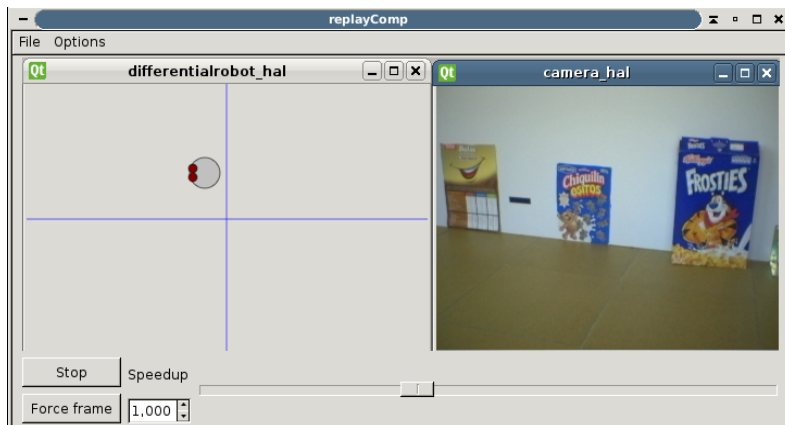


Figure 6.2: Graphical view of *RCReplay*.

This tool can run in two different modes: capture and replay. In capture mode, *RCReplay* can connect to different components and record their outputs. Its output file can then be transparently used in a replaying stage as input for other components (*i.e.*, components reading data from *RCReplay* will not be able to differentiate *RCReplay* from real components). This way, it enables software development when no robot is physically available.

Currently, *RCReplay* supports all the HAL component interfaces out of the box. Besides, the set of components that *RCReplay* can record and replay can be extended by writing small Python scripts that can be included in this tool as *plugins*.

6.3 RCMonitor and RCControlPanel

Developing new components involves the application of a test process to evaluate its operation. Without a specific tool, testing a component entails the creation of another component connecting to the new one in order to check its functionality. To facilitate this test process, RoboComp includes **RCMonitor**, a tool for component connection and monitoring. This tool allows the programmer either to include custom monitoring code or to use one of the templates available to test HAL components. ✓

RCMonitor provides a graphical interface that helps the programmer to carry out the component test process in an easy way. First, users describe how RCMonitor should connect to the corresponding component. The programmer has to indicate the endpoint of the component as well as its Slice definition file. Once the connection data is introduced, the next step is the insertion of the monitoring code. In order to facilitate this, the application uses the introspection capabilities of Python to show a list of the available symbols and remote methods. The language used for writing this code is also Python. It allows to minimize the size and the complexity of the testing code. Once the test code is written, developers can introduce all this information in a file so that RCMonitor can run tests without any user input.

As RCMonitor, **RCControlPanel** is also a tool for debugging and monitoring, however, there are three main differences: **a)** it supports working with more than one component concurrently; **b)** instead of being plugin-based, RCControlPanel is extended by including new *test classes* at compile time and **c)** plugins are programmed in C++. Therefore, it is generally used when we need to monitor more than one component at the same time and when Python is too slow. On the other hand, the development of monitorization code is not as simple and fast as with RCMonitor. ✓

6.4 RCLogger

This tool is an optional standard output alternative designed to analyze the execution of components and their interactions. It provides a graphical interface to display the information of interest, allowing the user to filter by: type, component, time stamp, source code file and line of code. In order to use **RCLogger**, components only have to include the RCLogger proxy (which can be automatically done using RCDSLEditor) and use the provided helper class *qLog* (which can be found in the RoboComp *Classes/qLob* subdirectory). ✓

Debugging the interaction of multiple components is generally complicated because the independent execution of components as different processes makes hard to serialize the output messages. Moreover, it is not comfortable to have as many open terminals as components to monitor. RCLogger provides the output messages in a centralized and serialized fashion.

6.5 RCDSLEditor

RCDSLEditor is a tool that enables developers to generate components using a concise yet powerful domain-specific language. Both the language (CDSL) and the tool were introduced in chapter 5.

6.6 RCInnerModelEditor

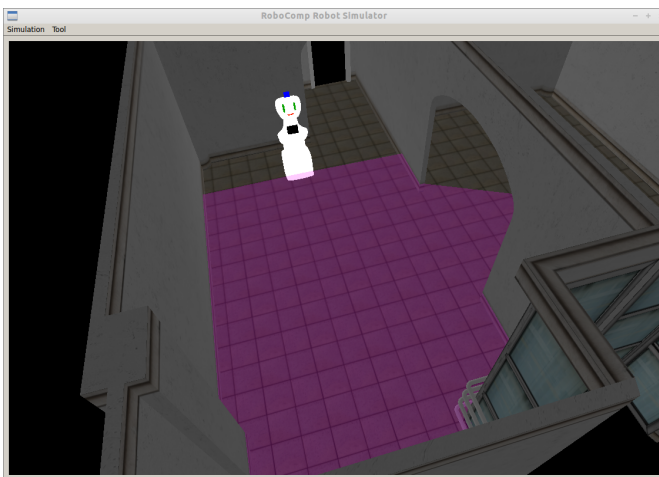
RCInnerModelEditor is a graphical tool designed to create, edit and view InnerModel DSL files (see section 5.5). Its ability to display the described kinematic trees in real-time allows users to spot errors in the descriptions easily.

6.7 RCInnerModelSimulator

Robotics simulators are extremely helpful tools that simulate virtual robots and their surrounding environments, providing access to the sensors and actuators of the simulated robots. Such access is transparent for the software using the robot in all current simulators. This way they can be used not only to assess how robots would behave in real life or to overcome the lack of access to one. In fact, during this research work these have only been seen as secondary applications of simulators. Here, the main advantage of using a simulator is the possibility to avoid the limitations of real hardware and control to what extent they behave as in real life. This control entails two interesting use cases for simulators: its use to test algorithms and to study how they would behave in different sensor and actuator noise conditions.

The first use case is achieved by providing perfect sensors and actuators: no noise and infinitely fast response. Given such setup, system failures can only be result of errors in the algorithms or their implementations. Thus, this is extremely useful in the first stages of development of new algorithms.

The second use case is achieved by enabling roboticists to control the error the simulator introduces in the simulated world and in the measurements it provides. By running the same algorithms with different error values in the simulator, researchers can provide error graphs indicating how robust algorithms are to noise. It also make possible to reproduce research results by providing the software and the simulated world the experiments were run with.



(a) Screenshot of the RCIS main window.



(b) Screenshot of a subjective camera.

Figure 6.3: Screenshots of RCInnerModelSimulator (RCIS).

Despite there already existed several open-source free simulators, none of them provided these features. The Player/Stage project [Gerkey et al., 2003] provides two simulators: Stage, a lightweight 2D simulator designed to work with multiple robots concurrently; and Gazebo, an

advanced simulator that aims to simulate the real behavior of robots. Despite Gazebo is a very interesting simulator (in fact, it has been integrated to work with RoboComp) to the date it does not allow to modify the error in the actuators directly (only by setting different configurations to the underlying physics engine) and does not introduce error in sensors at all¹. These were some the most important reasons why *RCInnerModelSimulator* (RCIS) was created. An interesting characteristic of the RCInnerModelSimulator is that it uses the same InnerModelDSL domain-specific language that is used to specify kinematic chains. Using the same InnerModelDSL file for simulation and geometric purposes guarantees that the kinematic chain is properly calibrated according to the actual (virtual) robot. ✓

6.8 RoboComp components

RoboComp provides a huge number of components, not only to access the hardware but also to process images, laser and RGBD data, SLAM and visual attention. This section describes some of the RoboComp interfaces used in this thesis, and the components that implement them.

DifferentialRobot

The DifferentialRobot interface provides access to differential robot platforms. It is implemented by the following components:

- robexComp: Controls the RobEx platform [Mateos et al., 2010].
- gazeboComp: Controls simulated differential platforms in the Gazebo simulator.
- smarComp: Controls the Smar robot in differential mode.
- RCIS: Controls simulated differential platforms in the RCIS simulator.

JointMotor

The JointMotor interface provides access to 1-dimensional rotation actuators. It is implemented by the following components:

- dynamixelComp: Used for Dynamixel servos.
- dunkermotorenComp: Used for Dunker Motoren motors.
- boshComp: Used for controlling Bosh motors.
- faulhaberComp: Used for controlling Faulhaber motors.
- RCIS: Controls simulated 1-dimensional rotation actuators in the RCIS simulator.

¹The development team of Gazebo is currently working toward these features.

Camera

The Camera interface provides access to RGB and gray cameras. It is implemented by the following components:

- `v4linuxComp`: For the cameras supported by V4L in GNU/Linux.
- `pointgreyComp`: For Point Grey cameras.
- `prosilicaComp`: For Prosilica cameras.
- `RCIS`: For simulated cameras in the RCIS simulator.

IMU

The IMU interface provides access to inertial measurement units. It is implemented by the following components:

- `imuComp`: For XSens inertial measurement units.
- `razorComp`: For SparcFun Razor inertial measurement units.
- `phidgetspatialComp`: For Phidget Spatial inertial measurement units.
- `RCIS`: For simulated inertial measurement units in the RCIS simulator.

RGBD

The RGBD interface provides access to RGBD cameras. It is implemented by the following components:

- `kinectComp`: For the Microsoft's Kinect sensor.
- `PrimeSenseComp`: For Asus Xtion Pro live and other OpenNI-supported RGBD cameras.
- `RCIS`: For accessing simulated RGBD cameras in the RCIS simulator.

LocalNavigator

The LocalNavigator interface is used for local navigation in two-dimensional environments. Currently it is only implemented by `vfhLocalNavigatorComp`:

CloudPrimitives

The CloudPrimitives interface is one of the core components used in the experiments described in sections 14, 17 and 18. It is used in order to detect and model planes in the environment given an point cloud stream.

Part II

Active Perception

Chapter 7

Environment modeling

7.1 Introduction

In order to achieve the tasks robots are supposed to perform they generally need to maintain information about the environment, their location and proprioception, and the plans used to achieve their goals. This topic has been widely discussed since the very early years of artificial intelligence and robotics [Brooks, 1991, Gibson, 1986, Miller et al., 1986, Arkin, 1998]. Two opposite points of view can be distinguished regarding this issue: reactive and deliberative robotics. Unlike *deliberative robots* —those whose control systems always act according to a well-defined plan, whether it is predefined or computed on-line— the control of pure *reactive robots* are based on *instinctive* actions only, they do not explicitly store any information about their surroundings¹. However, they are able to perform simple tasks such as obstacle avoidance or even trivial forms of manipulation [Brooks et al., 1989]. Using the principles of *swarm robotics* (i.e., complex robotic systems composed of a big number of simple robots) they have also proved to be able to achieve some non-trivial goals in a collaborative fashion such as exploration [Sheng et al., 2006], network deployment [Ulam and Arkin, 2004, Meng et al., 2006], or object clustering [Müller and Rodriguez, 1996]². However, their lack of memory prevents them from performing complex tasks autonomously, specially those that require planning. In order to have robots able to perform complex tasks such as social interaction, household chores or elder assistance efficiently, they must have access to a large amount of data, not only but including information about their environment. Since the environments are seldom known a priori (and in such cases they will probably need to be updated), robots need the ability to autonomously gather information about the world their surroundings.

This chapter provides an introduction to how robots can create models of their surroundings. Section 7.2 describes the most common choices make when building models: using a metric, a topological approach, or using a hybrid one instead. After section 7.2, in order to

¹Despite it is very common to distinguish between reactive and deliberative robots, these are opposite approaches which are seldom found. The most common option is to take advantage of both points of view and adopt some intermediate solution. *Hybrid robots* combine behavior-based robotics —those whose control activates and deactivates simple behaviors but not necessarily in an instinctive way (see [Arkin, 1998])— with the usage of internal representations and planning.

²Robot swarms can be seen as non-reactive systems since, despite they are composed of reactive robots, the swarm as a whole can modify the global behavior depending on the robots it is composed of. Thus, it can be considered that swarms can use their robots as memory storage. For example, leaving one robot of the swarm in a particular place could mean that the swarm has already explored that particular area.

understand the limitations of deterministic approaches, it is advised to read experiment 13: a preliminary experiment in which —using stereo vision— a robot models an environment composed of several connected rooms with obstacles in them. The conclusions of the experiment lead us to section 7.3, which introduces probabilistic representations (as opposed to deterministic ones), their advantages and why probabilistic models are generally preferred. In order to understand the probabilistic approaches used throughout this thesis, section 7.4 provides the necessary introductory concepts. Finally, the Bayes filter and its limitations are presented in section 7.5.

7.2 World model structures

Assuming robots need to perform complex tasks, environment representations can be considered something a robot would need. The next step is to decide the amount and nature of the information to store, and how to gather and update such information. The answers to these questions are not always easy and generally depend on the tasks to achieve. For some tasks, such as path planning or surveillance, spatial occupancy or navigability representations suffice, while other tasks might require additional information (*i.e.*, robots might need to know about object categories, materials, or any other physical or semantic property). In some cases it is preferable to have dense representations such as **occupancy grids** [Thrun, 2002]: arrays of cells representing the occupancy of contiguous regions of the physical world (usually small two or three-dimensional cubes, see figure 7.1). Frequently, more compact representations might be needed in order to improve efficiency (*e.g.*, planning using plain occupancy grids is rather slow, so its use is not recommended in large maps).

World models representing the environment geometrically, using quantitative information, are called **metric representations**. Metric representations consist of records of the geometry of the environment *as it is* in a flat organization (regardless of the underlying data structure, which can be optimized for faster access or smaller memory consumption). The most common metric approach is to use the previously mentioned occupancy grids, of two or three-dimensional cells. Despite not as common as occupancy grids, there exist many other different approaches to represent occupancy information, for example: line segments; solid geometry (which represents occupancy combining boolean operations with primitive objects); or planar and three-dimensional convex hulls [Angelopoulou et al., 1992]. Another interesting feature-based geometric representation is described in [Busquets et al., 2003], where hyper-graphs are used to store the information. The novelty of the approach is that the edges of the hyper-graphs (which are 4-uniform) encode the geometric relationship between four features so that knowing the relative position of three of them, the position of the remaining feature can be estimated.

The main drawbacks of metric approaches are that can only represent binary maps (*e.g.*, occupancy, navigability) and that —despite relatively fast algorithms have been proposed (see RRT [LaValle and Kuffner, 2001, Rodriguez et al., 2008])— depending on the size of the grid cells and the environment, planning can take a considerable time that might exceed usability limitations. These maps can also be extended by associating cells with other type of information, but they would not be pure metric maps anymore.

Several criticisms have been made to the view of perception as creating exact literal internal replicas of the world, even to claiming this *copying* process is necessary for regular robot or human activity [Brooks, 1991, Noë, 2004]. Replicating the world internally as it is does not

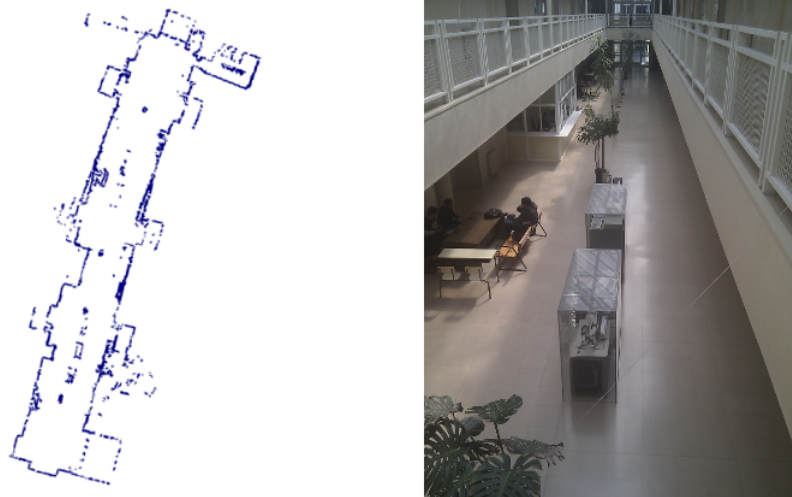


Figure 7.1: In the left hand side, an example of the most typical metric map model, a two-dimensional grid: blue means occupied, white means free. In the right hand side, a picture of the mapped area.

necessarily provides any benefits besides shorter times accessing the information stored in it, if any. As many authors have pointed out, since the world can be measured, it can be considered its most accurate representation [Brooks, 1991]. Such “representation” is available for any organism or machine, but it does not make intelligent all of them, it does not make all of them understand the structure of the environment or the physical and semantic properties of the objects on it. This is widely known by psychologists as the “little man in the brain” problem [Gibson, 1986]: if the information of the environment is accessible to anyone, how does duplicating the information inside our brains would make us intelligent? Is there a little man inside our brains? In fact, seeing robot perception as creating an unstructured replicas of the environment inside the memory of the robots leaves unanswered most perception and intelligence-related questions, such as how to use such representations. Therefore, the goal of roboticists should not be to create extremely accurate maps, but to create useful representations to enable robots succeed in their tasks. Creating really useful representation and perception mechanisms is the most important goal of this thesis.

On the other side of the barricade are *topological representations* : graphs which replicate ✓ the configuration of the real world regarding one or more properties. Unlike pure metric maps, topological ones use qualitative information and allow us to represent structure. Generally, the elements of the graphs represent concepts of higher abstraction than occupancy *e.g.*, visual or surface features or structured world elements such as rooms or doors. Their edges represent relationships such as visibility or reachability [Matarić, 1990, Angelopoulou et al., 1992] (see figure 7.2). Despite the semantics associated with these maps are more general and can thus model a broader set of properties, pure topological maps lack of proper mechanisms to represent geometric information, so achieving a precise localization is harder than using metric maps. This drawback makes pure topological representations inappropriate for most of the tasks robots are nowadays supposed to perform. However, this organization can be very helpful for solving other kind of tasks in an efficient way, such as global navigation [Martínez et al., 2011, Martínez and Caputo, 2011]. Thus, they are generally used in conjunction with metric information.

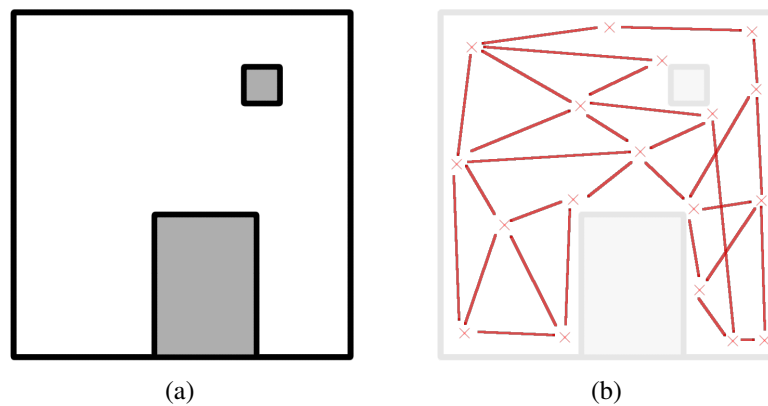


Figure 7.2: Figure 7.2.a shows an example of an indoor environment. Figure 7.2 shows a valid (yet incomplete) visibility map for the environment where crosses are recognizable features of the environment (not necessarily corners, appearance-related features are also acceptable) and edges are used to indicate that the locations corresponding to two different features are visible from each other's location.

Following a hybrid approach —representing both metric and topological information— is the solution followed by most state-of-the-art robots and one of the most promising research directions (especially for those supposed to perform moderately complex tasks in environments of considerable size). **hybrid representation** are models that do not fit in the previous categories because they combine metric with topological information. They are usually implemented whether as a metric model endowed with symbolic information (*e.g.*, as a grid accompanied with a list of objects associated with cells), or vice versa, a topological map where each node is associated with a local metric map. The term **symbolic representation** (or symbolic map) is used to denote maps that contain symbols (*i.e.*, entities which can be identified by a name), such as topological or hybrid maps.

✓ **Simultaneous localization and mapping** (SLAM) algorithms —those whose purpose is to simultaneously create maps of the environment, canceling out odometry errors on-line as the robots move— have succeeded creating accurate 2D occupancy grid maps of medium-sized areas. These representations are now easily built, however, they are also slow for path planning on large areas and have limited expressive power. As a consequence, hybrid maps with interconnected local occupancy grids have become popular. Several approaches on mobile robotics propose the use of hybrid representations to complement the metric information of the environment. In [Thrun, 1998] it is proposed to create off-line graphs by partitioning metric maps into regions separated by narrow passages. In [Simhon and Dudek, 1998] the environment is represented by a hybrid topological-metric map composed by a set of local metric maps called islands of reliability. In [Tomatis et al., 2003] the authors describe the environment using a global topological map that associates places which are metrically represented by infinite lines belonging to the same places. [van Zwynsvoorde et al., 2000] constructs a hybrid representation as a route graph using Voronoi diagrams. In [Yan et al., 2006] the environment is represented by a graph whose nodes are crossings (corners or intersections). [Montijano and Sagues, 2009] organizes the information of the environment in a graph of planar regions. In [Wolf and Sukhatme, 2008] grid maps are enhanced with navigation information: how appropriate cells are for easy navigation and how often dynamic objects occupy cells.

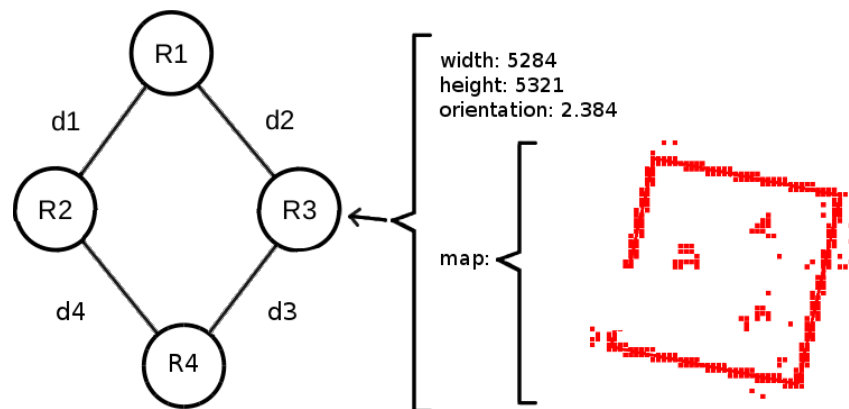


Figure 7.3: Example of the proposed world model. Maps are topologically structured (as can be seen in the graph on the left hand side) but, for each world element metric information is also available (in the right hand side, for the node 'R3'). In the particular case of the picture there are two types of metric information: first, a model-based rectangle model dependent and secondly, a grid map that allows to avoid obstacles. The first one, on the other hand, is of a higher level and can be used for a first low-detailed planning and for HRI, to specify places.

Despite is generally less relevant, another valid classification of the representations of the environment is as absolute or relative representations. In *absolute representations* the environment is static and the robot changes its position within the map when moving. On the other hand, in *relative representations*, the robot is set as the origin of the coordinate system (or the root node in case of topological maps).

This thesis proposes a hybrid approach, similar to the ones in [Thrun, 1998] or [Simhon and Dudek, 1998] in the sense that the resulting structures are graphs in which elements are connected according to topological relationships, resembling the structure of the environment. However, as seen in the experiment presented in section 13, the maps are built on-line, and instead of determining node's boundaries using conditions (such as the narrow passages of [Thrun, 1998]), it is followed a model-based approach for room modeling: robots actively try to match the elements of the environment with the models they know. For example, the resulting graphs may have symbols for rooms, adjacent if and only if there is a door connecting both rooms in the real world.

In comparison with pure metric maps, the usage of graph-like structures combined with model-based mapping do not only make planning much more efficient (because of the reduction of the size of the problem), they also make the map building process easier. On the other hand, model-based representations do not provide enough flexibility or the information to perform local path planning when there are world entities for which there is no known model. For example, if the robot can only represent rooms and doors, it could plan a trajectory which would make the robot collide in case there is an obstacle. To overcome these limitations, model-based information is combined with non-model-based, metric information, enhancing the flexibility of the robots and allowing to perform hierarchical planning with different levels of detail. Figure 7.3 provides an example of this.

A particularly interesting property of planning using these structures is that the resulting plans can be —using the terminology of computer functional programming— *lazy-evaluated*. For example, it is possible to conceive navigation planning as two-tiered: first, it is needed to

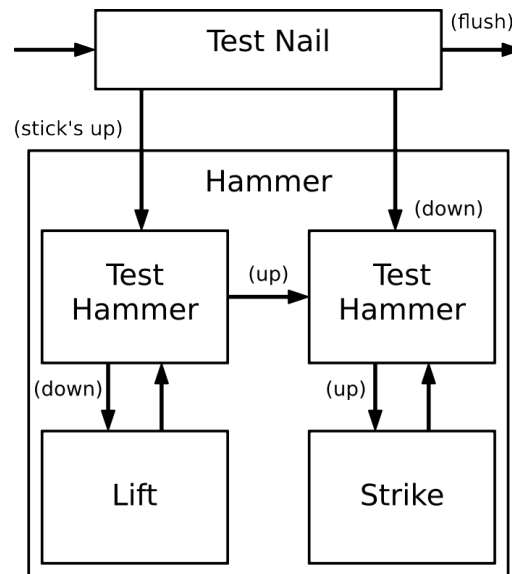


Figure 7.4: Example of how TOTE units can be used to express hierarchical plans. In this case, the hammering of a nail. There is a main plan composed of two sub-plans (which could potentially be composed of several plans).

decide which rooms to traverse, then, the particular path robots must follow to go to the next room (or the target if it is in the same room as the robot). This way, robots only need to create a detailed plan when needed and only in the context of a particular room.

Despite not all theories of philosophy of mind are necessarily applicable to robots, hierarchical partially-evaluated plans are supported by psychologists. The TOTE units presented in [Miller et al., 1986, Chapter 2] as a model of how plans could be created and used in humans are an example of it. The representation proposed here could be used by the TOTE units of [Miller et al., 1986]. TOTE stands for “*Test + Operate + Test + Exit*” and is a theoretical strategy for describing and executing hierarchical plans. Each unit is composed by a test and an operation which should be executed until the test is evaluated as true. TOTEs can be arranged sequentially and recursively, that is, the operation of a TOTE unit can be a sequence of TOTE units. Figure 7.4 provides an example of how plans made of TOTE units would look like.

At this point the reader might want to read chapter 13, which describes an experiment demonstrating how a robot can create hybrid maps combining model-based with occupancy grid representations. The limitations of the solution proposed in it will guide us to how can they be improved.

7.3 Deterministic versus probabilistic modeling

Regardless of whether the nature of the model used is pure metric, topological, or somewhere in the middle, the model and the algorithms used to update it can be classified whether as **probabilistic** or **deterministic**. **Probabilistic representations** are those taking into account the uncertainty of the models they hold. They provide a probability density function (PDF) covering every possible model (usually numeric approximations). The data that robots gather is used to update the PDF properly. Their use is generally less complicated, the model corresponding to

the maximum a posteriori distribution—the one maximizing the probability of the model being correct given the data acquired—is selected and used as if it was the *true* model. However, more sophisticated mechanisms evaluating risks also exist. Particularly, probabilistic modeling is appropriate in order to implement decision-theoretic planning in environments where people or robots can be harmed, or there exists any other kind of risk. For example, in [Thrun, 2000] it is presented a mechanism to learn arbitrary functions that take as input a PDF—risk is just an example of this.

On the other hand, approaches using *deterministic representations* always take into account a unique model which is considered the *true model*. The main difference is that, instead of defining a PDF, deterministic approaches define an algorithm for obtaining what model is chosen. In fact, deterministic models can be seen as special cases of probabilistic ones where their probability density function is concentrated in a single point (*i.e.*, using a Dirac delta function as its PDF). Therefore, since probabilistic modeling provides more information, leaving efficiency considerations aside, probabilistic models are usually preferred. Deterministic approaches only maintain a single hypothesis about the environment. On the other hand, probabilistic approaches, besides keeping track of the probability density function of the model, can also maintain an arbitrarily large number of different key hypothesis concurrently [Grisetti et al., 2006, Zhu et al., 2000] (see section 8.2.1.4). Despite it is considerably slower, maintaining different world hypothesis—using a PDF with multiple modes—makes robots much more robust, enabling them to recover from errors and ambiguities (at the expense of an increased complexity). Of course, this advantage is limited by the number of hypothesis (modes) maintained and by the nature of the model (despite due to the complexity of the world it is extremely rare, there might be scenarios in which a multi-mode PDF is not necessary). ✓

All these properties make worth working with probabilistic models. The next sections introduce the basic concepts needed to understand the problem of probabilistic modeling and control (section 7.4); the most important related algorithm, the Bayes filter (section 7.5); and one of the most simple—yet useful—implementations of a Bayes filter, the histogram filter (section 7.6).

7.4 Probabilistic modeling and control

Building models of the environment is a probabilistic inference problem where the variable to infer is composed of the state of the robot and the environment. In this work it is used the same notation used in [Thrun et al., 2005]: the variable to infer—the one that is used to describe the possible models—is usually called *state variable* (or state for short) and is denoted by X ; the variable measured using the sensors of the robots is denoted using the Z letter. The space in which the state variable lays is called *belief space*. The set of actions performed by the robots, if they play an active role, is denoted by U (*i.e.*, there are contexts in which robots might take no actions). U is not considered a stochastic variable because the actions performed by the robot are known, what is considered unknown is the consequences of the actions in the environment³. ✓

A state is said to be *complete* (or *Markovian*), if the world dynamics only depend on the latest state and action (x_{t-1} and u_{t-1} , respectively). The concept can be easily understood ✓

³It would also be possible to model *action consequences* as an unknown variable, but in that case it would be necessary to introduce a deterministic complementary variable containing the *intentions* of the robot. The initial setup is used because this setup would only increment the complexity without providing any advantage.

considering a ball tracking system where the state is only composed of the ball's position⁴. Given the current position of the ball, we can not estimate where the ball will be in the future because we do not know its velocity. However, if we include in the state the velocity, using Newtonian physics we can estimate the position and velocity of the ball at the very next moment. Thus, assuming the possible errors caused by any uncertainty sources (*i.e.*, elements which are not correctly modeled or not modeled at all) have small impact on the state, we can *assume* that the state is complete.

Now, assuming the opposite —that the state is not complete— given a specific point in time t , the inference process could potentially depend on all the previous known data: the actions of the robot (if any) and the measurements gathered from the environment, that is:

- the ordered set of all previous actions taken: $u_{1:t}$ and
- the ordered set of all measurements: $z_{1:t}$.

✓ Inference must be updated each time new information is available, modifying the probability density function of the state variable (also called **belief distribution**, or *belief* for short) according to the data. The belief at time t is denoted by $bel_t(x)$. The previously mentioned variables (*i.e.*, the measurements and previous actions) might condition the current state (x_t), which is what it is intended to infer. Thus, the belief is defined as:

$$bel_t(x_t) = p(x_t | z_{1:t}, u_{1:t}) \quad (7.1)$$

It must be noted that it is assumed that robots measure before perceiving or acting. That explains why the set of actions ranges from 1 to t , because u_t refers to the action the robot makes just before the environment transitions from x_{t-1} to x_t . This inference process can be graphically described using the Bayesian network shown in figure 7.5. Figure 7.5 only shows five time steps, it is important to understand that the corresponding Bayesian network grows indefinitely as the time passes. Therefore, trying to perform inference like this becomes unmanageable with time.

As introduced in section 7.3, modeling can be seen as the maximization of the probability of the state given all the information that might condition the current state (*i.e.*, finding the most likely state). It can thus be formulated as a maximum a posteriori estimation. Without any particular assumption, the *most likely* model (\hat{x}_{t_n}) can be mathematically expressed as:

$$\hat{x}_t = \underset{i}{\operatorname{argmax}} p(x_{t_i} | z_{1:t}, u_{1:t}) \quad (7.2)$$

✓ Now, assuming that the **state is complete**, most of the dependences can be removed from equations 7.1 and 7.2 as long as the previous state (x_{t-1}) is introduced. This context is graphically represented in figure 7.6 and coincides with the definition of complete state. Using the Bayesian network of figure 7.6 we can shorten equations 7.1 and 7.2. Equations 7.3 and 7.4 show their simplified versions after assuming a complete state, removing the dependences from previous measurements, and all the actions and states that came before the last ones. Unless explicitly noted, from now on it will be assumed that we will always be working with complete

⁴Assume we are located in a perfectly shaped spherical planet, that its physical properties and those of the ball are known, and that no objects can interfere with the ball besides the planet itself.

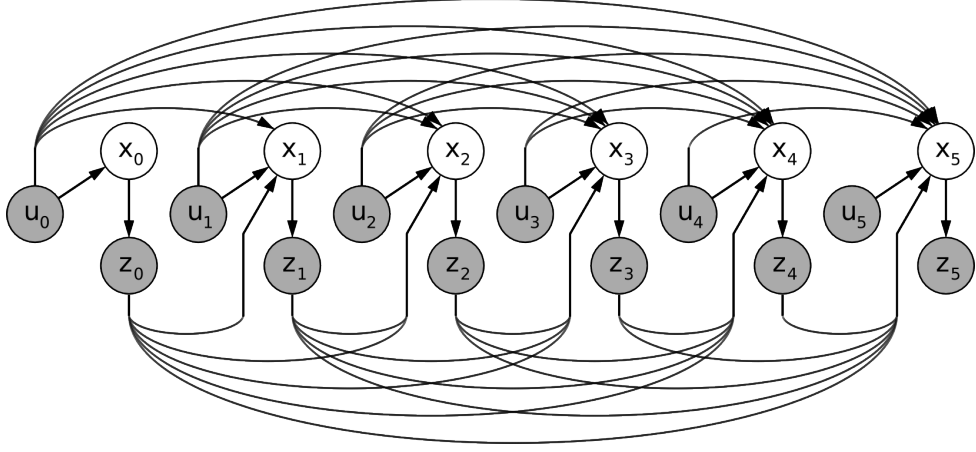


Figure 7.5: Graphical representation of the Bayesian network described for the general case of probabilistic state inference. Each state depends on all the measurements, and the previous actions. This model makes computing the probability of the state x_t complex and slow, since the network grows indefinitely with time.

states. If the state is complete, and thus it only depends on the last state and action, the state variable is said to form a **Markov chain** (note how figure 7.6 resembles a chain). ✓

$$bel_t(x_t) = p(x_t | x_{t-1}, z_t, u_t) \quad (7.3)$$

$$\hat{x}_t = \max_i p(x_{t_i} | x_{t-1}, z_t, u_t) \quad (7.4)$$

Obviously, this last situation is desirable because equations 7.3 and 7.4 are much simpler and

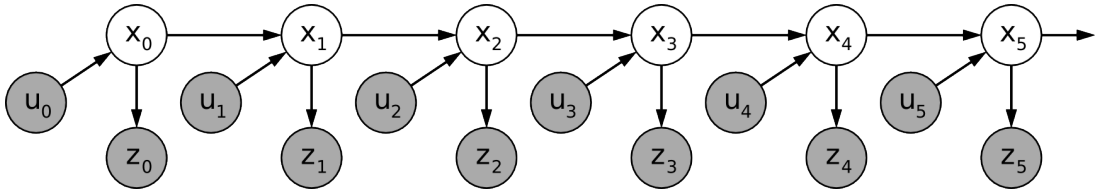


Figure 7.6: Graphical representation of a dynamic Bayes network, or Markov chain. Including from state x_0 to x_5 . Each state only depends on its previous state (x_{t-1}) and the last measurement, z_t (which is exactly the definition of a Markov model). It is worth noting that, regardless of the time step, the pattern of every x_t node is constant (see figure 7.7).

faster to compute than equations 7.1 and 7.2⁵.

To sum up, under Markovian assumptions it is possible to update the belief of the current state using equation 7.3 if it is known how to calculate the corresponding probability density function. The next section explains the Bayesian filter, a mathematical tool to recursively update the belief of complete states given the previous belief and the last action and measurement.

⁵Note that for $t = N$ equation 7.1 involves $2N$ variables, while equation 7.3 (which assumes a complete state) only two.

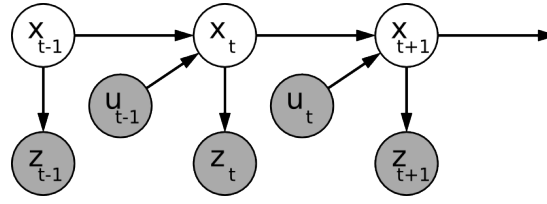


Figure 7.7: Short graphical representation of a dynamic Bayes network. The state x_t only depends on its previous state (x_{t-1}) and the last measurement (z_t).

7.5 Bayes filter

Introduction

In the context of on-line inference, a filter is a mathematical construct that allows to perform and update a belief state as new data is available. The aim of the Bayes filter is to provide a means to estimate a variable X — $bel_t(x_t)$ — according to equation 7.3. The probability of the model
 ✓ can be computed (or at least approximated) using a **Bayesian filter**, updating the knowledge about the world as new actions and measurements are provided, under two assumptions:

a) the world state is complete

This is a reasonable assumption when the state includes all or almost all the information needed to *predict* the environment in the very next moment. In fact, even if the state actually depends on any constant number of the last states (*e.g.*, besides the last action and measurement, x_t might depend on $x_{t-3:t-1}$), it is possible to redefine the state as the conjunction of those last states. This could be used in the *ball tracking* scenario previously described in this chapter in order to take into account the acceleration (in case the acceleration is not included in the state to begin with).

b) $p(x_{t-1}|z_{1:t-1}, u_{1:t}) \approx p(x_{t-1}|z_{1:t-1}, u_{1:t-1})$

Or equivalently (assuming a complete state): $p(x_t|z_t, u_t, u_{t+1}) \approx p(x_t|z_t, u_t)$. This is true when the last action is loosely coupled with the representation: that is, the robot does not perform actions, are constant, or selected at random. Also, it is acceptable when the actions of the robot can not be predicted with the last model of the environment (for example, if the robot needs a stable model in order to decide) or when isolated actions do not significantly modify the environment (which is usually the case).

7.5.1 Complementary concepts

The Bayes filter is supposed to be used in a Markovian context. However, there are two new general concepts that are necessary in order to understand the Bayes filter: the state transition function and the measurement probability. Actions modify the probability density function
 ✓ of the state variable according to a **state transition probability function**, which assigns a probability to each possible state transition when a specific action is executed. Given a previous state, an action and a possible next state, the function returns the probability of switching from x_{t-1} to x_t after executing action u_t (which might of course lead to remain in the same position). It is mathematically expressed as:

$$p(x_t|x_{t-1}, u_t) \tag{7.5}$$

The measurement process also influences the belief distribution. In equation 7.6 it is shown the **measurement probability**, that is, the probability of a particular measurement conditioned to a particular state:

$$p(z_t|x_t) \quad (7.6)$$

As introduced in 7.4, a **belief distribution** is the probability distribution of the state variable at a specific time, according to the data available. However, depending on the context, they might have different properties. Aside of the **initial belief distribution**, which is the initial probability density function, before gathering any data (hence generally very little informative), there are two interesting belief distributions: the so called predicted and corrected belief distributions. The **predicted belief distribution**, $\overline{bel}_t(x_t)$, is the belief before incorporating the last measurement but containing the expectations of the robot after executing a specific command. It can be used to decide what command to execute (by maximizing or minimizing a function taking such belief as input). It is defined as:

$$\overline{bel}_t(x_t) = p(x_t|z_{1:t-1}, u_{1:t}) \quad (7.7)$$

The **corrected belief distribution**, $bel_t(x_t)$, is the belief after incorporating the last measurement. Calculating it from the prediction is called **correction**. Generally, unless explicitly stated otherwise, the term *belief distribution* refers to the *corrected belief distribution*. It was already defined in section 7.4 as:

$$bel_t(x_t) = p(x_t|z_{1:t}, u_{1:t}) \quad (7.8)$$

7.5.2 The algorithm and its mathematical derivation

It is now possible to describe the Bayes filter. It is actually a family of similar algorithms, since, as we will see, each particular implementation depends on the properties of the environment and how it is modeled.

Simply put, the Bayes filters computes the belief of a particular state x_t (up to a scale factor) as the result of multiplying the likelihood of the data obtained in the last update (z_t) by the prior probability of being in such particular state in time t given the previous knowledge of the state and the latest action ($p(x_t|x_{t-1}, u_t)$). However, it is not evident how to compute this last probability, which coincides with the definition of predicted belief: $\overline{bel}_t(x_t)$. This is done by computing $\overline{bel}_t(x_t)$ as the probability of transitioning from any other possible previous state to x_t given the last action taken (using the theorem of total probability). By performing this with all the possible states, the filter can compute the belief distribution $bel_t(x_t)$ for any value of x_t .

The general steps of the filter are described in algorithm 1 using a mathematical notation. The algorithm has a *for* loop that computes a the probability for every possible state. The first line of the loop calculates the predicted belief and the second calculates the corrected belief given the predicted one. It does so up to a scale factor (η) which, depending on the purpose and it is usually the case, can be safely ignored (the actual PDFs can also be computed by normalizing them afterwards). One of the most typical cases in which the scale factor can be ignored is when looking for the location of maxima of the PDFs: since the scale factor is common to every state (*i.e.*, it only depends on the measurements and actions, not on the states) it remains constant, so the maxima is not affected.

Algorithm 1 Bayes filter as defined in [Thrun et al., 2005]:

Require: bel_{t-1} : The *corrected* belief at time $t - 1$.

Require: u_t : The last action.

Require: z_t : The last measurement.

```

1: forall  $x_t$  do
2:    $\overline{bel}_t(x_t) = \int p(x_t|x_{t-1}, u_t) bel_{t-1}(x_t) dx_{t-1}$ 
3:    $bel_t(x_t) = \eta p(z_t|x_t) \overline{bel}_t(x_t)$ 
4: end forall
5: return  $bel_t$ 

```

The demonstration is straightforward. Applying Bayes to equation 7.8, we can get:

$$bel_t(x_t) = \frac{p(z_t|x_t, z_{1:t-1}, u_{1:t}) p(x_t|z_{1:t-1}, u_{1:t})}{p(z_t|z_{1:t-1}, u_{1:t})} \quad (7.9)$$

Given that Bayesian filters assume that the state is complete, equation 7.9 can be rewritten as:

$$bel_t(x_t) = \frac{p(z_t|x_t) p(x_t|z_{1:t-1}, u_{1:t})}{p(z_t|z_{1:t-1}, u_{1:t})} \quad (7.10)$$

Substituting the denominator by a constant, since it is common to all states:

$$bel_t(x_t) = \eta p(z_t|x_t) p(x_t|z_{1:t-1}, u_{1:t}) . \quad (7.11)$$

The following is the tricky part: using the theorem of total probability with the variable x_{t-1} , the third factor of equation 7.11 can be rewritten as:

$$p(x_t|z_{1:t-1}, u_{1:t}) = \int p(x_t|x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1}|z_{1:t-1}, u_{1:t}) dx_{t-1} \quad (7.12)$$

which, using the Markovian assumption in the first probability inside the integral, and assuming that the last control, u_t , does not condition the previous state, x_{t-1} (see assumption *b*) at the beginning of the section), can be rewritten as:

$$p(x_t|z_{1:t-1}, u_{1:t}) = \int p(x_t|x_{t-1}, u_t) p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx_{t-1} \quad (7.13)$$

Including the right hand side of equation 7.13 back in 7.11:

$$bel_t(x_t) = \eta p(z_t|x_t) \int p(x_t|x_{t-1}, u_t) p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx_{t-1} \quad (7.14)$$

As the reader might notice, the second probability within the integral of equation 7.14 matches up with the definition of $bel_{t-1}(x_{t-1})$. Making such substitution we arrive to line 2 of algorithm 1 expanded in the third one, that is:

$$bel_t(x_t) = \eta p(z_t|x_t) \int p(x_t|x_{t-1}, u_t) bel_{t-1}(x_{t-1}) dx_{t-1} \quad (7.15)$$

From algorithm 1 and equation 7.15 it can be deduced that, in order to implement a Bayesian filter, three functions are needed:

1. the transition function: $p(x_t|x_{t-1}, u_t)$,
2. the state-conditioned measurement probability: $p(z_t|x_t)$,
3. and, by induction, the belief at time 0, that is, the initial belief: bel_0 .

Unfortunately, Bayesian filters can only be analytically implemented for very specific cases. In order to do so, the integral of line number 2 of algorithm 1 must be solved. The integral contains two factors:

1. $p(x_t|x_{t-1}, u_t)$, the state transition function, and
2. $bel_{t-1}(x_{t-1})$, which is recursively computed as the result of the previous update of the algorithm.

The challenge here is that —since it is computed recursively— $bel_{t-1}(x_{t-1})$ must always have the same distribution in order to ensure that the integral can always be correctly computed the same way. This only happens with very specific combinations of PDFs of the involved probabilities (*i.e.*, $p(x_t|x_{t-1}, u_t)$, $bel_t(x_t)$ and $p(z_t|x_t)$), mainly because robots usually need to maintain distributions of an unknown number of modes. Due to the difficulty of performing such integrals, Bayesian filters are generally unknown in the case of continuous belief spaces (also, in practice, in the case of very large discrete spaces). The measurement probability function ($p(z_t|x_t)$) usually has many modes because ambiguities entail an unknown number of local maxima in the measurement probability function. The belief tends to have multiple modes because it is computed as the multiplication of the prediction measurement probability function (which has generally multiple modes) with the prediction ($\overline{bel}_t(x_t)$). Since there are no parametric functions with indefinitely many modes to integrate (and the environments where robots move are complex enough to need such feature), robots generally relay on non-parametric approximations.

Only if the transition function is the result of adding normal noise to a linear function of u_t and the measurement probability is also normal, the inference is solved analytically by the Kalman filter (see the original [Kalman, 1960] or the introductory technical report [Welch and Bishop, 1995]). The Extended Kalman filter and the Unscented Kalman filter are extensions of the original Kalman filter to allow non-linear transitions. However, since these can only be used in very specific cases, they are not covered in the text (see [Thrun et al., 2005] for more information).

The most straightforward way to approximate a Bayesian filter is to produce a density function based on a manageable set of samples. The next section provides a intuitive method to perform such approximation, the histogram filter.

7.6 Histogram filter

The *histogram filter* is a discretized version of the general Bayes filter. In order to overcome the problem of computing the integral of the predicted belief distribution using analytical functions, the method uses histograms for representing beliefs. Since integrals in discretized spaces are actually summations, such integrals can be computed numerically, regardless of the shape of the underlying PDF. ✓

Its pseudo-code is shown in algorithm 2. As the reader might notice, the code is almost the same as the code of the general Bayes filter, the only significant changes are the summations that substitute the integrals. The summation can still be done even if one of the factors is a continuous function, so $p(z_t|x_t)$ and $p(x_t|x_{t-1}, u_t)$ are still treated as continuous functions (this is more general and potentially reduces discretization error when possible). If these functions must necessarily be discrete, the filter can still be used with no problems.

Initially, the η variable is generally ignored, however —if the histogram is required to reproduce a PDF— the bias caused by this can be corrected by normalizing the histogram bins to sum up to 1.

Algorithm 2 Histogram filter:

Require: bel_{t-1} : The *corrected* belief at time $t - 1$.

Require: u_t : The last action.

Require: z_t : The last measurement.

```

1: forall  $x_t$  do
2:    $\overline{bel}_t[x_t] = \sum_{x_i} p(x_t|x_i, u_t) bel_{t-1}[x_i]$ 
3:    $bel_t[x_t] = \eta p(z_t|x_t) \overline{bel}_t[x_t]$ 
4: end forall
5: return  $bel_t$ 

```

The histogram filter can be used as a workaround for continuous spaces where the integral can not be computed (*e.g.*, because of a dynamic number of modes) or as an exact Bayesian filter in discrete spaces as long as the histograms have as many bins as the actual belief space (it would still be an approximation otherwise).

Using histogram filters is a very easy task as long as the initial belief (as a histogram in this case) and the two probability density functions needed by the algorithm are provided (*i.e.*, the measurement and state transition PDFs).

Example

In order to achieve a better understanding of how histogram filters and, in general, Bayesian filters work, we will design a histogram filter for a theoretical robot. This robot lives in a 1-dimensional alike environment. It always faces walls looking directly to their closest points and can only move forwards. It measures distance using a range device which returns the distance to the closest point in its front (see figure 7.8). It is known that the maximum distance between the walls is five meters (it is worth noting that histogram filters can not be used with unbounded distributions because the computation of the summation would be intractable⁶). The objective is to model the distance to the wall it is currently facing.

In order to design a histogram filter one only need to define three elements:

1. The initial belief: \overline{bel}_0

⁶Despite unbounded distributions can not be used, it is possible to implement the histogram filter with dynamically changing finite bounds.

2. How to compute $p(x_t|x_{t-1}, u_t)$
3. How to compute $p(z_t|x_t)$

Designing the functions that model the uncertainty of the sensor and the dynamics of the model from the control is the main challenge when implementing these techniques. Since it depends on the specific problem it will be assumed that the functions were provided by a third party. An example of marginalization of the measurement probability density function is shown in figure 7.8.a. An example of marginalization of the state transition probability density function is also shown in figure 7.8.b: when the robot moves forward u_t it is expected that the wall will get closer u_t , but with a random distribution (normal in this case) due to the uncertainty of the execution of the actions (*e.g.*, slippery floors, mis-calibrated odometry). For the initial belief distribution, we will use a uniform distribution (non-informative). Given these three elements, the histogram filter can be easily implemented.

Figures 7.9, 7.10 and 7.11 show three iterations of an execution of the filter in which the robot is turned on, process a measurement and then starts moving forwards.

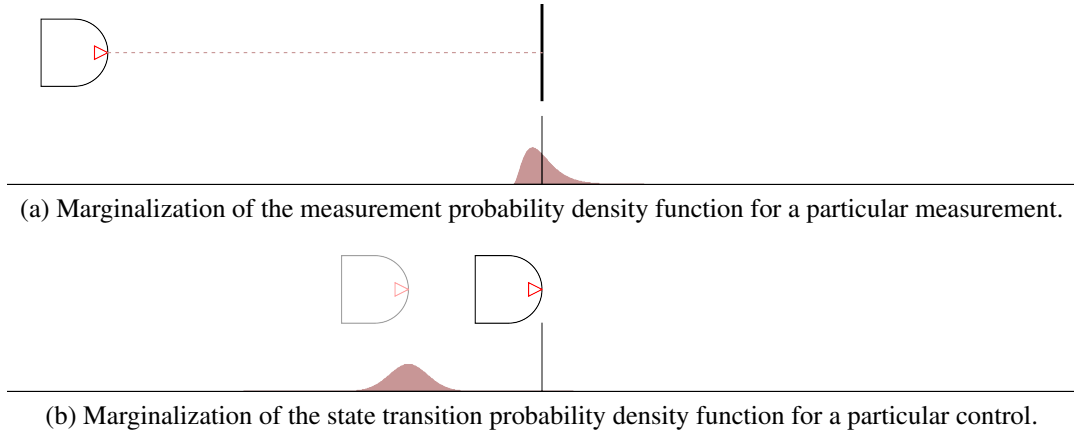
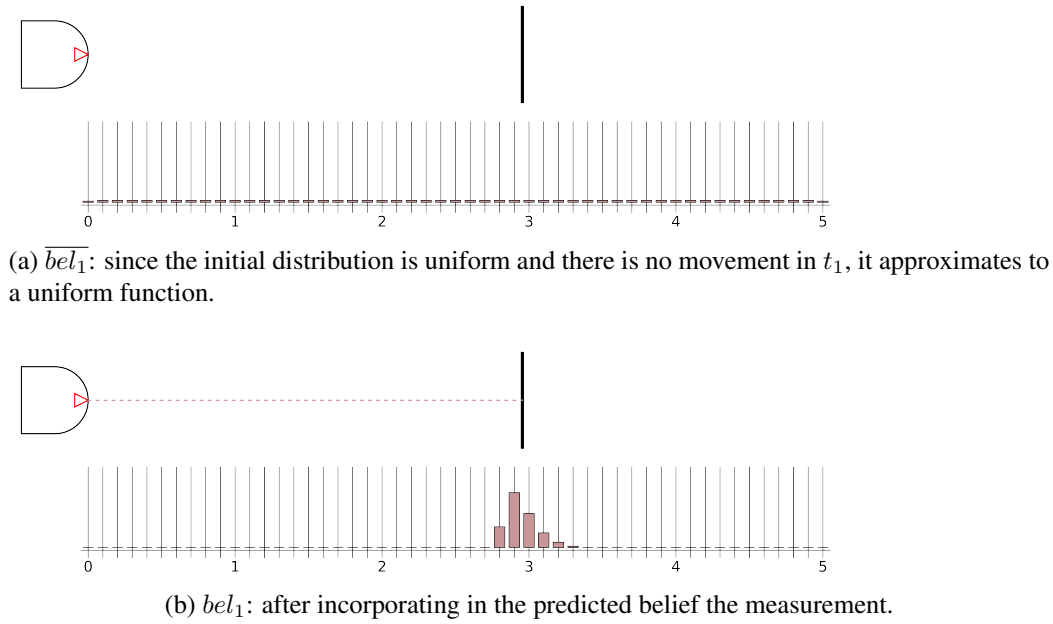


Figure 7.8: a) Marginalizations of the measurement probability and state transition density functions. They model how the errors of the measurements and controls are distributed.

Figures 7.9.a and 7.9.b show the predicted and corrected beliefs for t_1 . As previously mentioned, the initial belief was uniform. Since the robot does not move in the first iteration, the predicted belief is approximately —not exactly— uniform. The corrected belief shows the influence of the measurements in it.

Figures 7.10.a and 7.10.b show the predicted and corrected beliefs for t_2 . The predicted belief of figure 7.10.a reflects how the movement modifies the previous belief (before processing z_2). Despite the function generating the state transition PDF introduces normal noise, it is important to remark that \overline{bel}_2 is not a normal distribution: since each of its bins is the result of the convolution of $p(x_2|x_1, u_2)$ and bel_1 , it is slightly skewed to the right. The corrected belief \overline{bel}_2 is more concentrated than in \overline{bel}_1 because its corresponding prediction also is more concentrated.

Figures 7.11.a and 7.11.b show the predicted and corrected beliefs for t_3 . This iteration is similar to the previous one, the main difference being that the variance of both distribution are smaller.

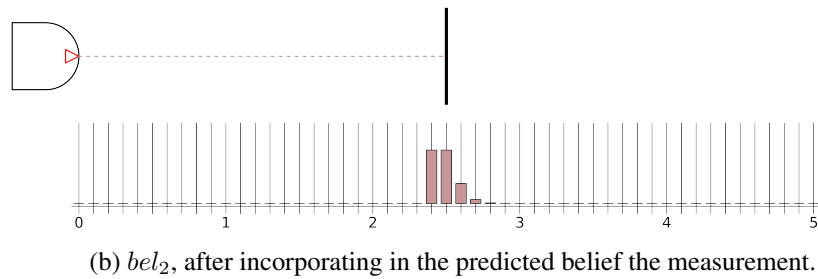
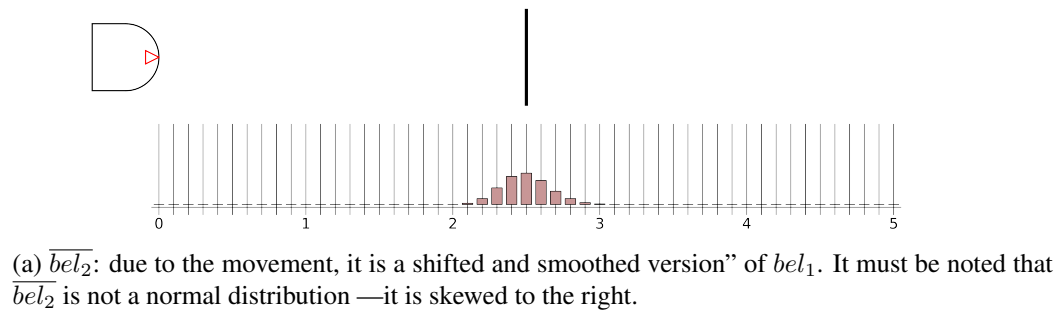
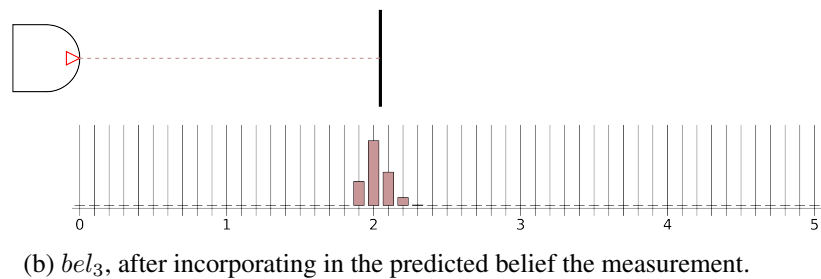
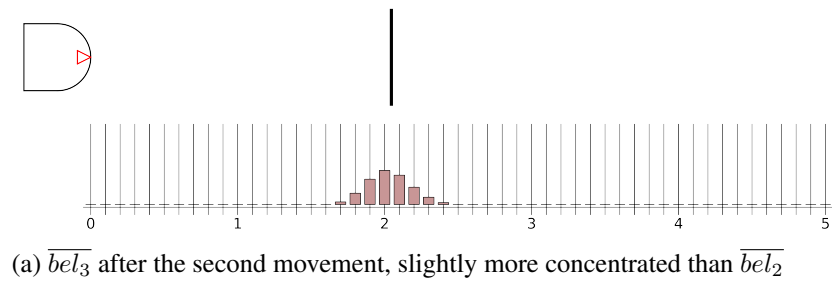
Figure 7.9: Beliefs for t_1 .

Discussion

The only topic left unexplained is how to build the procedures that compute $p(x_t|x_{t-1}, u_t)$ and $p(z_t|x_t)$. Despite being a topic that depends on the application, these procedures heavily condition the performance of Bayesian filters. In order to generate appropriate probabilities, the transition probability should model how likely is to get from the state x_{t-1} to the state x_t after executing action u_t ; the measurement probability should provide how likely would be to measurement z_t if the state was x_t . Unfortunately, there is no mathematical way to simplify the problem, it is the duty of roboticists to model these functions as good as possible.

The implementation of the histogram filter is easy, but its specific requirements limit its applicability dramatically. It perfectly models a posteriori PDFs as long as the necessary assumptions are met: it is used for discrete, bounded distributions. As shown in this section, it can also be applied to continuous states, but in such cases its precision heavily depends on the size of the bins of the histograms.

It is worth mentioning that, given unlimited time and memory resources, it could be possible to select an arbitrarily tiny bin size and use a histogram for almost any problem, computing the a posteriori probability of every of the huge number of possible cases. Given that this unlimited scenario is never the case, more advanced methods have been proposed to overcome the limitations of the histogram filter by representing and manipulating distributions performing non-uniform sampling. The most remarkable techniques proposed to generate and use approximated probability density functions for filtering purposes are described in chapter 8.

Figure 7.10: Beliefs for t_2 .Figure 7.11: Beliefs for t_3 .

Chapter 8

Stochastic sampling techniques

The previous chapter introduced the problem of environment modeling and its most common approaches. Two main classifications were described: a) distinguishing metric, topological and hybrid representations, and b); distinguishing deterministic and probabilistic representations. All approaches in the first classification can be appropriate depending on the objective of the robot. However, due to the noise and uncertainty of the real world, there are few cases in which deterministic approaches could be preferred over probabilistic ones (mostly, due to efficiency considerations). The Bayes filter was presented as a principled means to update probabilistic models. Unfortunately, it was highlighted that Bayesian filters can only be implemented analytically for very specific cases and that non-parametric methods are generally needed. Finally, the simplest non-parametric Bayesian filter, the histogram filter, was described along a discussion on its limitations.

The purpose of this chapter is two-folded: first, to describe how can probability density functions be represented using non-uniform sampling (section 8.1) and the most important family of algorithms designed for that purpose, the Monte Carlo methods (section 8.2). These will allow us to overcome some of the limitations of the histogram filter that were presented in the last chapter and achieve higher performance. The second goal is to provide a discussion on—probably—the most important Monte Carlo method in robotics (section 8.4) and some of its most important improvements (section 8.3). Thus, this chapter provides the necessary theoretical foundations needed to design and implement approximated non-parametric Bayesian filters—and to understand the most important contributions of this thesis.

8.1 Numerically representing probability distributions: the PDF - sample set duality

Instead of being devoted towards computing most-likely a posteriori estimations (as in equations 7.2 or 7.4) or recreating approximated PDF functions, most literature is devoted towards inferring the expectation of a function regarding a stochastic variable. The problem of inferring expectations of functions regarding stochastic variables can be mathematically expressed as:

$$\mathbb{E}[f(X)] = \int p(x)f(x)dx \quad (8.1)$$

where X is the stochastic variable and f the function of interest. However, despite these methods are generally not formulated with the aim of estimating probability density functions, their

generated sample sets can usually be exploited to produce approximations of PDFs.

Assuming that a set of independent and identically distributed random samples of a probability density function are drawn, several techniques can be used to approximately reconstruct the function using a histogram or mixture models the same way that a PDF can be used to obtain a sample set (hence the duality). Figure 8.1 shows an example of how can this be achieved using a simple histogram (figure 8.1.b) and a Gaussian *mixture model* (figure 8.1.c). Later in this chapter more advanced techniques than independent and identically distributed random sampling will be presented.

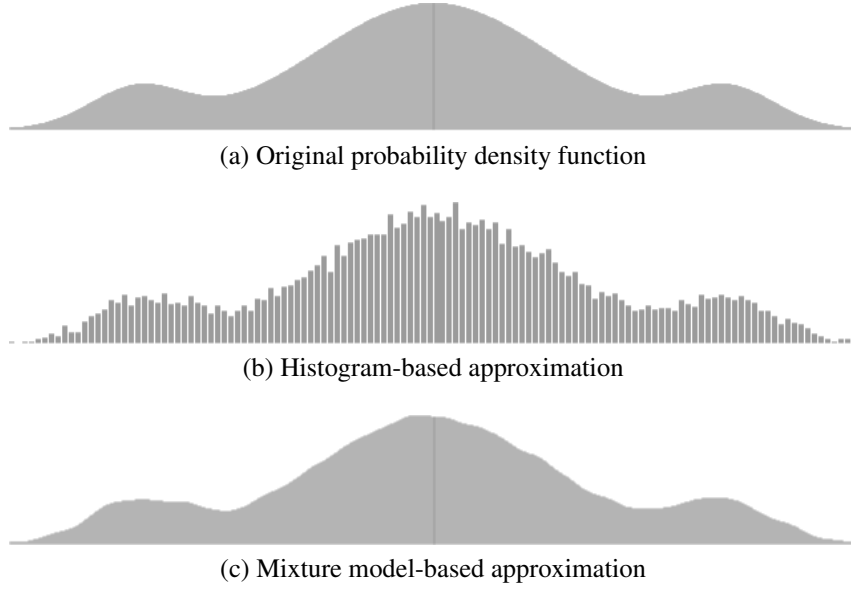


Figure 8.1: Figures (b) and (c) represent the PDF shown in (a). Figure (b), using a histogram. Figure (c) using a Gaussian mixture model. In order to generate these figures a total of 10000 samples were gathered using rejection sampling (section 8.2.1.1).

✓ **Mixture models** are methods to create probability density functions as the weighted sum of a finite number of simple distributions —kernels. The most common mixture model uses as kernel Gaussian PDFs and is known as Gaussian mixture model. However, the use Dirac delta functions is also common. A mixture model $\pi(x, \alpha_{1:N}, \omega_{1:N})$ composed of N kernels associated with samples $\alpha_{1:N}$ and weights $\omega_{1:N}$ can be mathematically defined as:

$$\pi(x, \alpha_{1:N}, \omega_{1:N}) = \sum_{i=1}^N w_i f(x, \alpha_i) \quad (8.2)$$

However, given that robots have sometimes to make crisp decisions based on these models, it is generally common to look for the most-likely point of the distribution instead of for a model of the distribution itself or the expectation of a function: expectations are usually of little help when robots have to deal with multi-modal distributions:

$$\hat{x}_t = \operatorname{argmax}_{x_t \in \alpha_{1:N}} p(x_t | z_t) \quad (8.3)$$

The remaining sections describe how to compute the probability density function of the belief and its maxima using sampling-based numerical approximations.

8.2 Monte Carlo methods

Monte Carlo methods are a family of algorithms used to sample, integrate or perform any kind of inference over arbitrarily complex probability density functions. The main characteristic of these methods is that, instead of sampling from a regular grid of points, Monte Carlo methods are based on randomly distributed samples. According to [Carlin and Louis, 2009], Monte Carlo methods can be classified as iterative and non-iterative, according to how the sample set is generated. Non-iterative methods generate N samples at once. On the other hand, iterative methods repeatedly include one sample at a time, taking into account the information provided by previous samples. The difference is therefore whether if the information of previous samples is used or not to produce the next sample. ✓

8.2.1 Non-iterative Monte Carlo methods

8.2.1.1 Rejection sampling

Rejection sampling is one of the most easy-to-understand Monte Carlo method. In order to sample from PDFs in closed form, it is necessary to analyze the inverse of their indefinite integral. Since it is not always known or can be difficult to calculate, sampling from arbitrarily complex PDFs requires alternative techniques. This is the purpose of *rejection sampling*. ✓
The basic idea is to sample from a distribution of our choice which we can readily sample and, depending on the values of such distribution and the distribution we want to sample on each sampled point, reject or accept the sample. The easy-to-sample distribution which is actually sampled as reference is usually called *proposal distribution*, while the distribution from which is intended to obtain samples as a result is known as the *target distribution*. ✓

The technique is better understood using the graphical example shown in figure 8.2, in which the goal is to sample points from the green area. Despite it is not trivial to sample the green area, it is easy to sample the red square (assuming we are able to draw samples from 1-dimensional uniform distribution, sampling from a square is just as easy taking two samples and using each of them as 2-dimensional coordinates). Because the green area is contained within the red one, drawing independent and identically distributed random samples from the red area and ignoring those that do not belong to the green area is equivalent to drawing independent and identically distributed random samples from the green area.

Similarly, since continuous PDFs provide the probability density for each point, this technique can also be used to sample functions. In order to achieve this, besides the target ($p(x)$) and proposal ($q(x)$) distributions, it is also needed a constant $k \in \mathbb{R}$ so that

$$\forall x \in X, p(x) \leq k q(x). \quad (8.4)$$

The pseudo-code to draw a sample from an arbitrary function, is shown in algorithm 3. It executes a loop until a satisfactory sample from the proposal distribution is obtained. In each iteration, algorithm 3 draws a sample x from the proposal distribution (see figure 8.3a). Then, it is randomly decided whether to reject or accept the sample depending on the value of a second sample y , taken from a uniform distribution (see figure 8.3b):

$$y \sim U(0, k q(x)). \quad (8.5)$$

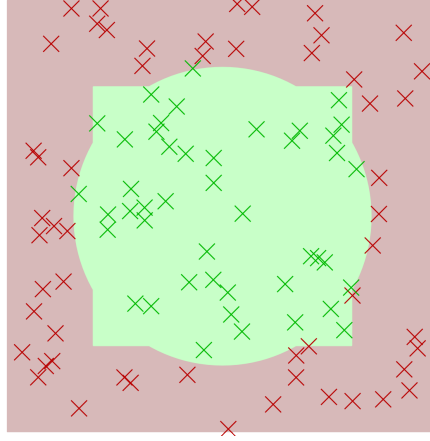


Figure 8.2: Graphical example of rejection sampling. Since the green area is contained within the red one, those samples drawn independent and identically distributed from the red area that lay within the green area are also independent and identically distributed from the green area.

Algorithm 3 Rejection sampling:

```

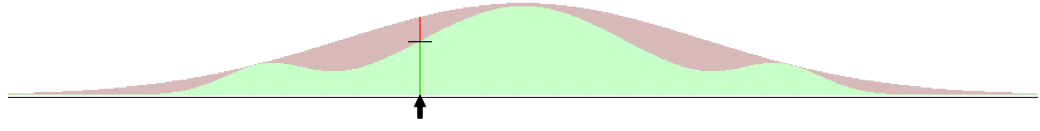
1: finished := False
2: x := 0
3: while not finished do
4:   draw  $x \sim q(x)$ 
5:   draw  $y \sim U(0, k q(x))$ 
6:   if  $y \leq k q(x)$  then
7:     finished := True
8:   end if
9: end while
10: return x

```

The sample x is accepted only if $y < p(x)$ (see figure 8.3a), otherwise the loop continues drawing potential samples. The mathematical derivation of rejection sampling algorithm can be found in [Smith and Gelfand, 1992]. It is worth mentioning that, given that $p(x)$ and $q(x)$ are both probability density functions (*i.e.*, they integrate to 1), the rejection and acceptance ratio are $(k - 1)/k$ and $1/k$ respectively, thus the algorithm becomes inefficient as the k value increases.

Despite rejection sampling is a technique whose sole purpose is sampling, not inference, it can be easily used for approximating estimations. Since points with higher probability are sampled more often, their contribution is also proportionally higher. Based on this fact, assuming a set of samples $x_{1:N}$ has been independent and identically drawn from $p(x)$, equation 8.1 can be approximated using such samples

$$\mathbb{E}[f(X)] = \int p(x) f(x) dx \simeq \sum_{i=1}^N f(x_i) \quad (8.6)$$



(a) In red, the area under the function proportional to the proposal distribution $k q(x)$. In green, the area under the target distribution. The first step of the rejection sampling algorithm consists on drawing a sample from the $k q(x)$ function (the sample is represented by the arrow).



(b) The second step consists on drawing a sample from a uniform distribution from 0 to $k q(x)$. If the sample (represented by the arrow) is less or equal to $p(x)$, as is the case in this figure, the sample is accepted, otherwise it is rejected.

Figure 8.3: Graphical example of the process of drawing a sample using rejection sampling.

8.2.1.2 Importance sampling

Importance sampling (IR) is a technique used to produce estimations of functions over stochastic variables. Using rejection sampling along equation 8.6 for estimating expectations can sometimes be inadvisable if there is no known PDF such that rejection sampling can be used with an acceptable low k value: high k values implies a high rejection rates, $(k - 1)/k$. If $p(x)$ has several *spiky* maxima, the k value has generally to be quite high. Here, the term *acceptable* depends on the application, specifically on the computational cost of computing the probabilities $p(x)$ and $q(x)$, which is needed for each iteration of the loop of rejection sampling. In robotics, this is especially the case of measurement probabilities when sensor readings are complex. For example, computing the probability of acquiring a particular image given a specific world model can be quite time-consuming.

As rejection sampling, importance sampling is also based on drawing samples from another easy-to-sample proposal distribution. However, instead of rejecting samples from the proposal distribution, importance sampling assign weights to the samples so that, those where the proposal distribution is higher than the target PDF, have lower weights (lower than 1) and vice versa. This way, despite the samples are generated using the proposal distribution, as the number of samples increases, the weighted sample set approximates to the target distribution asymptotically. Despite, theoretically, the only requirement of the proposal function is to be greater than 0 anywhere the target function is greater than 0 (because, theoretically, the number of samples should tend to infinity and this way it is guaranteed that the weights can balance the bias caused by the proposal function), in practice, the similarity between both functions plays a very important role in the efficiency of importance sampling. Thus, as opposed to rejection sampling, using importance sampling does not require finding an appropriate k value. Moreover, the main advantage of importance sampling over rejection sampling is that it exploits all samples.

Given a proposal distribution, $q(x)$, which can be easily sampled, equation 8.6 can be written as:

$$\mathbb{E}[f(X)] = \int p(x)f(x)dx = \int q(x)\frac{p(x)}{q(x)}f(x)dx \quad (8.7)$$

Assuming a third function $g(x) = \frac{p(x)}{q(x)}$ and a set of samples $s_{1:N}$ independent and identically

drawn from $q(x)$, using the same reasoning as in equation 8.6

$$\mathbb{E}[f(X)] = \int q(x) \frac{p(x)}{q(x)} f(x) dx = \int g(x) f(x) dx \simeq \sum_1^N g(s_i) \quad (8.8)$$

Substituting $g(x)$ back for its formula leads us to the equation of importance sampling

$$\mathbb{E}[f(X)] \simeq \sum_1^N \frac{p(s_i)}{q(s_i)} f(s_i) \quad (8.9)$$

where $s_{1:N}$ is a set of independent and identically samples drawn from $q(x)$. The ratio between the target and proposal probabilities $p(s_i)/q(s_i)$ in equation 8.9 is usually referred to as *sample weight*. Figure 8.4 provides a graphical example of the sample of one point.

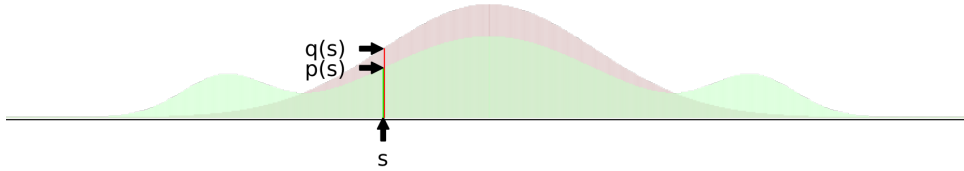


Figure 8.4: Graphical example of the sampling procedure used in importance sampling. All samples are used and, in order to cancel out the bias caused by the proposal distribution, the samples are weighted according to the $p(s_i)/q(s_i)$ ratio.

Additionally, it can be proved that, even if $p(x)$ can only be calculated up to an unknown scale factor importance sampling is still valid with minor modifications (see [Smith and Gelfand, 1992]):

$$\mathbb{E}[f(X)] \simeq \sum_1^N w_i f(s_i) \quad (8.10)$$

where, defining $p^*(x)$ as the scaled version of $p(x)$, w_i is defined as:

$$w_i = \frac{p^*(s_i)/q(s_i)}{\frac{1}{N} \sum_{j=1}^N p^*(s_j)/q(s_j)} \quad (8.11)$$

This is extremely important in Bayesian filters, where there is a normalizing factor which is very time-consuming—or impossible—to compute. This is the case, of the η variable of the correction stage of the Bayesian filter (see the Bayesian filter algorithm 1 and equation 7.15). Due to this importance, most of the remaining Monte Carlo algorithms covered in this work will also have this property.

This normalizes the weights so that their sum is always 1. Assuming this more general case, algorithm 4 provides the pseudo-code corresponding to importance sampling. Lines 1 to 7 compute the sampling set s and the normalizing factor W , while lines 8 to 12 compute equation 8.11. The validity of this approximation is demonstrated in [Smith and Gelfand, 1992].

Algorithm 4 Importance sampling:

```

1:  $s := \emptyset$ 
2:  $W := 0$ 
3: for  $i \in 1 \dots N$  do
4:   draw  $x \sim q(x)$ 
5:    $S := S \cup x$ 
6:    $W := W + (p(x)/q(x))$ 
7: end for
8:  $sum := 0$ 
9: forall  $s_i \in s$  do
10:   $sum := sum + f(s_i) \frac{p(s_i)/q(s_i)}{W}$ 
11: end forall
12: return  $sum/N$ 

```

8.2.1.3 Sampling/Importance Resampling

The *sampling/importance resampling* algorithm (*S/IR*) [Rubin et al., 1988], also known as *bootstrap resampling* [Efron, 1982] has the purpose of drawing samples according to complex stochastic variables. As importance and rejection sampling, sampling/importance resampling is also based on a proposal distribution. The novelty of the technique is that it uses importance sampling weights to produce a second sample set which approximates to a sample set drawn from $p(x)$. ✓

First, N samples are drawn using the proposal distribution (the technique used to draw these samples is irrelevant) (see the upper-right part of figure 8.5). Then, they are assigned weights according to importance sampling (see the sample colors in the bottom-right part of figure 8.5). Finally, a new N -sample set is generated from the previous set of samples, so that the probability of drawing a particular sample from the previous set is proportional to its weight (bottom-left part of figure 8.5). ✓

The pseudo-code for the sampling/importance resampling is shown in algorithm 5. Lines 1..6 are similar to those of algorithm 4. Lines 7..10 generate T , a temporal, weighted set of tuples with the initial samples and their corresponding weight. Finally, lines 11..16 create and return the new set of N samples R , where each element t_i is drawn from T with a probability proportional to its weight.

Intuitively, the algorithm creates a discrete distribution represented by the tuples in T , which according to importance sampling, resemble the target distribution. Thus, sampling from this tuples with a probability proportional to their weights is similar to sampling from the target distribution. Proof in [Smith and Gelfand, 1992].

In importance sampling the sample set is distributed according to the proposal distribution, it is the weights that correct the target-proposal distribution bias. Comparing to regular importance sampling, the main advantage of sampling/importance resampling is that (once S/IR is finished) the generated sample set is distributed according to the target distribution, like when using rejection sampling. No weights are needed to correct the target-proposal distribution bias anymore. Comparing to rejection sampling, the advantages of S/IR are that it is no necessary

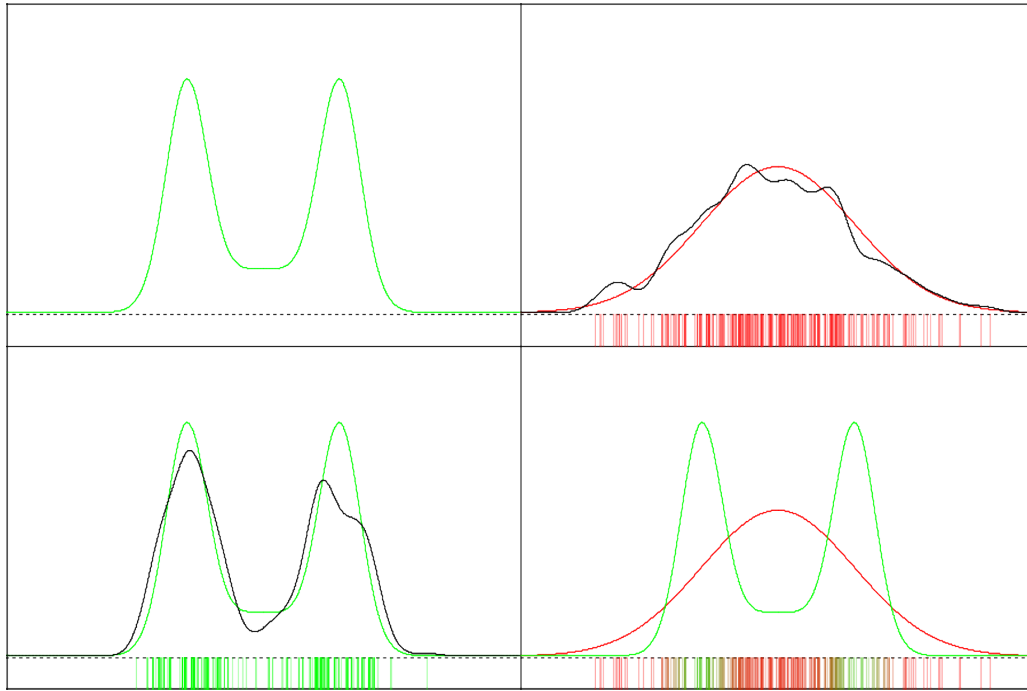


Figure 8.5: **Top-left:** target distribution (green). **Top-right:** proposal distribution along the initial sample set (red) along a the function reconstructed using a Gaussian mixture model (black). **Bottom-right:** both distributions and the samples colored according to their weight (green represent high weights, red lower ones). **Bottom-left:** target distribution reconstructed using a Gaussian mixture model (black) after resampling.

to know the k used in rejection sampling and that since all samples are used, no computational time is wasted in generating samples which are going to be rejected and thus not used. In robotics the PDFs have usually unpredictable maxima (both in number and magnitude) so a high k constant for rejection sampling would be needed making the process highly inefficient. Moreover, unlike regular importance sampling, S/IR can be used not only for estimating the expectation of functions but also for approximating the distribution itself.

8.2.1.4 Particle filtering: Sequential Importance Resampling

✓ **Particle filtering** (also known as “sequential importance resampling Monte Carlo” or “bootstrap filter” depending on the authors), is nowadays the most important probabilistic technique in robotics. Its purpose is creating approximated Bayesian filters even when the related transition and measurement probability functions are complicated—even with an unknown number of modes— without the need of explicitly computing the a priori necessary integrations [Gordon et al., 1993, Newton, M. A. and Raftery, A. E., 1994].

Particle filtering is based on S/IR. Once sampling/importance resampling is understood, it is fairly easy to understand how particle filters work. The key idea is the following: given the *sample set - PDF duality* and that S/IR allows us to sample from complex PDFs, we can use S/IR to obtain a set of samples resembling the posterior distribution $bel(x_t)$ using $bel(x_{t-1})$ as starting point. For best understanding, we are first going to split the process in the two stages: first, sampling $\bar{bel}(x_t)$ from $bel(x_{t-1})$, and then sampling $bel(x_t)$ from $\bar{bel}(x_t)$, as we did in

Algorithm 5 Sampling/Importance resampling:

```

1:  $S := \emptyset$ 
2: for  $1 \dots N$  do
3:   draw  $x \sim q(x)$ 
4:    $S := S \cup x$ 
5: end for
6:  $T := \emptyset$ 
7: forall  $s_i \in S$  do
8:    $w_i := p(s_i)$ 
9:    $T := T \cup \langle w_i, s_i \rangle$ 
10: end forall
11:  $R := \emptyset$ 
12: for  $1 \dots N$  do
13:    $i \stackrel{\text{iid}}{\sim} T \propto \{w_0, \dots, w_N\}$ 
14:    $R := R \cup s_i$ 
15: end for
16: return  $R$ 

```

section 7.5.

Assuming that $p(x_t|u_t, x_{t-1})$ can be easily computed, we can sample $\overline{bel}(x_t)$ from $bel(x_{t-1})$ using **ancestral sampling**. Basically, the idea is that when it is desired to sample from a $\sqrt{}$ distribution $p(A) p(B|A)$ the process can be done in two steps: **a**) obtaining a value for the A variable using the unconditioned PDF ($p(A)$) and then; **b**) obtaining a value for the variable B using $p(B|A)$ given the parameters of A just sampled. In other words, first a sample a , is drawn from $p(A)$ (the parent node if the context is represented in a Bayesian network), then, such sample is used to sample $p(B|a)$ —instead of $p(B|A)$ — and compute (if necessary) $p(A) p(B|A)$.

Using the same reasoning and, assuming that u_t is known and the previous posterior distribution is approximated by the sample set $S_{bel,t-1}$, the PDF $\overline{bel}_{t-1}(x)$ can be approximated from $S_{bel,t-1}$ as:

$$\overline{bel}_t(x) = \int p(x|u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1} \simeq \sum_i^N p(x_t|u_t, S_{bel,t-1}(i)), \quad (8.12)$$

where x_t is sampled on-line from $p(x_t|u_t, S_{bel,t-1})$ relying on the known u_t and the given $S_{bel,t-1}(i)$ values.

Performing the summation, would provide us with an approximation to $\overline{bel}_t(x)$. However, since the purpose is not to compute $\overline{bel}_t(x)$ for a specific x but to compute a sample set approximately drawn from it, instead of computing the summation, we will only store the samples.

Now, we can use S/IR to achieve a sample set approximately drawn from $bel(x_t)$ provided that we already have a sample set approximately drawn from $\overline{bel}_t(x)$. The weights used in S/IR are calculated like in the regular IS as the ratio $p(x)/q(x)$, where $p(x)$ is the target function

($bel(x_t)$ in our case) and $q(x)$ is the proposal function ($\overline{bel}(x_t)$):

$$w_i = \frac{bel(x_t)}{\overline{bel}(x_t)} = \frac{p(z_t|x_t) \overline{bel}(x_t)}{\overline{bel}(x_t)} = p(z_t|x_t) \quad (8.13)$$

After performing S/IR this way, the remaining sample set is approximately drawn from $bel(x_t)$

The pseudo-code for the sampling/importance resampling is shown in algorithm 6. Lines 1..5 implement the sampling of the predicted distribution $S_{\overline{bel}.t}$. Lines 6..10 are similar to those of algorithm 5 using the weighting calculated in equation 8.13. They generate the sample set for the posterior T and return, a temporal, weighted set of tuples with the initial samples and their corresponding weight. Finally, lines 11..16 create and return the new set of samples that approximate $S_{bel.t}$, just as in algorithm 5.

Algorithm 6 Particle filtering (long version):

Require: $S_{bel.t-1}$: The sample set corresponding to the *corrected* belief at time $t - 1$.

```

1:  $S_{\overline{bel}.t} := \emptyset$ 
2: for  $i = 1 \dots N$  do
3:   draw  $x_t \sim p(x_t|u_t, S_{bel.t-1}\langle i \rangle)$ 
4:    $S_{\overline{bel}.t} := S_{\overline{bel}.t} \cup x$ 
5: end for
6:  $T := \emptyset$ 
7: forall  $s_i \in S_{\overline{bel}.t}$  do
8:    $w_i := p(z_t|x_t)$ 
9:    $T := T \cup \langle w_i, S_{bel.t-1}\langle i \rangle \rangle$ 
10: end forall
11:  $S_{bel.t} := \emptyset$ 
12: for  $1 \dots N$  do
13:    $i \stackrel{\text{iid}}{\sim} T \propto \{w_0, \dots, w_N\}$ 
14:    $S_{bel.t} := S_{bel.t} \cup S_{bel.t-1}\langle i \rangle$ 
15: end for
16: return  $S_{bel.t}$ 

```

Algorithm 6 provides the algorithm for particle filters, in a three-staged version for better understanding. It can be shortened if we avoid to create a set for the predicted belief ($S_{\overline{bel}.t}$), as shown in algorithm 7.

The experiment in section 15 provides a real example of how to use particle filtering to estimate the yaw of a robot in an indoor environment using a RGBD camera.

8.2.2 Iterative Monte Carlo methods

As the previously described methods, iterative Monte Carlo methods can be used to sample, integrate or perform inference over arbitrarily complex probability density functions. Unlike

Algorithm 7 Particle filtering:**Require:** $S_{bel,t-1}$: The sample set corresponding to the *corrected* belief at time $t - 1$.

```

1:  $S_{bel,t} := T := \emptyset$ 
2: for  $i = 1 \dots N$  do
3:   draw  $x_t \sim p(x_t | u_t, S_{bel,t-1}\langle i \rangle)$ 
4:    $w_i := p(z_t | x_t)$ 
5:    $T := T \cup \langle w_i, S_{bel,t-1}\langle i \rangle \rangle$ 
6: end for
7: for  $1 \dots N$  do
8:    $i \stackrel{\text{iid}}{\sim} T \propto \{w_0, \dots, w_N\}$ 
9:    $S_{bel,t} := S_{bel,t} \cup S_{bel,t-1}\langle i \rangle$ 
10: end for
11: return  $S_{bel,t}$ 

```

non-iterative methods, in iterative Monte Carlo each sample drawn is recursively conditioned on the previous one. Because this process resembles a Markov Chain, iterative Monte Carlo is also called **Markov Chain Monte Carlo** (**MCMC** for short) —see figure 8.7. The main advantage of iterative Monte Carlo methods is that they generally scale better than non-iterative methods such as importance resampling when the number of dimensions is relatively high. ✓

8.2.2.1 Metropolis sampling

As the previous methods, the **Metropolis** algorithm [Metropolis et al., 1953] relies on an easy-to-sample proposal distribution $q(x_t | x_{t-1})$ which is used as transition function for the Markov chain. If enough samples are drawn using the Markov chain generated using the transition function, these samples will approximately follow the target distribution. In order to ensure that the algorithm works as expected two conditions must be met: ✓

- The transition function must be a symmetric function of its arguments (*i.e.*, $\forall x_a, x_b : q(x_a | x_b) = q(x_b | x_a)$).
- The target distribution can be evaluated up to a constant scale factor.

The samples are generated using algorithm 8 (see [Bishop, 2007] for the mathematical derivation). Given an initial sample, the algorithm iteratively includes a sample: a new sample x^* , drawn from the transition function $q(x^* | x_{i-1})$ if a temporary sample drawn from an uniform distribution $U(0, 1)$ is smaller than the $p(x^*)/p(x_{i-1})$ ratio, or the last included sample otherwise:

Despite that Metropolis, and Markov Chain Monte Carlo algorithms in general, are usually better fitted for sampling high dimensional spaces, there are two limitations that must be taken into account if the generated samples are going to be used as samples drawn from the target distribution:

1. Depending on the dimensionality of the function to sample, since x_0 is arbitrary, if the initial sample is selected from a low-density area, the initial samples might not be repre-

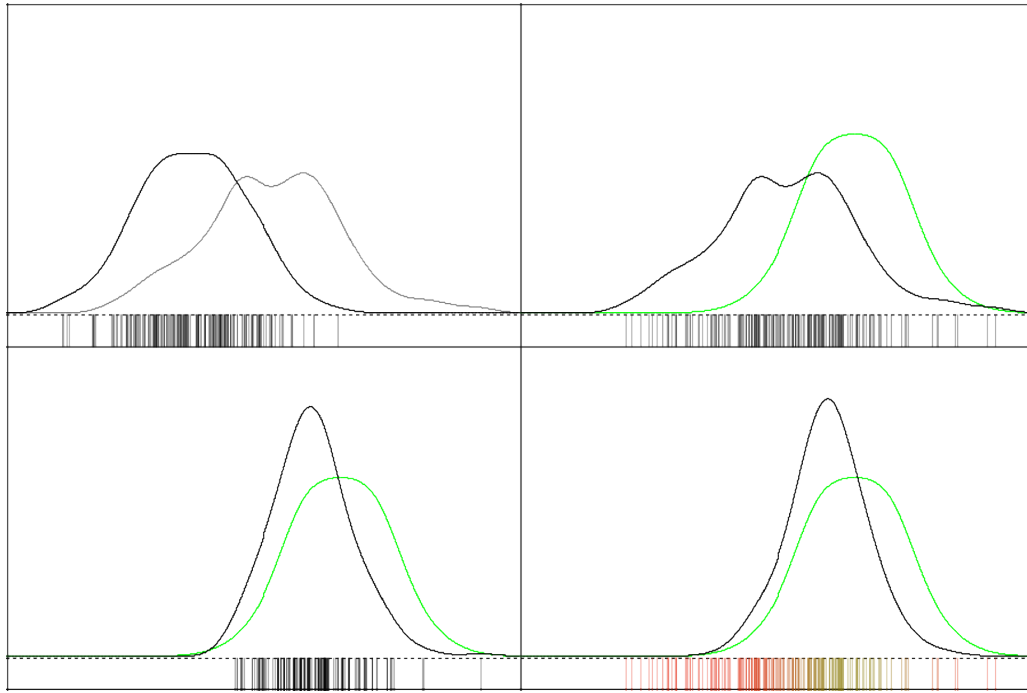


Figure 8.6: An example of a particle filter update. **Top-left:** In black, the initial set of particles ($bel(x_{t-1})$) and its reconstruction. **Top-right:** In black, $\overline{bel}(x)$, the sampled set of particles (according to $p(x_t|x_{t-1}, u_t)$) and its reconstruction (also shown in the top-left subfigure in gray). See line 3 of algorithm 7. **Bottom-right:** In black, the reconstruction of the weighting of the predicted set of particles according to the $p(z_t|x_t)$ function (in green). Lines 4 and 5 of algorithm 7. **Bottom-left:** The set of particles after the resampling stage (representing $bel(x_t)$). Lines 8 and 9 of algorithm 7.

sentative of the target distribution. This, often called *burn-in period*, makes necessary to discard some of the initial samples, depending on the dimensionality of the function to sample.

2. Because each sample depends on the previous one, consecutive samples will be correlated to some extent, depending on the variance of the transition function. In practice, in order to obtain, uncorrelated samples, only the n -th samples with n multiple of M and $M \ll N$ are actually considered. The higher variance, the lower M can be. However, the rejection rate will also increase.

As opposed to rejection sampling, it must be noted that the Metropolis algorithm does not waste the time spent in generated samples, since a new sample is always generated in each iteration (a new one or the previous one).

8.2.2.2 Metropolis-Hastings sampling

- ✓ The **Metropolis-Hastings** algorithm is a generalization of the Metropolis algorithm [Hastings, 1970], where the transition function is not necessarily symmetric. This way, the Metropolis-Hastings algorithm can perform more clever sampling than the regular Metropolis algorithm (a naive random walk) if a transition function which produces a lower rejection

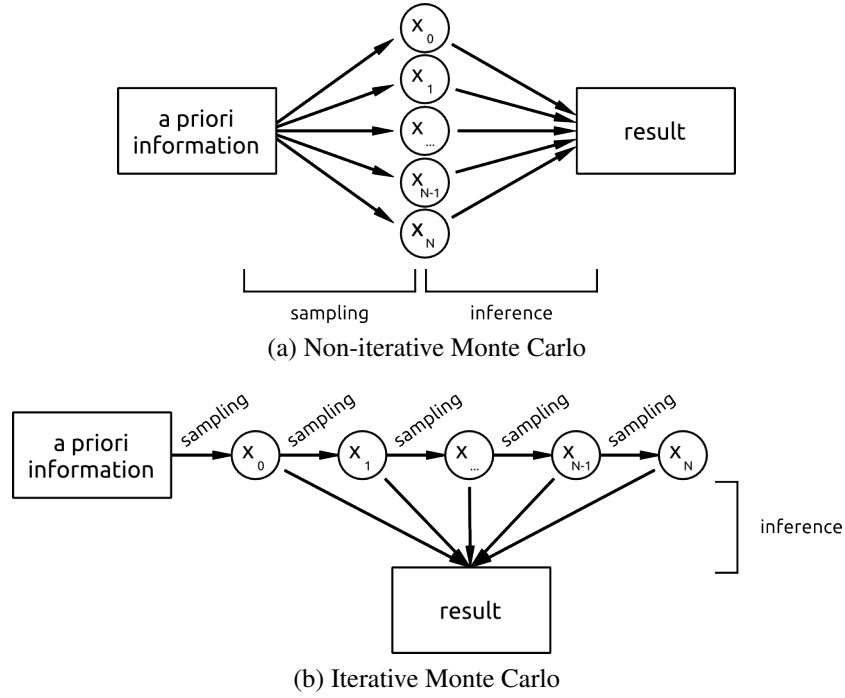


Figure 8.7: General description of how the different Monte Carlo methods work. Non-iterative methods (subfigure a) generate independent samples given the known data. On the other hand, iterative methods generate a Markov Chain of samples, where the first sample is drawn from an a priori distribution but the rest is based on the previous one.

rate is known, even if its is not symmetric.

The samples are generated using algorithm 9, a slight modification of algorithm 8 (see [Bishop, 2007] for the mathematical derivation). As the regular Metropolis algorithm, given an initial sample, the Metropolis-Hastings algorithm iteratively includes a sample x^* , drawn from the transition function $q(x^*|x_{i-1})$ if a temporary sample drawn from an uniform distribution $U(0, 1)$ is smaller than the a ratio of equation 8.14. Otherwise the last sample remains for another iteration.

$$a = \frac{p(x^*) p(x_{i-1}|x^*)}{p(x_{i-1}) p(x^*|x_{i-1})} \quad (8.14)$$

8.2.2.3 Gibbs samplers

Gibbs sampling is a special case of Metropolis-Hastings algorithm designed for very high dimensional spaces. The key difference is that given the current sample, the next proposed sample is not obtained in a single step; instead, each individual space of the samples is updated one by one, generating the proposal after iterating over all the subvariables (see algorithm 10).

Despite algorithm 10 only yields a valid sample once every M one-dimensional samples, it might be considerably more efficient than sampling from the whole joint distribution

$$q(x_1, x_2, \dots, x_{M-1}, x_M) \quad (8.15)$$

because the internal samples

$$q(x_j^*|x_1^*, \dots, x_{M-1}^*, x_M^*) \quad (8.16)$$

Algorithm 8 Metropolis algorithm:

Require: x_0 : The initial sample.

```

1:  $S := \emptyset$ 
2: for  $i = 1 \dots N$  do
3:   draw  $x^* \sim q(x^*|x_{i-1})$ 
4:    $a = p(x^*)/p(x_{i-1})$ 
5:   draw  $r \sim U(0, 1)$ 
6:   if  $a \geq r$  then
7:      $x_i = x^*$ 
8:   else
9:      $x_i = x_{i-1}$ 
10:  end if
11:   $S = S \cup x_i$ 
12: end for
13: return  $S$ 

```

Algorithm 9 Metropolis-Hastings algorithm:

Require: x_0 : The initial sample.

```

1:  $S := \emptyset$ 
2: for  $i = 1 \dots N$  do
3:   draw  $x^* \sim q(x^*|x_{i-1})$ 
4:    $a = \frac{p(x^*) p(x_{i-1}|x^*)}{p(x_{i-1}) p(x^*|x_{i-1})}$ 
5:   draw  $r \sim U(0, 1)$ 
6:   if  $a \geq r$  then
7:      $x_i = x^*$ 
8:   else
9:      $x_i = x_{i-1}$ 
10:  end if
11:   $S = S \cup x_i$ 
12: end for
13: return  $S$ 

```

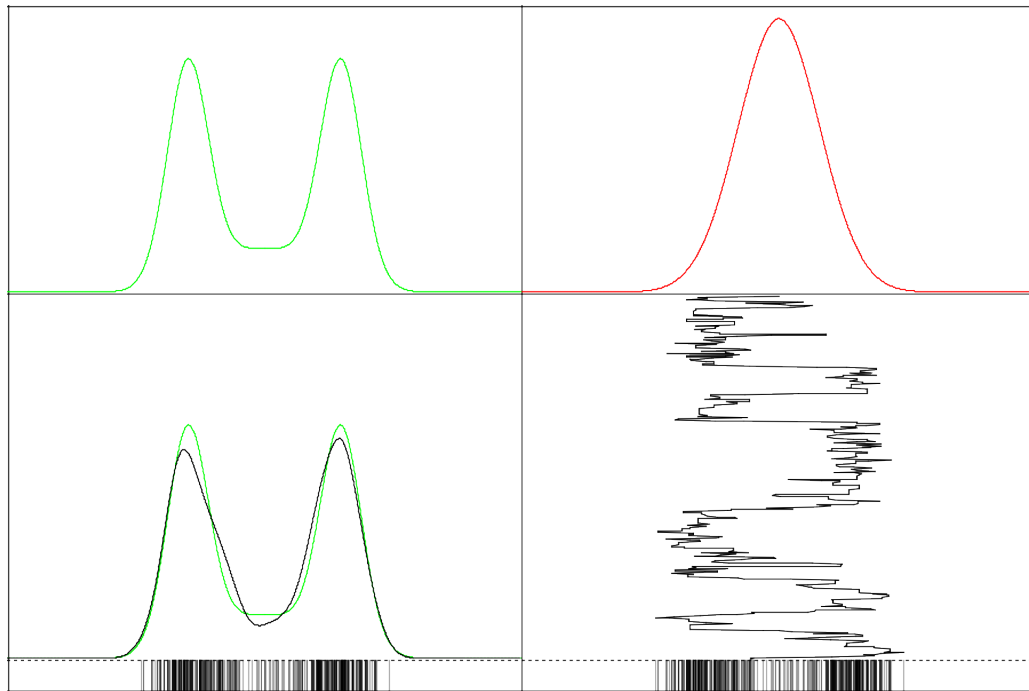


Figure 8.8: In the top-left and top-right, the target and transition functions, respectively. In the right-hand side of the bottom, a representation of the Markov Chain sampled. It must be noted that, despite some transitions between the two modes exist, most samples are drawn from the same node as the preceeding sample (this phenomena can be reduced by incrementing the variance of the transition function at the cost of a higher rate of rejected new samples). Finally, the probability density function reconstructed using the samples drawn using the Metropolis algorithm.

are almost always drawn much faster.

8.2.3 Simulated Annealing in MCMC

As it was previously pointed out, most of the robotics and computer vision applications use particle filters to obtain a maximum a posteriori estimate (generally the most likely model or the most likely position of an object being tracked). In the context of Markov Chain Monte Carlo techniques these estimates will be located in one of the samples of the chain —if the estimate is obtained by selecting the most-likely sample— or anywhere else if the search is based on the reconstruction of the PDF using a kernel-based technique. Assuming a considerable amount of samples have been taken and, due to efficiency considerations, only the first option is used in practice. Thus, since the PDF is not going to be reconstructed, there is little interest in sampling unlikely areas (besides for a very time-consuming robustness). Based on this fact and inspired in the concept of annealing in materials science ¹ Kirkpatrick et al. developed the concept of *simulated annealing* [Kirkpatrick et al., 1983, Neal, 2001].

The method consists on initializing a Metropolis-Hastings simulation with a high variance

¹In metallurgy “annealing” refers to a technique used to enhance the properties of materials by increasing the temperature (entropy) and slowly decreasing it. This way, their molecules are able to find more stable configurations, of less energy than the initial one. Simulated annealing aims to find more likely or efficient configurations.

✓

Algorithm 10 Gibbs' mechanism for sample generation:

Require: x_i : The current sample, composed of M subvariables.

```

1:  $x^* = x_i$ 
2: for  $j = 1 \dots M$  do
3:    $A = x_1^*, x_2^* \dots x_{j-1}^*$ 
4:    $B = x_{j+1}^*, x_{j+2}^* \dots x_M^*$ 
5:   draw  $x_j^* \sim q(x_j^* | A, B)$ 
6: end for
7: return  $x^*$ 

```

(resembling a high temperature) and then lowering it until it slowly *freezes*. When the variance is low and no substantial changes occur, [Kirkpatrick et al., 1983] suggests to select the last sample as the optimization result. However, it is also possible to store the best sample and use such sample as the result. Despite it will generally lay close to the last sample of the chain, when comparing the candidate configurations is relatively fast, this policy is generally desirable. The variance of the proposal function is generally updated (diminished) according to an exponential function (multiplying the previous variance by a number relatively smaller than 1), despite any monotonically decreasing function can actually be used.

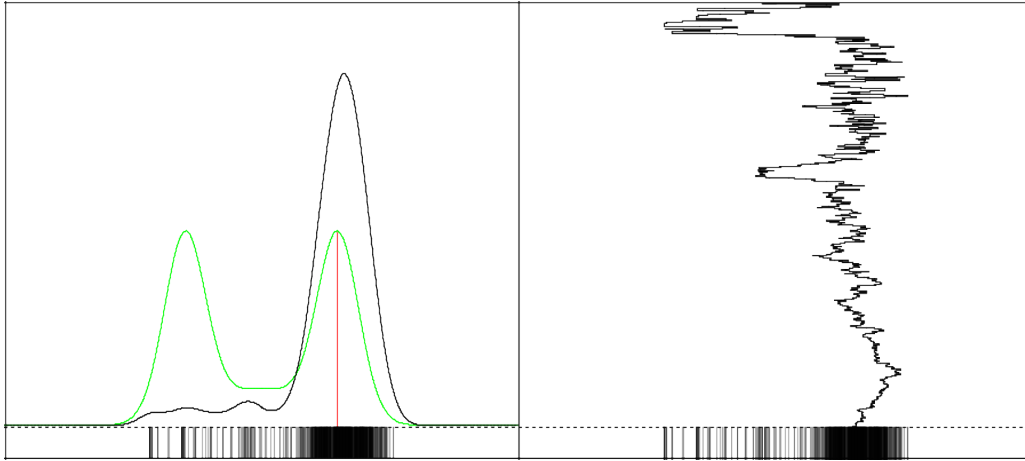


Figure 8.9: In the right hand side of this figure it is shown the result of applying simulated annealing to the Metropolis-Hastings algorithm in order to look for the maximum of the function shown in green in the left hand side. Also in the left hand side, it is shown the higher sample as a red vertical line.

The applications of simulated annealing are numerous. For example, it is used in [Sanchez et al., 2012] to search for the optimal kinematic calibration of a robotic arm whose structure is known. In fact, we used annealed particle filtering in this thesis to estimate the distance from the robot to a wall using a RGBD camera (see the experiment in section 16).

8.2.3.1 Bayesian filters using MCMC

Iterative Monte Carlo methods have also been used to implement Bayesian filters by replacing the S/IR sampling step of regular particle filters with MCMC alike sampling techniques, generally Metropolis-Hastings. The most common method to perform this kind of particle filtering is to substitute line 3 of algorithm 7 with a Metropolis-Hastings policy for particle transitioning. See [Pitt and Shephard, 1999] and [Septier et al., 2009] for a review of these techniques and other slightly different variations. In particular, [Pitt and Shephard, 1999] proposes using auxiliary particle filtering (see section) using either MCMC or S/IR sampling.

MCMC-based particle filtering has mostly be used for multiple target tracking, see [Khan et al., 2004] or [Septier et al., 2009].

8.2.3.2 Reversible jump MCMC

Besides several algorithms that boost the convergence of MCMC algorithms taking into account information about the derivative of the target function—which is usually not known in robotics or computer vision—the most remarkable method left to describe is **Reversible-jump MCMC** (**RJMCMC**) [Green, 1995]. RJMCMC is a generalization of Metropolis-Hastings where the sampling process can take place in different dimensions. As opposed to regular Metropolis-Hastings (where the samples drawn share always the same state space with randomly updated values), in Reversible-Jump MCMC the proposal function can also return states of different sizes. RJMCMC has successfully been used for different applications such as image parsing [Han and Zhu, 2009] or multi-object tracking [Smith, 2007]. Despite the definition of RJMCMC is quite simple and its usage can be extremely efficient, the simulation must be properly designed in order to work. ✓

First, it must be ensured that any jump can be reverted (usually by making sure the chain is *ergodic i.e.*, that any state can be reached in a finite time regardless of the initial state). Thus, the proposal function should not only increase the state space but also decrease it with non-zero probability.

Second, in order to prevent increasing dramatically the rejection ratio, it is generally advised to avoid performing jumps that modify the state space and modify the previous values at the same time. This behavior usually leads to samples with very low posterior probability. Because of this fact and in order to simplify the proposal function, instead of implementing a quite complex proposal function, it just stochastically selects a function from a set of simpler ones that only modify the values or somehow modify the state space. Despite it is common to select the function by assigning a constant probability to each of them, this process can be more sophisticated.

Finally, one the most hard-to-solve issues regarding the use of reversible-jump MCMC is how samples are evaluated and updated (both their values and their structure). Special care must be taken when increasing the dimension of the samples since, depending on the measure probability function ($p(z_t|x_t)$), the sampling process might end up in degenerated cases. For example, consider a multi-object tracking system: if the measure probability function is designed to provide a high score to the models (x_t) explaining the input data (z_t), it would provide a perfect fit to a model with containing as many objects as pixels in the images. Therefore, the corresponding proposal function must penalize unlikely complex models.

8.3 Improving Particle Filtering

Particle filtering is the most widely used technique used in robotics in order to maintain beliefs in time over the state variables that robots use to store the information they need. Moreover, they require a low number of very reasonable conditions in order to be used. This section provides an overview of the most common and promising improvements over regular particle filtering.

8.3.1 Rao-Blackwellized particle filters

One of the most important drawbacks of particle filters is that the number of particles—and time—needed to update a belief with a certain precision grows exponentially with the number of dimensions (*i.e.*, the so called *curse of dimensionality*). Sometimes a state space x can be divided into two subspaces x' and r (such as in figure 8.10) so that: a) x'_t can be estimated using particle filtering by marginalizing r and b) r_t given x'_t can optimally be estimated using a parametric filter. In such cases **Rao-Blackwellized** particle filters [Gelfand and Smith, 1990]

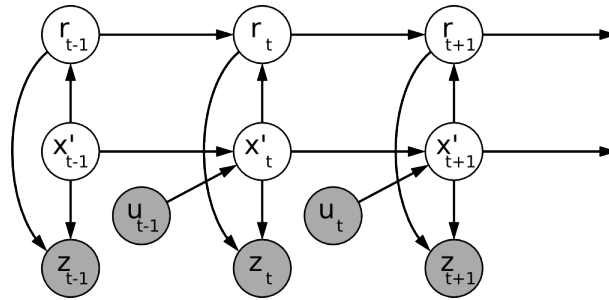


Figure 8.10: A dynamic bayesian network whose the state, composed of two variables x and r , can be more efficiently filtered using a Rao-Blackwellized particle filtering (instead of a regular one).

(RBPF) can be used to estimate $bel(x_t)$. They are a type of particle filters where only part of the variables of the particles (x'_t) are updated using regular particle filtering. The rest of the variables (u_t) are marginalized and updated using a parametric Bayesian filter (generally the Kalman filter) using the information of x'_t already available in the particles.

The reduction of the size of the state space makes possible to reduce the number of particles whilst maintaining precision. Despite that a parametric Bayesian filter must be run for every particle, their complexity is generally constant, so RBPFs still represent an important performance boost. Rao-Blackwellized particle filters are very known for its usage for SLAM, see [Grisetti et al., 2006]. For a deeper introduction to Rao-Blackwellized particle filters, see [Doucet et al., 2000].

8.3.2 Annealed particle filters

✓ **Annealed Particle Filtering** (*APF*) was proposed in [Deutscher and Reid, 2005] in the context of 3-dimensional human body tracking (despite there were previous similar approaches). As it was pointed when covering simulated annealing MCMC (section 8.2.3), most of the robotics and computer vision applications use particle filters to obtain a maximum a

posteriori estimate (the most likely model or the most likely pose of an object which is being tracked).

The cooling schedules of simulated annealing MCMC can also be used along particle filtering to look for the maximum a posteriori estimate. Annealed particle filters perform a M fixed number of updates to the particles per timestep, where each update is called *layer* and introduces less noise to the particles. The particle set remaining after the execution of the M layers is considered the output of the update of the annealed particle filter.

8.3.3 Auxiliary particle filters

Auxiliary particle filtering (APF) was proposed in [Pitt and Shephard, 1999] and is another improvement proposal of the regular particle filtering algorithm (see [Vlassis et al., 2002] and [Johansen and Doucet, 2008] for deeper understanding and applications). According to the literature, by redefining the sampling process, auxiliary particle filters behave better when the actual parameters are located in the tail of the predicted belief or when the measure PDF is peaked. See figure 8.11, where most of the particles for $bel(x)$ correspond to the same weighted samples (particles are drawn semi-transparent, so opaque lines correspond to many particles in located in the same point).

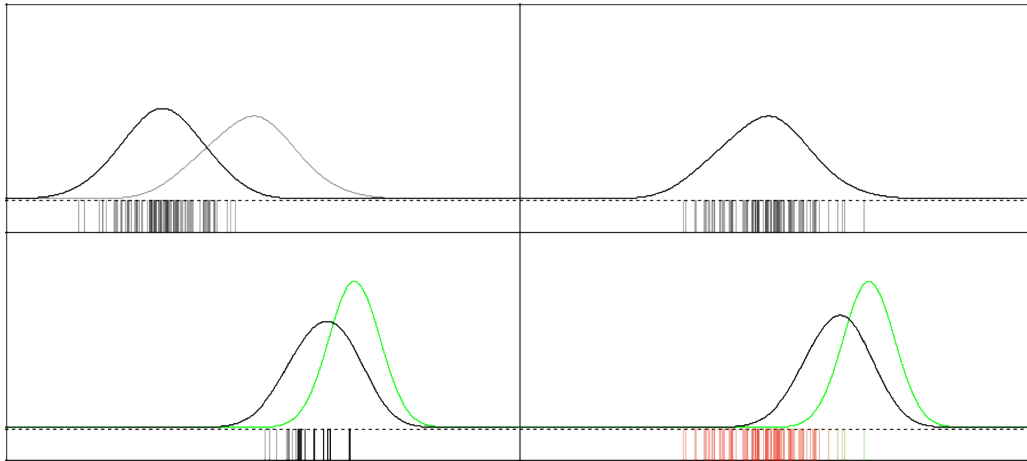


Figure 8.11: Update step of a regular PF where the actual parameters are in the tail of the predicted belief and the measure PDF is peaked. Most of the samples (particles) are wasted, since most of them are located in the same position.

Despite APF can be seen from different points of view [Johansen and Doucet, 2008], the novelty of the method is often interpreted as an auxiliary filtering process run when sampling from the $\overline{bel}(x)$ that enforces the sampling of the most likely particles of the prior. In order to do so, instead of using the same probability for all the particles as in line 3 of algorithm 7, particles representing $\overline{bel}(x)$ are drawn $\sim p(x_t|u_t, S_{bel,t-1}(\hat{i}))p(z_t|S_{bel,t-1}(\hat{i}))$.

8.3.4 Partitioned Sampling

Partitioned sampling [MacCormick and Blake, 2000] is another proposal to reduce the impact of the curse of dimensionality when working with high dimensional states. As described in [Bandouch, 2010] it can be seen as the ordered execution of a sequence of particle filters

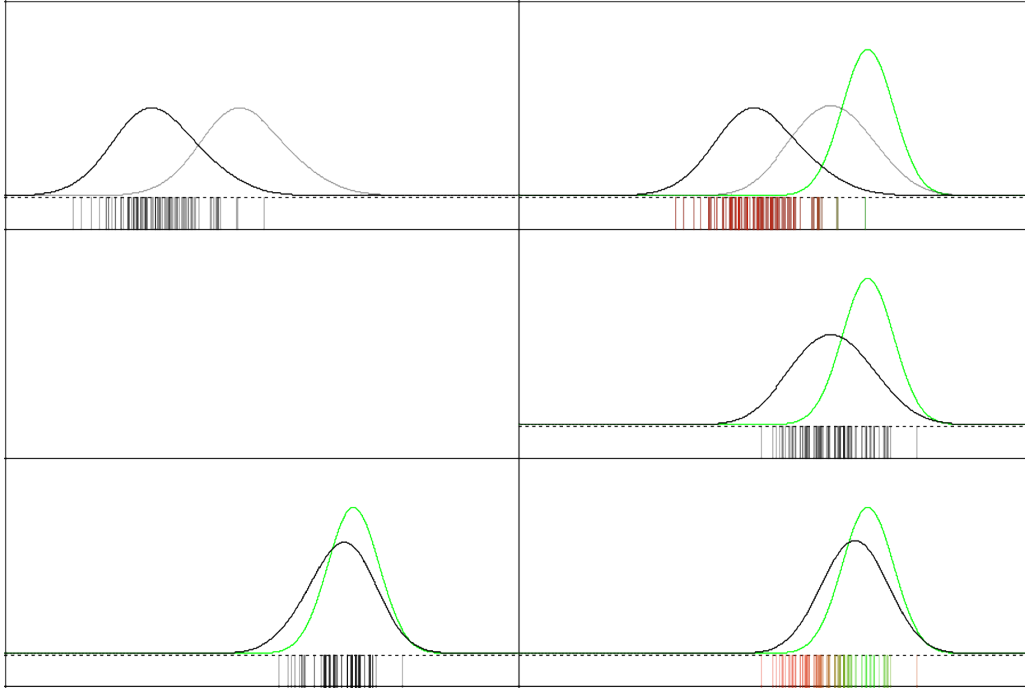


Figure 8.12: An example of an auxiliary particle filter update. **Top-left:** In black, the initial set of particles and the reconstruction of $bel(x_{t-1})$. **Top-right:** In black, the particles are deterministically updated according to u_t and weighted (in grey, the weighted representation of the particles). In green, the likelihood function ($p(z_t|x_t)$). **Middle row:** In black, the particles are sampled from $p(x_t|x_{t-1}, u_t)$, each particle being selected for update with a probability proportional to its weight. **Bottom:** The previous set of particles is used as if they represented $bel(x)$ in the regular PF.

estimating different subspaces of the state variable x : $x^1, x^2 \dots x^M$. In order to ensure that partitioned sampling is a valid belief estimation for x_t two conditions must be met:

- The transition dynamics of each subspace x^i must be statistically independent of those of the subspaces following them in the sequence of particle filters. As pointed out in [Bandouch, 2010] this is case *e.g.*, when the variable to estimate corresponds to a kinematic (tree-like) structure. Let $g^i(x_{t-1})$ be a function predicting the dynamics of the subspace number i :

$$p(x_t|x_{t-1}) = g^M(\dots(g^2(g^1(x_{t-1})))) . \quad (8.17)$$

- For every subspace i it must be possible to evaluate a function $p(y_t|x_t^i)$ that must be peaked in the same regions as $p(x_t^i|y_t)$.

8.3.5 Subspace hierarchical particle filters

✓ **Subspace hierarchical particle filtering** (proposed in [Brandao et al., 2006]) is another extension to regular particle filtering aimed to reduce the impact of the curse of dimensionality. Subspace hierarchical particle filtering (SHPF) focuses on improving the efficiency (*i.e.*, reducing the number of particles needed) by replacing a single particle filter taking into account the

whole state space of the problem with a network of smaller particle filters —subspace particle filters. Despite each subspace particle filter is specialized in optimizing a particular subset of the state space and can manage a varying number of particles all of them share the same likelihood function. The resulting network is an acyclic graph where each node represents a subspace filter and edges represent dependencies. The reason is that in order to sample a subspace A it might be necessary to previously sample another subspace B so that the filter in charge of sampling B uses the most-likely values gathered from the previous stage. Also, because every SHPF must behave like a single particle filter as a whole, there must be a unique node with no incoming arrows (the initial subfilter) and a unique node with no outgoing arrows (the ending subfilter).

As previously mentioned, some subspace particle filters depend on the result of others if they need the values of the particles it produces, case in which those values are *inherited* from the parent filter. The network formed by all subfilters is incrementally built using a simple rule: if a subspace particle filter C depends on the result of another subspace filter A they are run in a serial fashion (C after A), otherwise, unless C depends on the result of another subspace filter B that is dependent on A , they are run in parallel.

In order to merge the results of parallel subspace filters (converging nodes) it is used the concept of aggregation nodes, where the particles of their parent filter are merged using a *crossover operator* similar to those used in genetic algorithms. The particles resulting from the crossover operation randomly combine the values of the parent filters taking into account for each parent only the subset of the state space that it has optimized.

The paper [Brandao et al., 2006] also provides an algorithm to detect high-correlated subsets of the state space which can be used to decide how to partition the state space and a closed-form algorithm to arrange the subfilters given their dependencies.

8.3.6 Branched Iterative Hierarchical Sampling

Branched Iterative Hierarchical Sampling (BIHS), presented in [Bandouch, 2010], is the result of introducing branching to the combination of the use of annealed particle filtering (section 8.3.2) and partitioned sampling (section 8.3.4). ✓

As opposed to the branching scheme used in subspace hierarchical particle filters (section 8.3.5, in which branches are used for optimizing different subspaces in parallel that are then crossed over and then evaluated again², BIHS branches perform the same optimization using different partitioning patterns (see figure 8.13). This simplifies the merging process of the branches, since all of them contain particles of the same dimensionality which have been optimized for the same state space but with different annealing and partitioned sampling patterns.

The objective of BIHS is also improving the efficiency by reducing the number of particles needed. Despite it might be counterintuitive, [Bandouch, 2010] provides empirical data supporting that due to the use of multiple ways (branches) of estimating the most-likely point of the state space the particle filter as a whole is more robust to noise and partial occlusion, decreasing the actual number of particles needed to estimate most-likely particle.

²With a smaller set of particles in comparison with what it would be needed if sampling the whole state space at a time, hence the computational improvement.

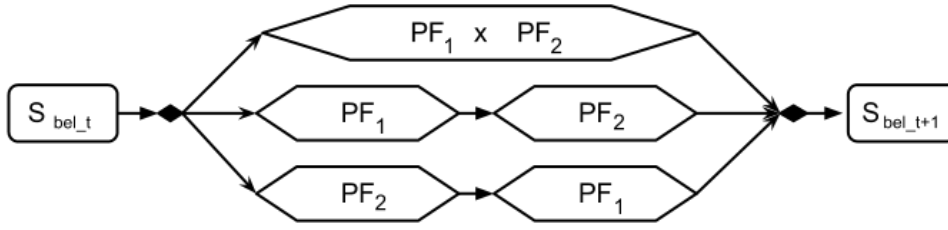


Figure 8.13: A BIHS filter with three branches. Assuming the state space X can be divided in two subsets (X_1 and X_2) there are three possible branches: a) one that samples X_1 and then X_2 ; b) one that samples X_2 and then X_1 ; and finally c) the one that samples X as a whole ($X_1 \times X_2$).

8.4 Discussion

Bayesian filtering is a principled way to update uncertain world representations, accomodating new information in the models as it is available. Due to its ability to implement non-parametric Bayesian filters with complex probability functions (such as those with a dynamic and unpredictable number of maxima), particle filtering is currently one of the most important probabilistic techniques in robotics and computer vision. Depending on the prevailing issue: performing robustly with multimodal or high-dimensional probability density functions, particle filters, MCMC, or a combination of both can be used (since Markov Chain Monte Carlo methods can also be used to implement Bayesian filters). Most of the techniques that allow to perform Bayesian inference are based to some extent in sequential resampling, however the use of Metropolis-Hastings-alike sampling might make these techniques more efficient when the state space is of a high dimension.

As reported in [Neal, 2001], MCMC sometimes fails to sample different modes whilst it generally work better on high-dimensional single-mode distributions. In this respect, when dealing with the low-dimensional models used in this work (high order representations, as opposed to 3D point clouds or grids) the number of particles managed by the filters is dramatically reduced. Moreover, given that the variables to estimate must be supported by measurement data, even in large maps, only the *visible* variables —which can be supported with data— will be included in the particle filtering process. All these factors heavily reduce the number of dimensions that the particle filters have to account for in practice.

Particle filtering has, however, some limitations. As occurred with exact Bayesian filters, the computed belief might take many recursive invocations to converge to the actual area where the model to parametrize actually lays. This depends on how well the probability models are, how appropriately the particle set $S_{bel,t-1}$ is initialized, the size of the belief state space (which make the number of particles needed grow exponentially) and the number of particles itself —of course, the time needed to compute a step of the filter grows linearly with the number of particles. Several improvements to the initial particle filtering algorithm in order to increment its efficiency have been presented: Rao-Blackwellized PF (8.3.1, Auxiliary PF (8.3.3), Annealed PF (8.3.2), Partitioned Sampling (8.3.4), Subspace Hierarchical PF (8.3.5) and Branched Iterative Hierarchical Sampling (8.3.6).

In order to handle noise and occlusion and improve efficiency (see [Brandao et al., 2006, Bandouch, 2010]), branched techniques can be of particular interest. However, branching should be done properly:

- All the variables which are being optimized must have measurement data supporting

them. Otherwise, the noise introduced in the sampling process of the predicted distribution will make the particles perform an uncontrolled random walk. In such cases, since there is no supporting data, all particles will be assigned the same measurement probability and, thus, the same importance, therefore the uncontrolled update.

- The belief space must be the same for all particles in order to avoid degenerate cases (see the example of section 8.2.3.2) or at least, different state spaces must be used with extraordinary precaution. Otherwise, particles carrying more information might be assigned lower probabilities than those with smaller models. That is, assuming that a dynamic state space comprising $X^{(a)}$ or $X^{(a)} \cup X^{(b)}$, it can be possible to find that:

$$\exists x_t^{(a)} \in X^{(a)} : \forall x_t^{(b)} \in X^{(b)} p(z_t|u_t, x_t^{(a)}, x_t^{(b)}) > p(z_t|u_t, x_t^{(a)}) \quad (8.18)$$

This would mean that regardless of the value of $x_t^{(b)}$, the particle filter would evaluate the first option (the one of a higher dimensional state space) as more likely.

As pointed out in [Bandouch, 2010], most of the extensions of the standard particle filtering algorithm suffer from the curse of dimensionality. Most of the techniques reduce the number of particles needed, but it still grows approximately exponentially with the dimension of the state space.

With the exception of partitioned sampling, subspace hierarchical sampling and branched iterative hierarchical sampling, previous approaches lack of a proper mechanism to consider data dependencies among the possible subsets of the state space (such as kinematic dependencies), that can be used to considerably reduce the effects of the dimensionality. However, even these techniques still have some limitations that are of big importance in robotics:

- The data to produce the desired maximum a posteriori estimate is generally considered complete. Depending on the specific problem, robots may need to move their sensors in order to provide a useful estimate (*e.g.*, when modeling a room).
- In some cases there is no possible complete view, so the whole robot might need to move to gather information (*e.g.*, when building high-order representations of buildings).

Interestingly, in those cases, robots need to increase the size of the state space while they explore the environment.

Active perception is not taken into account in any of the previous approaches and sometimes it is necessary to act in order to perceive. For example, there is no reliable method to determine the emotional state of a human, robots need to ask and interact to achieve a reasonably robust guess. In order to overcome these limitations, this work presents *Active Grammar-based Filtering* in the following chapter.

Chapter 9

Active Grammar-based Modeling

9.1 Autonomous robots and active perception

For real applications, it would be desirable that robots would be able to autonomously model and interact with non-trivial entities such as compound objects, people or their environment. Unlike in the earliest experiments of robotics, the floor is not necessarily restricted to texture-less shadow-free surfaces anymore, and objects are not necessarily plain and perfectly shaped boxes [Roberts, 1963]. Nevertheless, the environments in which robots perform their tasks are not made of randomly distributed mass either, so modeling them as three-dimensional grids is far from efficient. Robots can use the readily available a priori information about the environment to build the higher-order, compact representations that robots need for developing complex tasks (see chapter 7). Thus, in order to build actually intelligent and robust robots—especially indoor robots, whose environments are much more predictable—this information must be properly used (see [Gibson, 1986]) not only to process the information acquired by their sensors, but also to act.

The traditional approach used to explore and build high-order representations of the environment in robotics is to develop separate systems to act and estimate the different sets of variables the state space is composed of. For example, a robot creating a high-order model of an apartment would have one or more attentional or exploration systems, a room modeling system, a table modeling system, a mug modeling system and so forth. All these systems are generally separated and are not run in parallel, so it is difficult to estimate joint probabilities related to the known variables that can be observed. The active side of perception is usually handled by another separate system that takes as input the world model and selects the action to take. The information provided by the structured nature of world models (about what can and can not be seen depending on the context) if used, is generally hard-coded. Moreover, as the whole system grows it tends to become considerably complex and difficult to understand and maintain, and eventually turns into a problem itself. Despite this kind of approach can be valid in different contexts and applications, it can be improved.

The previous chapter described some of the most relevant techniques that can be used when building and maintaining world models. Despite in many different contexts they are extremely useful to provide estimations of the most-likely model given the provided input data (for which some of them also estimate the belief distributions), these techniques are not well suited for some scenarios which are very common in real-life robotics applications:

1. Those in which robots can not simultaneously gather information about all the variables the state space is composed of. Despite particle filtering can be used with partial data (if the particle likelihood and transition functions are appropriate), in the best case scenario, only the variables supported by data would be estimated. It is desirable to enable robots to gather input data autonomously.
2. Those in which it is necessary to interact in order to gather information. For example, to determine the emotional state of a human, robots might need to interact with them to get valid input data (*e.g.*, voice, gestures) and achieve a reasonably reliable guess.
3. Those in which the state space is not initially known. Robots generally need to explore the environment to increase the size of the state space or to do it properly, ensuring low uncertainty.

This last issue —coping with unknown and varying state spaces— is a particularly important topic. In fact, its study is one of the main goals of this work (to the point that its title is inspired on this issue). Reversible Jump Markov Chain Monte Carlo (section 8.2.3.2) is one of the few techniques conceived with the purpose of coping with unknown and varying state spaces. However, RJMCMC does not provide any method to reason about what actions —if any— should be executed in order to perceive each object or the contexts in which they might be perceived. It lacks of mechanisms to ensure that the generated models will make sense: the transition function must ensure by itself that it only generates valid models (what can be rather complex).

In the case of Subspace hierarchical particle filtering (section 8.3.5) the issue is only taken into account temporarily, since its aim is not working with different state spaces but to improve the performance of regular filters. Moreover, it can not be said that the particles are located in different state spaces, since their likelihood function is always the same, they are just updated within the bounds of different subspaces of the same state space.

Branched iterative hierarchical sampling (see section 8.3.6) uses different partitioned sampling schemes to get to the same state space, so again, despite the particles are located temporarily in different state spaces, they all end up in the same, initially known state space.

To the date, the only known technique that allowed to work with different workspaces was Reversible Jump Markov Chain Monte Carlo. In fact, several worth-reading works have used it: [Zhu et al., 2000], [Tu and Zhu, 2002], [Tu et al., 2005] and [Han and Zhu, 2009] for rectangle detection, segmentation and scene understanding (see section 9.2), or [Del Pero et al., 2012], used for room modeling.

Even though RJMCMC has been proved successful, comparing samples in different state spaces is extremely hard:

- Sometimes, samples with more variables are easily assigned lower weights unless the additional variables on such particles are very close to the actual model. This requires RJMCMC, or any other similar method to draw a huge number of samples, otherwise assigning high weights to samples with more variables would be unlikely.
- On the other hand, if the system is permissive with particles with new variables for which the data does not fit very well, noise and special cases may make the filter create odd degenerate models for which the data ends up fitting (see example of figure 9.1). In these cases action can be the only way to verify what is going on.

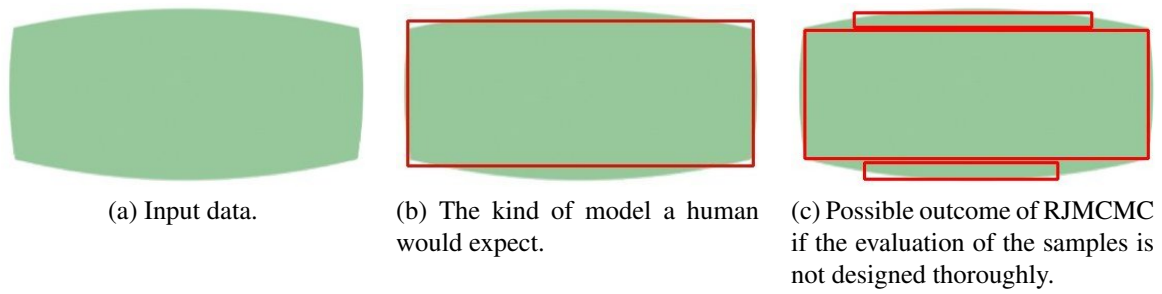


Figure 9.1: A multi-rectangle tracking system where particles can be in different state spaces can result in degenerate cases. Despite that these cases are unlikely, using only Bayesian statistics one can do nothing but ensure that unlikely transition functions are effectively unlikely. The philosophy behind AGM is, when possible, make the robot gather the best data it can to decrease the probability of building wrong models.

The challenge of developing successful RJMCMC-based algorithms relies on designing extremely successful transition and likelihood functions (*i.e.*, $p(x_t|x_{t-1})$ and $p(z_t|x_t)$ respectively) that take into account all the cases that robots may encounter. This is moderately difficult for models of medium complexity such as multi-target visual tracking, but creating these functions for high-order models of realistic human environments is extremely hard: the functions evaluating the joint probability of all the variables given the data tend to be extremely complex, and some of the variables may even not be supported by the input data.

Leaving complexity issues aside, an option would be to use RJMCMC (or its particle-based counterpart) where only the set of variables that can be supported by data is updated at each time, and the actions to take are automatically selected somehow. However, this would entail maintaining samples of high dimensionality, so a huge number of samples would be required.

One of the possible approaches to reduce the dimensionality is to make *crisp decisions* on the model assuming that the values of a specific part of it are true—such as BIHS does—and then continue with other dimensions. In fact, when the robot needs to act—which is considered mandatory for perceiving in a general scenario—the actions might only be coherent with some of the representations. Among other features to support active perception, **AGM** provides a means to automatically reason about what dimensions of the state space should be frozen, and to select the most appropriate filter (or filters) to update the models of the robots. ✓

Despite the previously introduced issues are among the most important regarding active perception in robotics, they are far from being the only challenges that roboticists have to deal with when developing robots able to build and maintain world representations and perform useful tasks. Some of those sources of difficulties are related to software development, such as complexity, readability or scalability. Regarding complexity, it has been successfully handled from different points of view. Control architectures such as those in [Gat, 1998, Volpe et al., 2001, Nicolescu and Matarić, 2002] suggest how to organize control and information flows. Robotics frameworks such as RoboComp [Manso et al., 2010] (see part I), ROS [Quigley et al., 2009], SmartSoft [ScGutierrezhlegel et al., 2010], or OpenRTM [NAIST, 2010] handle implementation issues from a software engineering point of view using component-oriented programming techniques. Some of them also provide model-driven developing facilities: RoboComp [Romero-Garces et al., 2011], SmartSoft [ScGutierrezhlegel et al., 2010], or OpenRTM [NAIST, 2010]. Planners and state machine-based sequencers such as those described

in [Quintero et al., 2011] and [Cintas et al., 2011] contribute to make robot control code more scalable and understandable. However, none of these technologies can be directly used as a tool to support active perception nor to solve the binding of perception and action. The control logic of active perception algorithms still tends to be formed by hard-coded if-then-else constructs that map robot proprioception and its world model to specific perceptual states and actions, which is error-prone. Thus, despite using these technologies makes robotic systems better designed and easier to manage, the complexity of the control logic associated with the perception of the environment is hardly reduced.

✓ Our proposal, *Active Grammar-based Modeling* (*AGM*), intends to provide a means to overcome, or at least mitigate, several limitations:

1. It enables robots to decide what action to take depending on the current context (their world model) so they can receive the information they need and complete their missions.
2. It removes the need to know the state space from the beginning, allowing to generate models of unknown size and topology. It is only necessary to know the grammar rules that describe how the model can be built.
3. It provides a means to incrementally ensure that the model hold is valid according to the grammar (also known as model checking) and other additional perceptive features.
4. This is made in a way that is more scalable and easy to understand in comparison with, less structured, previous approaches.
5. Additionally, since it makes use of grammar rules describing how the environment can be updated, including not only changes triggered from the inspection of new areas or third-party modifications but also the modifications performed the robot itself, AGM can also be used to support general planning and task execution.

Simply put, the main idea is to use a set of grammar-controlled model-modifying agents that may perform physical actions. In this context, agents are assumed to perform simulation-based modeling techniques such as particle filtering or MCMC. Each of the agents are in charge of one or more dimensions of the possible state spaces and are activated or deactivated by a grammar-based controller according to the mission of the robot.

Unlike, Branched Iterative Hierarchical Sampling, which used branches to allow estimating the most-likely values of a fixed model in a fixed finite number of ways ([Bandouch, 2010], see section 8.3.6), AGM uses grammar rules to decide which branches (particle filters taking into account different subsets of the model) should be active and which modifications may be possible to perform to the model. Thus, Active Grammar-based Modeling combine —not replace— the previous approaches depending on what robots perceive and their objectives. By associating different robot active-perceptual behaviors to the grammar rules, the resulting system is able to generate coherent models of initially unknown state spaces and compute what actions should be taken in order to get to a target world state if there is any.

9.2 Grammars in robotics

✓ A *grammar* is a theoretical tool used to describe the rules governing the formation of specific sets of structures. String grammars, those that deal with building strings (generally used in nat-

ural and computer languages) can be seen as a specific case of graph grammars, that generalize the concept of string grammars so that productions can also be applied to graphs. In fact, strings can be seen as undirected graphs such that all nodes (characters), except those at sentence endings, have an edge linking them to the adjacent character on their right. Thus, graph grammars extend string grammars in order to support indefinitely complex connection patterns.

Grammars have been used in robotics and artificial vision for a wide range of applications. In [Bauer and Wilhelm, 2006] it is proposed a new algorithm for graph grammar verification, that is, to verify the properties of the graphs generated by using the rules of a specific grammar. In this work graph grammars are defined as a set of transformation rules and an initial graph, where vertexes are labeled but edges are not. [Bousassida et al., 2010] describes a set of graph grammar rules that can be used to achieve self-configuring adaptable software architectures. In this work graphs are of fixed order. In [Smith et al., 2009] it is proposed a similar approach for coordinating multi-robot systems where robots are represented by graph vertexes. The graph, which is shared by all the robots of the system, is also of fixed order. Coordination is achieved by modifying the linking pattern and the label (*i.e.*, role) of the robots.

In [Zhu et al., 2000, Tu et al., 2005] and [Han and Zhu, 2009] the authors present a series of related works. In [Zhu et al., 2000] and [Han and Zhu, 2009], an attribute graph grammar is designed in order to parse images taken from man-made scenes. It uses projected 3D rectangles as terminal nodes and different rectangle layouts as production rules. Bottom-up and top-down mechanisms are used in order to improve rectangle detection and parsing. Since different possible models can explain input images, the algorithm chooses the one that maximizes the posterior probability or minimizes a descriptor length. In [Tu et al., 2005] a similar approach is used for segmenting and recognizing generic scenes. In [Lin et al., 2009] it is used a similar approach to the one described in [Han and Zhu, 2009] for representation and recognition of objects. In this case, both the set of primitives and production rules are wider, but the foundations are the same. These approaches have two main differences from what is proposed in this work: a) they use static input data, which is a very hard restriction for robotics; b) they are based on string grammars instead of graph grammars, reducing the possible applications.

In [Hasemann, 1994], graphs are used to describe plans. It also covers how plans can be dynamically modified as sub-tasks are accomplished or conditions change. Grammar rules are proposed in order to modify the current plan.

In [Siskind et al., 2007] *Spatial Random Tree Grammars* (SRTG) are proposed for image parsing. SRTG's are context-free grammars with at most two children in which rules are labeled with information for determining the spatial distribution of their nodes (*i.e.*, vertically or horizontally distributed). While in string grammars this is not necessary (*i.e.*, productions are always horizontally distributed), it guarantees the unambiguous interpretation of parse trees from graph grammars. In this work, a probability distribution is also associated to production rules so the probability of a specific parse can be estimated.

In [Gupta et al., 2010] a related work for understanding still images is presented. Despite it is not explicitly grammar-based, it uses a generative approach, and the result of the algorithm is a parse graph. The focus of the paper is to provide an image understanding algorithm consistent with physic laws, going beyond unembodied image labeling.

Despite several works use grammars to generate representations of the contents of images ([Zhu et al., 2000, Tu and Zhu, 2002, Han and Zhu, 2009, Lin et al., 2009], all of them approach perception as a passive process where data is static and complete from the beginning. This work proposes to use graph grammars to describe how the representation of the world

may be updated and which *behavior* should the robot adopt if such changes are desired. The particularities of the grammars proposed in this work do not only allow us to define how the world model can be transformed but also to achieve other goals such as: deciding what the robot should do to obtain the necessary information; incremental model checking (the architecture guarantees that the models generated using it are created according to the grammar); or decreasing programming errors (the developer does not need to hard-code the grammar or what the robot should do in a general-purpose language, but in a higher level language that can be graphically visualized).

9.3 Active grammar-based modeling system description

Now that the goal, the known approaches and some of the most relevant grammar-based techniques have been introduced, this section provides an iteratively detailed description of the design of AGM. The final diagram is shown in figure 9.5.

A first approximation to how a AGM would look like can be found in figure 9.2. There is a set of different particle filters that propose modifications to the model and a *grammar-based controller* that selects the most likely modification that complies with the grammar of the world. The controller would also enable and disable the filters according to the context (some modifications might only make sense in specific contexts). Some of the filters would only update the values of the current model (named *subspace update PF* in the figure), others would modify its structure (*grammar rule PF* in the figure).

At first sight, it is quite similar to RJMCMC: the filters only modifying the values of the current model (*subspace updaters* in figure 9.2) would correspond with *update moves* in RJMCMC terminology; the filters modifying the structure of the model would correspond with *birth*, *death* and *merge moves* in RJMCMC terminology. However, there are some important differences:

- Model modifications are controlled by a grammar description: this ensures that only valid models can be generated. In RJMCMC the programmer has to ensure this on its own.
- AGM is an iterative approach where the model is continuously fed back and updated, generally with varying data (not an action-decoupled algorithm that is run with static data). This might be seen as a disadvantage, since sometimes atomic changes might get stuck in a local minimum; however, the filters can also propose non-atomic changes (e.g., implementing a RJMCMC-alike particle filter that makes different modifications at a time within some of the filters of figure 9.2).

Since, to the grammar-based controller all filters can be seen the same way (model modifiers), figure 9.2 can be simplified by ignoring the different nature of the filters. Now, assuming that actions can also modify the representation of the environment we get to the diagram in figure 9.3. Again, the diagram can be simplified by considering any update is triggered by a *model-modifying agent* (regardless of whether the change is made by an action-only agent, a model-modifying agent or a symbol-updater agent). This leads to the diagram shown in figure 9.4.

Previous figures provide conceptual descriptions of AGM. However, from a design point of view, the whole system is composed of four elements (see figure 9.5):

- The **world** in which the agents sense and operate —potentially through HAL software components that are not included in the figure.

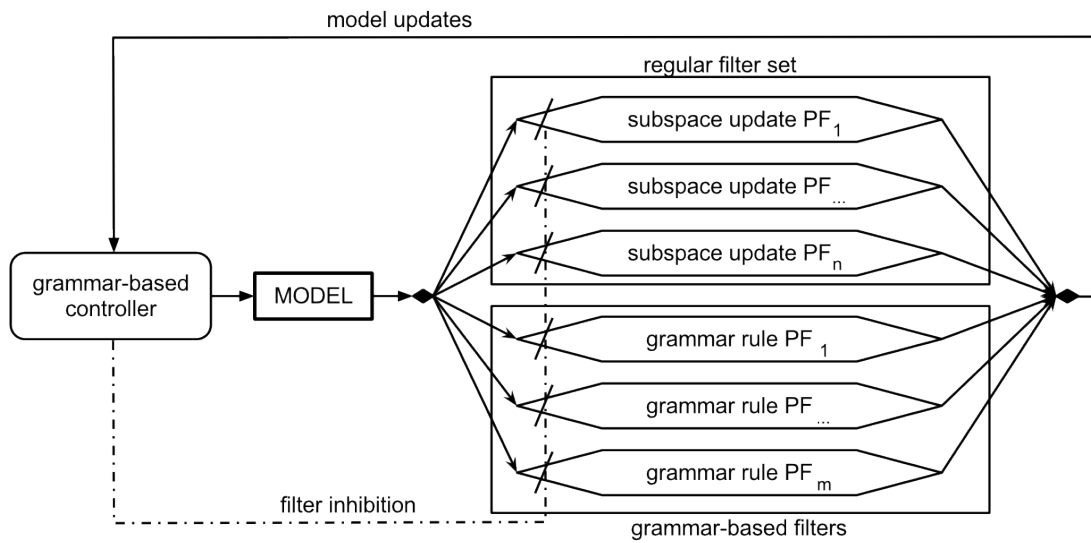


Figure 9.2: A first approximation to the information flow of AGM.

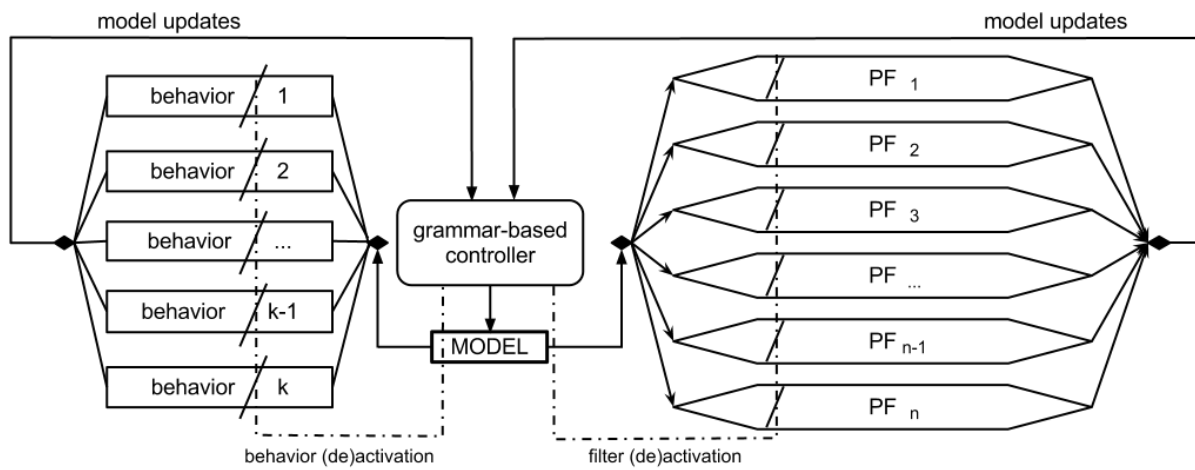


Figure 9.3: A more detailed description of the information flow in AGM.

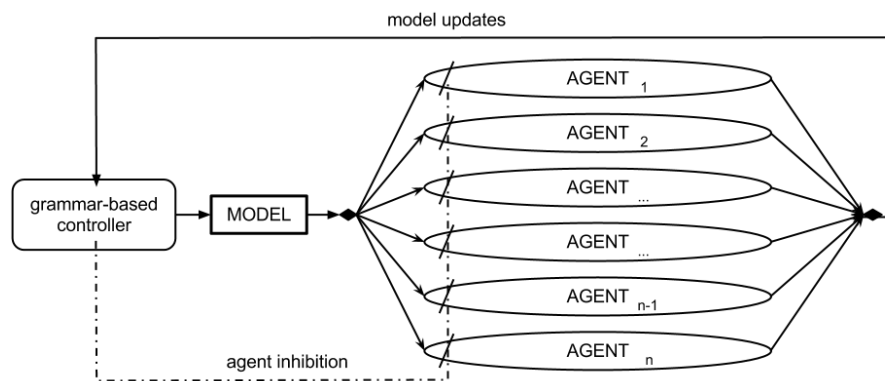


Figure 9.4: Final conceptual description of the information flow in active grammar-based modeling.

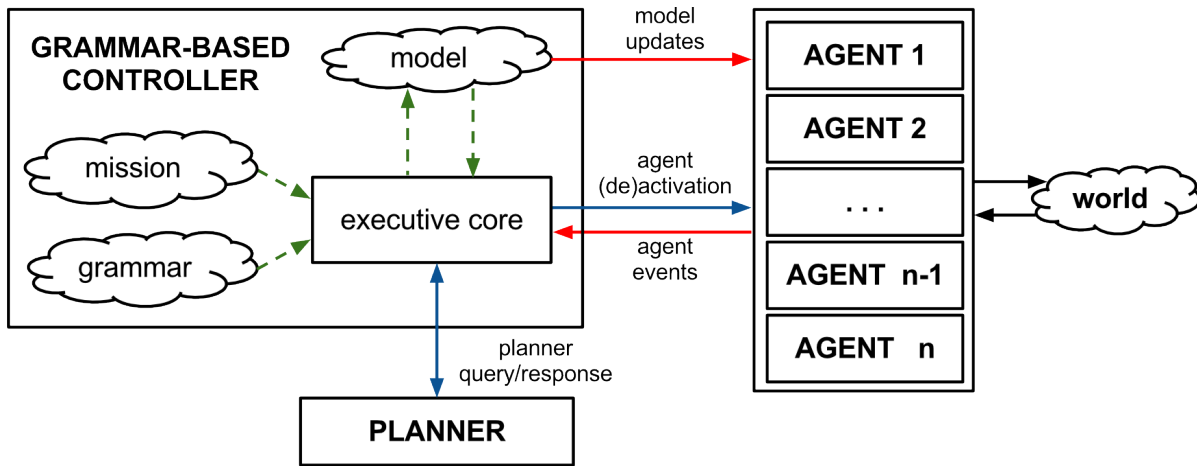


Figure 9.5: Design diagram of active grammar-based modeling. The mission, grammar, executive and the planner would correspond to the *grammar-based controller* of figure 9.4. The world and data are represented in clouds while software elements are represented in boxes.

- The **agents**, that move the robot and update and modify the representation of the world according to the acquired data.
- A **grammar-based controller** in charge of (de)activating the agents and accepting or rejecting the model modifications they propose depending on whether or not the modifications are valid according to the grammar. It is composed of an *executive core*, the model which it maintains, the mission of the robot and the grammar that describes the possible changes that may occur in the model.
- A **planner**. Used by the controller to **a**) plan what to do in order to complete the mission of the robot; and **b**) check whether or not a modification is valid.

As explained in chapter 7, we want the the world **model** to be a graph of attributed symbols. Despite the attributes are *semantic* properties (not taken into account in a syntactic level), the symbols and their connection patterns are updated according to a world **grammar** (see example rule in figure 9.6). These grammars can be graphically described and compiled into the PDDL language [Ghallab et al., 1998] (the de-facto standard AI planning language) using two tools specifically developed during the course of this work: *AGGLEditor* and *aggl2pddl*. Once the grammars are described in PDDL, the **executive** of the **controller** can use them to query an AI **planner** for plans —expressed as an ordered list of model transformations— that can take the current model to a target one. This is used with two purposes:

- To check if a model modification is valid. Case in which the target model would be the modification proposed.
- To plan a set of model transformations that would make the robot accomplish its mission (in this case the target model would be the **mission**) and configure the agents accordingly by associating each of the grammar rules with a set of agent configurations.

Agents interact with the **world** by sensing or activating the actuators of the robot and proposing modifications to the **executive**. The executive evaluates the proposals and accepts and broadcasts to the agents those which are valid according to the grammar.

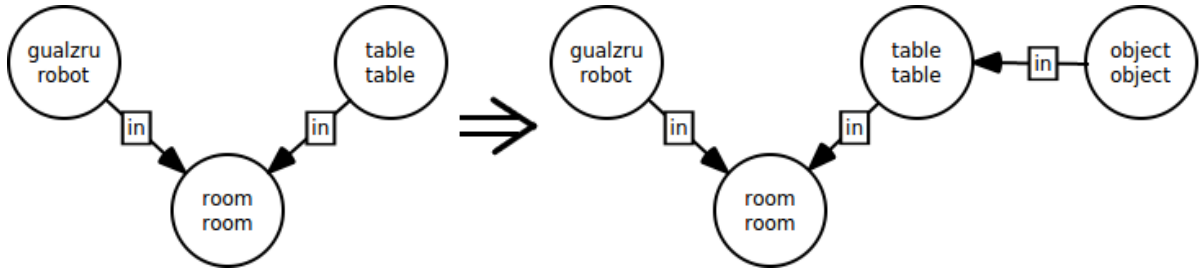


Figure 9.6: Graphical representation of an AGGL rule. This rule, which is used to include new objects in a table located in the same room as the robot, belongs to the experiments described in chapters 17 and 18.

The remaining of the chapter provides deeper descriptions of several aspects of AGM: section 9.4 describes how to specify grammars for AGM using the Active Graph Grammar Language language; section 9.5 provides a simple grammar example (more complex grammars are shown along experiments in chapters 17 and 18); section 9.6 details additional advantages that AGM provides; section 9.7 details several interesting implementation issues; finally, section 9.8 provides a discussion about AGM, their strengths and weaknesses and future works.

9.4 Describing grammars

The grammars in AGM are described using the *Active Graph Grammar Language* (*AGGL* [✓]). AGGL is a visual domain-specific language (see section 3.4 for an introduction to domain-specific languages) specially designed to describe how graph-based models can be generated (symbolic and, with limited support, hybrid¹). Instead of using the textual DSLs proposed in chapter 5, here we suggest to use a visual language because of the complexity that would entail for a human to understand graph structures described in text files. However, if a developer wants to edit the file manually, it is still possible to do it using a regular ASCII editor. Additional tools are provided so that descriptions based on this language can be translated to computer code and used for different purposes, such as to link perception and action, to perform incremental model checking or to improve perception robustness.

As any other grammar, graph grammars are sets of grammar rules —also known as graph rewriting rules in some contexts— where each one describes a valid transformation that the representation may suffer. They are described using pairs of patterns $G_1 \Rightarrow G_2$, meaning that the pattern G_1 can be substituted with pattern G_2 (see figure 9.6). The left-hand side of the pair is usually referred to as LHS, and the right one as RHS.

In AGGL, each symbol is represented by a node with two strings, one for the type of the node (in the bottom of the corresponding circle) and one for the alias (in the top of the circle) used to identify nodes (see figure 9.6). Relationships between the elements of the model are represented by a labeled link. There can be more than a link between two different nodes, but those with the same label are ignored.

As opposed to string grammars, graph grammars lack of a generally accepted formalism for specifying their behavior. Given the absence of a generally accepted formalism for graph gram-

¹ It must be noted that, despite the symbols of these models can be attributed with additional metric properties, these are ignored by the grammar since they do not exist at a syntactic level.

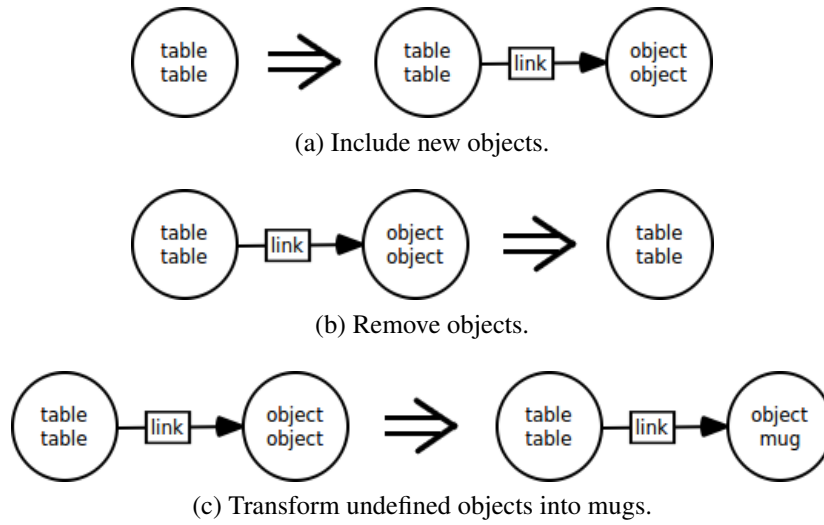


Figure 9.7: Examples of simple graph-grammar rules.

mars, we chose to use a slight modification of the *double push-out* formalism [Heckel, 2006]. The behavior of rules in AGGL is the following:

- A rule can only be applied if it is found a match between the elements and connection patterns of the LHS of the rule with the ones in the model.
- In order to apply a rule, the nodes and edges that are present in the LHS but not in the RHS are removed, the ones present in the RHS but not in the LHS are created, those appearing in both or no side remain.
- If the origin or end of an edge correspond to a node that will be removed, such edge will also be removed.

For example, rules can be used to express the possibility that tables can be associated with new objects in the robot world model (figure 9.7a), that objects can disappear from tables (figure 9.7b), or that objects in tables can be converted into mugs (figure 9.7c):

Figure 9.8 provides a screenshot of, *AGGLEditor*, the application which is used to describe grammars. The upper part of the figure is used to show the left and right hand sides of the rules. Rules can be added using the main menu and selected for viewing using the list on the left (labeled as *Rules*). Rules can be edited selecting the tool in the list on the center (labeled as *Tools*). Finally, the list on the right (*Configuration name*) is used to select a behavior for the current rule if there is any.

9.4.1 Associating active-perceptual configurations to rules

AGGL is not only used to define the rules of the grammar but also the behavior that robots should adopt when they expect to trigger them. This is done in three steps:

- first the possible behaviors and agents are defined,
- then, a list of the possible configurations of each agent is specified,

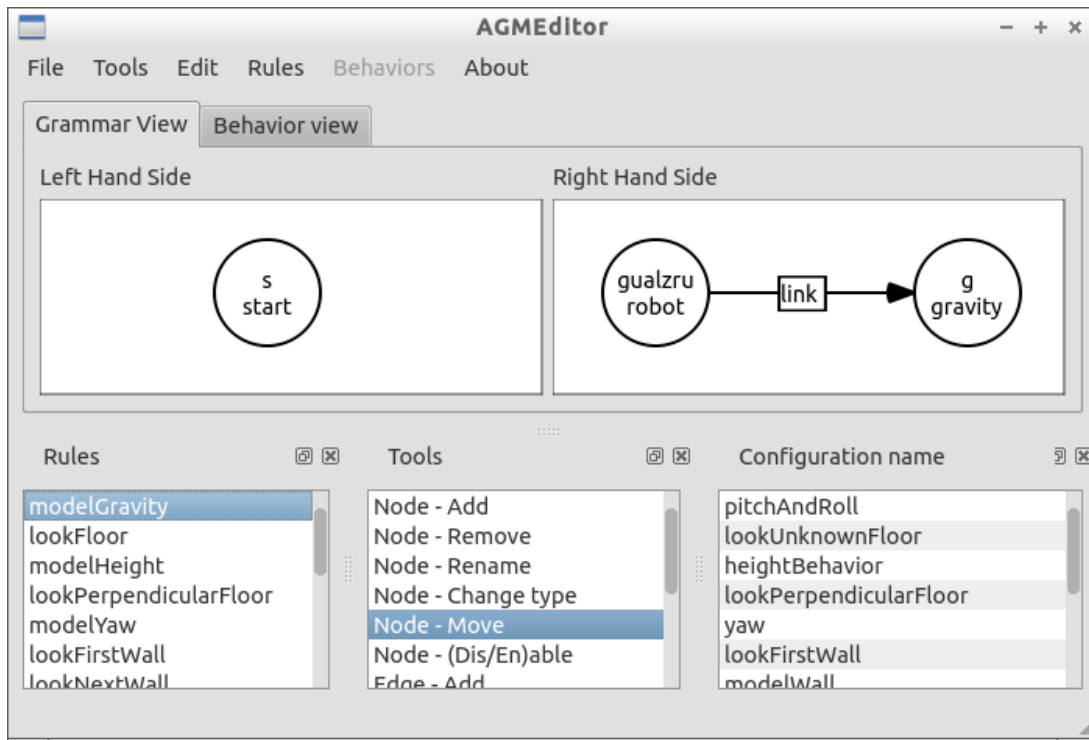


Figure 9.8: Screenshot of the *AGGEditor* application in the grammar view mode (note the left tab is active).

- once the agents, the behaviors, and the possible configurations of each agent are defined, each behavior must be associated with a specific configuration value for each agent,
- finally, rules are decorated with their desired behavior.

Figure 9.9 provides a screenshot of *AGGEditor* while defining the possible behavioral configurations. The upper part of the user interface shows the configuration of each agent for the currently selected behavior. The behaviors (that can be added using the main menu) are selected by clicking in the leftmost of the lists (labeled *Configuration name*). Agents and states for the current agent (on list *Agents*) can also be added using the main menu.

9.4.2 Dealing with qualitative metric information

Most PDDL versions, which is the language in which AGGL files are compiled into, do not support working with numbers—it must be noted that the original PDDL language has numerous revisions and alternatives. Even if they did, there are cases in which the planner itself will not suffice and it is necessary to include *expert* information in the model. For example, when assigned a navigation task, robots must evaluate if it is possible to get from one room to another (probably as the precondition to execute an action). Despite it would be theoretically possible to implement such thing in PDDL, AGM provides an alternative: using the AGM agents to include and remove symbols and links to ensure such preconditions are met (or the opposite). The existence of specific links or symbols can be easily used in the left-hand side of the grammars to avoid rules to be potentially triggered.

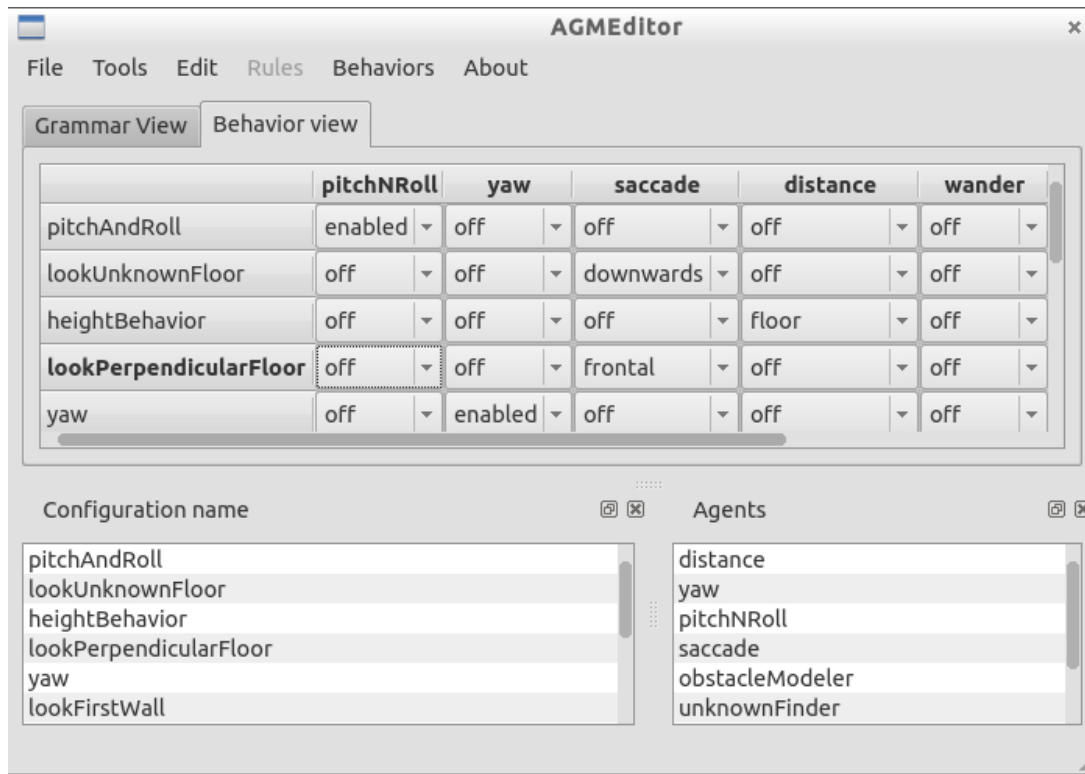


Figure 9.9: Screenshot of the *AGGEditor* application in the behavior view mode (note the right tab is active).

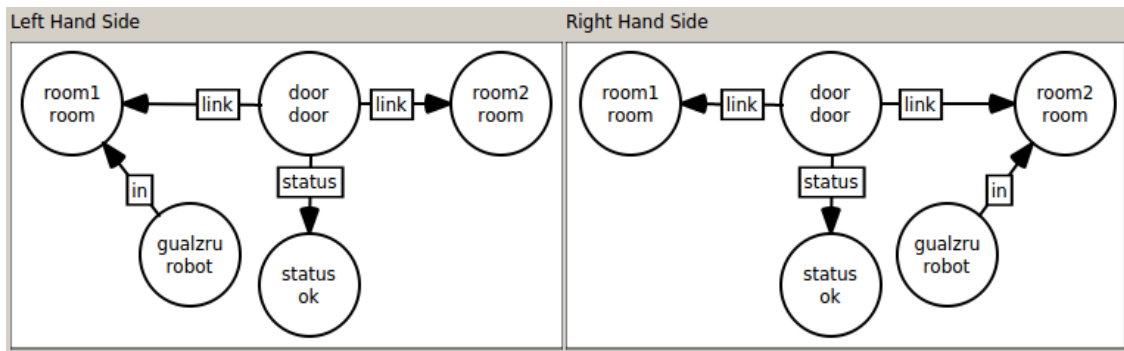
Following the previous example, it would be possible to take into account the status of a door —free or blocked— in the rule associated with door traversing in order to make the system avoid trying to cross blocked rooms. This could be done using the following two rules (see figure 9.10):

- A robot can traverse a door as long as it is free. In figure 9.10a it is denoted using a *ok* symbol attached to a door symbol.
- Doors can be set to free status by attaching *ok* symbols to them. See figure 9.10a for a formal example.

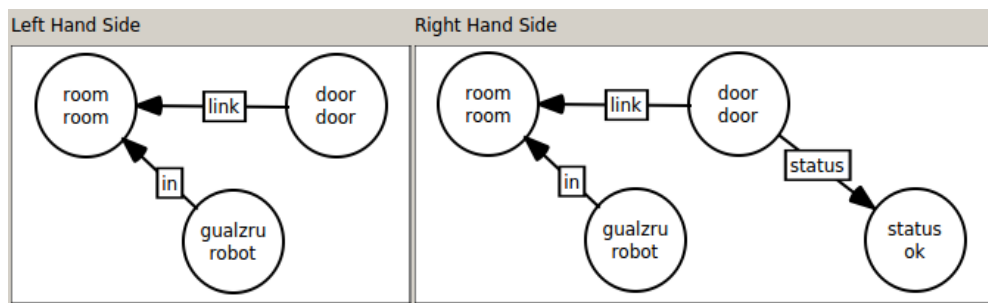
In a real scenario it would also be necessary to include a rule to remove the “free status” of doors when the robot realized they are blocked, however, for this example these two rules are enough. Of course, these rules would be triggered by agents that should be active while the robot tries to cross doors. Generally, it would also be advisable to make such agent or agents active previously, in order to evaluate if a robot can navigate to the target room before actually trying it.

9.5 A simple example

At this point we can describe how AGM would work with a complete but extremely simple example (see the experiments shown in chapters 17 and 18 for real use cases).



(a) Robots can traverse doors.



(b) Robots can model the status.

Figure 9.10: Example of how to include qualitative metric-related information in the model.

9.5.1 Grammar

Consider an AGM system with the following three-rule grammar:

$$\begin{aligned}
 S_1 &\Longrightarrow room_1 \} \text{ modelRoom} \\
 room_1 &\Longrightarrow room_1 \rightarrow_{has} table_2 \} \text{ lookForTables} \\
 table_1 &\Longrightarrow table_1 \rightarrow_{has} mug_2 \} \text{ lookForMugs}
 \end{aligned} \tag{9.1}$$

The first rule substitutes the start symbol with the symbol of the room, the second rule introduces tables, and finally, the third rule introduces mugs in tables.

9.5.2 Agents

Consider also the following five agents:

roomModel: Assuming the robot is moving and acquiring new data appropriately, models the room and introduces or updates the values of the *room* symbol. It can be configured to introduce the symbol of to update its values.

wander: Makes the robot move randomly.

tableModel: Models and introduces in the representation new tables.

findObjects: Makes the robot look towards not modeled objects. It can be configured to look for objects in the floor (tables) or objects in tables (mugs).

mugModel: Models and introduces in the representation new mugs.

Table 9.1: Example table of agent configurations.

| conf. \ agent | roomModel | wander | tableModel | findObjects | mugModel |
|----------------------|-----------|--------|------------|-------------|----------|
| modelRoom | create | active | idle | idle | idle |
| lookForTables | update | active | active | activeFloor | idle |
| lookForMugs | update | idle | idle | activeTable | active |

9.5.3 Configuration

Table 9.1 describes how agents are configured depending on the configuration:

9.5.4 Execution

Assuming the goal of the robot is to find a mug, the first thing the executive (see figure 9.5) would do is to ask the planner for a sequence of rules that would take the robot to the goal. The planner would provide a plan in which the first rule (action in PDDL terms) would be the first of the grammar, so the executive would configure the agents according to the configuration associated with the rule (*modelRoom* in table 9.1). That would activate agents *roomModel* (in create mode), *wander*, and deactivate the rest.

At some point, the agent *roomModel* would model the room and propose to the executive a new representation including the model of the room. The executive would verify the proposal and, assuming it is correct, it would accept it and re-initiate the process: ask the planner for a plan, reconfigure the agents and wait for modification proposals.

In the second iteration, the planner would find a plan with the second rule in the first position of the plan sequence. Thus the executive would configure the agents according to configuration *lookForTables*. This time *roomModel* would update the values of the room symbol (it is worth noting that symbol values updates are accepted automatically) as the robot moves (since *wander* would be active) and it would be agent *mugModel* the one proposing a change: the inclusion of a mug associated with the previously modeled table in the model (which at that point would include the room, the table and the mug in the table). Such model would be accepted and the forth iteration of the executive would start.

After that point the robot would have accomplished the mission, so the plan would be empty and no agents would be active.

A non-obvious interesting extension of this setup would be to use

9.6 Using Active grammar-based modeling

Active Grammar-based Modeling can be used to achieve different interesting perceptual phenomena which, so far, have only been concisely explained. This section provides a deeper explanation of the different possibilities of AGM.

All published grammar-based perception techniques can be classified in one of the following types: a) bottom-up parsing, b) context-aware perception restrictions, c) model verification, and d) covert perception. In addition to these, AGM can also be used to generate and execute general-purpose plans.

9.6.1 Bottom-up parsing

Regular particle filters and parsing algorithms are designed to work with static and complete input data (despite in the case of incomplete data some algorithms may be able to provide partial models). Robot perception generally entails base and camera movements that allow robots to sense their whole environment. Since these movements change the input data, common approaches can not be used to model or parse it.

Bottom-up parsing in AGM is performed by generating missions that include new symbols in the model. When the robot is not performing any critical task, it should activate agents that detect non-modeled entities in the world and create new symbols for the unknown objects.

Despite is not mandatory, the active philosophy of AGM recommends modeling objects in, at least, two stages. First, when an unknown object is detected, it is recommended to include symbols marked as *unknown* so the robot can generate specific plans to approach and model objects appropriately, adopting a favorable points of view. Only then, the grammar should activate the agents that the grammar designer considered appropriate that transform *unknown* symbols to any of the symbol types used in the grammar.

9.6.2 Context-aware perception restrictions

Graph grammars express how graphs (models in our case) can be built. To some extent, by doing so they also describe how they can not be built. By restricting which world elements can be perceived in each moment according to the limitations imposed by the grammar on the context, the number of false positives (*i.e.*, misrecognized world elements) can be reduced. In order to illustrate this, consider the grammar example shown in equation 9.2 (where subindexes represent the identifiers of the symbols).

$$\begin{aligned} torso_1 \implies torso_1 \rightarrow_{has} head_2 \} modelHead \\ torso_1 \rightarrow_{has} head_2 \implies torso_1 \rightarrow_{has} head_2 \rightarrow_{has} nose_3 \} modelNose \end{aligned} \quad (9.2)$$

If one of the agents of a robot using this grammar thinks it is a good idea to attach a nose to a torso, the grammar would avoid such invalid change. This is because given the grammar and the current graph model, the planner used by the executive would report that there is no applicable rule (or plan) that would directly link a nose to a torso.

A classic example of context-aware restriction in humans can be found in [Torralba et al., 2010]: the same visual input that in the proper context is recognized as a specific object is not when the context varies. Despite the work presented in [Torralba et al., 2010] is not grammar-based, AGM have the power to support it.

9.6.3 Model verification

Model verification, also known as *model checking* is a research field on its own. It consists on verifying that a model complies with a predefined set of conditions. Since AGM incrementally verifies each change —using the grammar rules as the conditions— we make sure that any model generated using AGM is valid.

9.6.4 Covert perception

Covert perception is one of the most interesting applications of graph grammars. In poor conditions, such as when parts of the object to detect are occluded or there is too much noise, bottom-up object detectors tend to decrease their effectiveness dramatically. Grammars are also a valid framework to enable covert perception [Svensson et al., 2009]. The information they provide can be used to influence bottom-up perception—not just to filter its output—by enforcing the detection of specific parts of the model. Thus, it not only reduces false positives in object detection but also reduce false negatives.

Consider the extension of the grammar shown in equation 9.2 with the following rules:

$$\begin{aligned} torso_1 \implies torso_1 \rightarrow_{has} unseenHead_2 \rightarrow_{has} nose_3 \} \quad bigNose \\ torso_1 \rightarrow_{has} unseenHead_2 \rightarrow_{has} nose_3 \implies torso_1 \rightarrow_{has} head_2 \rightarrow_{has} nose_3 \} \quad forceHead \end{aligned} \quad (9.3)$$

Introducing these rules we actually enable the robot to model the head using two different strategies: a) by regular detection using an ad-hoc detector/modeler and; b) by detecting a nose, which makes evident there is a head supporting it that should be modeled. It must be noted that, in the second case, the information provided by the modeled nose can be used by a “head detector” to improve its accuracy.

Other examples of this type of techniques can be found in [Zhu et al., 2000, Tu et al., 2005, Han and Zhu, 2009] or [Lin et al., 2009]. In the cited papers, the phenomena is not implemented exactly as in the previous paragraph. They consider a linear grammar-based heuristic approach in order to *hallucinate* elements. However, the principles are similar and using AGGL we could benefit from code auto-generation and better understanding of the grammar.

9.6.5 Planning and execution

Action selection has already been covered in section 9.6.1, it consists on deciding what a robot should do next depending on its goals and context. Specifically in the context of AGM, action selection deals with determining the best set of agent configurations that should make the robot closer to satisfy its goals. Despite in section 9.6.1 it was considered for scene understanding, AGM can also be used for general-purpose planning and execution using the previously covered mechanisms. The only requirement is that all the actions robots may perform and their consequences should be included in the grammar. Thus, the world model representation and the mission (described in terms of a partial or total target world model) can be translated to PDDL and the same AI planner can be used to find a plan that satisfies the mission. The result of the planning is a list of rules that would take the current world model to the target state (the one that defines the mission). Execution is finally solved by selecting the first rule of the list and using its associated configuration to configure the agents appropriately.

9.6.6 Intersecting filters

When using AGM there does not exist a global particle filter but a set of specialized filters covering specific dimensions and do not share sampling information between them. Sometimes filters work better on specific cases and it is convenient to design several filters that are activated exclusively depending on the context. For example, there might be a filter that works well on

empty rooms but that provides poor results in the presence of obstacles. In such case, it would be interesting to use one or another depending on the context. The different filters can be selected using AGM by imposing different preconditions in the LHS of the grammar rules.

9.7 Implementation issues

9.7.1 Tools

Creating AGM-based systems and designing and translating graph-grammars to languages supported by state-of-the-art AI planners could be error-prone and time-consuming if the appropriate tools were not provided. During this work several tools have been developed in order to ease these tasks.

AGGLEditor is a graphical editor for AGGL files. It helps developing active graph grammars in a human-understandable way. Figures 9.8 and 9.9 provide screenshots of the two view the editor has: the grammar view, used to describe grammar rules and associate the behaviors with the rules (figure 9.8); and the behavior view, used to describe how agents are configured in the different behaviors (figure 9.9).

There are also two tools related to code-generation. **aggl2pddl** is the tool used to translate AGGL files to PDDL so the grammars can be used along the planners to support perception. **aggl2cpp** generates small C++ files that can be used when developing AGM-based systems.

9.7.2 AGG to PDDL translation

The translation from Active Graph Grammar Language to PDDL is done by creating a PDDL action for each grammar rule. The main challenge of this process is that PDDL requires a closed-world, so actions can not include new symbols in the world. This limitation is overcome by including in the problem definition a set of *unknown* symbols that can be transformed in *known* symbols as they become necessary. These symbols are arranged in a stacked fashion in order to limit the combinatorial explosion that the planner may suffer when searching for solutions.

Translating the effects of rules on edges is straightforward. For each rule it is analyzed all of the edges in the rules and are classified in three groups: those that remain unchanged, those that are created after applying the rule and those which are removed.

The edges of each of the groups are treated correspondingly: remaining edges are ignored, new ones are created by the effect of the PDDL action, and removed ones are negated.

Handling nodes is slightly challenging. As happened with edges those nodes that remain are ignored, they are just required to exist in the precondition of the PDDL action. Removed nodes are required to exist in the precondition as a regular symbol (as their corresponding edges did). In the effect part of the actions their type is removed and the symbol is stacked. New nodes are required to exist in the precondition in the stack. In the effect part of the actions new nodes are unstacked and assigned a type.

9.7.3 Component-oriented implementation

Active Grammar-based Modeling is a principled and well-structured way of designing the perceptual system of robots. It is, however, of greater interest if the distributed nature of AGM is

taken to an implementation level using the component-oriented techniques presented in part I. Implementing each of the active elements of diagram 9.5 as actual independent software components provides several desirable features:

Computation distribution: Using component-oriented programming is a simple way to distribute the different tasks in different cores and computers.

Decoupling: Dependencies between the different subsystems of monolithic systems, tend to create undesirable side-effects. Distribution helps decreasing them (*e.g.*, blocked waits in the subsystems or their user interface).

Robustness: If an agent goes down the whole system does not necessarily go down. The remaining modules can analyze the status of the system and, if desirable, reinitiate the needed components.

Flexibility: Implementing the system using component oriented programming drastically improves the flexibility of the system. It makes easier to prototype, substitute or include new components in the system and enables roboticists to implement each module in a different programming language

Besides all the advantages of programming-oriented component, using RoboComp provides additional advantages:

- Using RCDSLEditor (described in section 6.5) new agents can be automatically generated in less than five minutes.
- RCLogger (section 6.4) allows to log messages and the interactions between the agents, the executive and the planner for debugging or any other purposes.
- RCInnerModelSimulator (section 6.7) can also help prototyping and finding bugs in the agents.

9.8 Discussion

Active Grammar-based Modeling has been proposed as a solution for robot active perception based on a domain-specific language used for define graph grammars and specify the robot behavior that is supposed to maximize the probability of triggering each rule.

It has been described how to achieve different perceptual phenomena:

- Bottom-up perception
- Context-aware restrictions
- Model verification
- Covert perception

It has also been described how AGM can be translated to PDDL and how using component-oriented programming —and RoboComp specifically— would help designing and improve AGM-based systems.

The current implementation of the parser and compiler of AGGL only supports positive patterns. It would be desirable being able to specify negative patterns in the left-hand side of the rules as described in [Manso et al., 2012]. The implementation this feature in AGM is left for future works.

A possible criticism in comparison to RJMCMC is that the scope of the particles in AGM is limited to each agent. However, AGM does not restricts agents to atomic changes in the model. When considered appropriate, an agent can implement RJMCMC as long as the changes in the model do not violate the grammar (of course, if they do violate the grammar the executive wont accept such changes). In fact, since using AGM makes available an explicit description of the grammar —as opposed to embedded in the code—, agents working with non-atomic changes can use the description in order to guide the sampling process. Additionally, using AGM enables robots to act and perceive scenarios that require moving the robot, their sensors, or any other kind of interaction. It must be highlighted that AGM is an extension, not a replacement for any of the variations of particle filtering or Markov chain Monte Carlo.

Another advantage of having explicit descriptions of the grammars of the environment is that they can also be used for learning purposes, since it is possible to include new rules and agents dynamically. However this topic is left for future works.

Chapter 10

Grammar-based Cognitive Subtraction

Particle Filtering (PF) and Markov Chain Monte Carlo (MCMC) (described in chapter 8) are very useful tools to fit models to sensor data. Robots frequently need these models to reason and select the actions that lead to the fulfillment of their assigned tasks. Many current techniques combine these top-down approaches with bottom-up ones to provide fast and accurate results. Generally, bottom-up processing transforms raw data into more manageable representations that can be efficiently used by top-down model fitting algorithms. A proper combination of both approaches provides a good trade-off between the need for restrictions of the former and the low efficiency of the latter.

Despite the remarkable usefulness and range of applications of the previously mentioned techniques, they also have limitations when used by robots for complex tasks. Active Grammar-based Modeling, presented in chapter 9, was proposed as a solution for several active perception-related limitations. Among other uses, AGM allows robots to decide what actions to take, which world modifications are valid or how to configure the agents that should be active. However, there is another important limitation that AGM can not handle by itself: where to find unknown objects. When a robot is commanded to find an object and fit a model to it, even assuming it is actually looking at it, the robot should first perform a tentative segmentation of the object. If there is not enough a priori knowledge to segment the object to some extent, even the most effective modeling methods will get in trouble (*e.g.*, when the number of outliers is high or in the presence of other suitable objects that may or may not have already been modeled).

These issues are usually bypassed using very simple scenarios where all but the relevant content is removed. A typical workaround is to work with scenes in which robots only see a few isolated objects on a table, and the floor, walls, furniture and people are out of sight (see [Rusu et al., 2009], [Holz et al., 2011] or [Richtsfeld et al., 2012] for examples). In such cases, specially when using point clouds, segmenting the objects in the scene is trivial since robots only have to remove the points corresponding to the biggest plane on the scene—the table—and perform an euclidean clustering. The fundamental limitation of the *objects-in-table* scenario is not using a priori information about the world, which is generally a good thing, but the assumption of a very specific and simple structure. While it is interesting to use simple scenarios in the research of static object perception, autonomous robots require active perception procedures that can lead to more sophisticated incremental segmentation and modeling algorithms. The solution proposed in this thesis is to interleave segmentation and modeling, incrementally building the representations of the environment as new world elements are detected and modeled. Possible new objects are detected by segmenting the differences between what

robots expect to perceive and what they actually perceive. The technique presented here for segmenting possible new objects, which has been named *Cognitive Subtraction* (*CS*), follows the general idea suggested, among many others, by Cotterill [Cotterill et al., 2001] that rational behavior is “internally simulated interaction with the environment”. The elements of the world are modeled using AGM-guided Annealed Particle Filters and incorporated to an internally run robotics simulator. Thereafter, the *Cognitive Subtraction* (CS) algorithm detects differences between the data acquired from the sensors and the synthetic data obtained by *imagining* the output of the robot’s sensors in the simulator using the current world model. The result of this subtraction, along the grammar of the world representation—that can be used to describe the complex hierarchical and graph-based relationships that may be present in the environment—guide the subsequent actions of the robot, specifically those directed to acquire new models of the elements in the environment. In this vein, the grammar incorporates rules that the agents can use to detect possible non-modeled objects and introduce *unknown* symbols in the representation (with their corresponding attributes) so that AGM can drive the attention to them (if that helps satisfying the goals of the robot, of course).

The idea of linking senses and simulation is not new at all, dating back to Descartes [Descartes, 1641], where the philosopher considers the possibility that human senses could be manipulated by an *evil demon*. Since then, the issue has been addressed innumerable times and simulation has even been explicitly used to explain imagination processes, with extensive experimental work supporting modern versions of the hypothesis in human cognition [Hesslow, 2002]. In the field of robotics, and specially in biologically inspired robotics, there have also been different works using simulators that *imagine* what robots should perceive [Holland, 2003], [Ziemke et al., 2005], [H. and R., 2004]. Despite they do not necessarily have to explicitly simulate the output of the sensors, standard model-fitting methods such as particle filtering rely on the so called *observation model* function, that assigns a probability to the input provided by the sensors of the robot assuming a possible state (world model), sometimes by generating the measurements that sensors would acquire if such state was the actual one. Nevertheless, although it is not very common, some methods have actually used simulators in an explicit fashion. In the work presented in [Murtra et al., 2010] the authors generate synthetic laser measurements acquired using an ad-hoc simulator to evaluate samples in a particle filtering-based SLAM system. In [Nuske et al., 2009], a three-dimensional map of visual edges of the permanent structures of the environment is maintained to simulate how would edges be viewed from other points of view in order to support a visual SLAM particle filter. Another three-dimensional edge map of doors and walls in indoor environments is used for edge-based localization in [Kosaka and Kak, 1992].

Despite CS can be used with any type of sensor, the rest of this chapter describes how can this subtraction be performed using sensors providing three-dimensional point clouds. We also present here a small experiment in order to compare the algorithm used to find those parts of the input signal that can not be explained by the world model with other previous techniques. An experiment using the whole approach is presented in chapter 18 to model the room in which the robot is located, the tables and the mugs in the tables found.

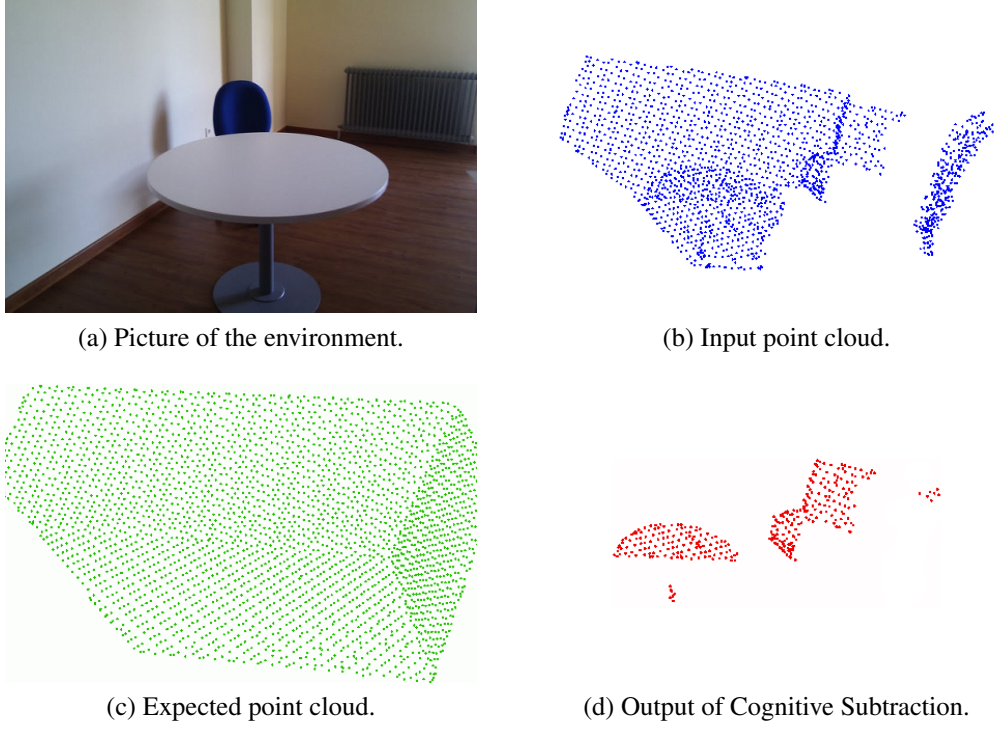


Figure 10.1: Using *Cognitive Subtraction* to find objects in a room once it has been modeled. Figure 10.1a shows a picture of the environment. Figures 10.1b, 10.1c, and 10.1d show the input point cloud, the expected point cloud and the detected outliers, respectively.

10.1 Cognitive Subtraction

Given a world model M_t and an input signal I_t coming from a sensor in the robot at time t , we define *cognitive subtraction* as:

$$CS \equiv I_t - Proj(M_t) \quad (10.1)$$

where $Proj$ is a function that generates the expected measurement data the robot would acquire if the environment was as described in M_t . The algorithm can be decomposed in three stages:

1. Generation of the **expected measurement data** given the world model (section 10.1.1).
2. **Adjustment** of the expected measurement data to the input sensor data in order to reduce the impact of small errors (*e.g.*, small modeling or synchronization errors). Covered in section 10.1.2.
3. **Detection of outliers** by segmenting the differences between the input and expected data (section 10.1.3).

Figure 10.1 depicts how cognitive subtraction would help finding objects in an indoor scene using a RGBD sensor once the room in which the robot is located has been modeled.

10.1.1 Generation of expected measurement data from the model

Many robotics simulators have the ability to simulate different sensors. However, the generation of simulated measurements according to a world model representation updated in real-time goes

beyond a regular robotics simulator. Those simulators that provide a software interface suitable for replicating the world model of a robot and updating it in real-time according to the updates of the model are not that common. Despite, theoretically, any robotics simulator providing such interface could be used for this task, none of the known simulators have been designed with this purpose in mind and provide a limited API. Therefore, we propose using the RCIS simulator [Gutiérrez et al., 2013] to update the simulated world in real-time because it provides a full-featured interface for this purpose. In fact, it was one of the objectives of RCIS since it was designed. The interface of the simulator should be well designed because the efficiency, readability and correctness of the systems using it will heavily depend on it. In particular, the management interface of RCIS includes:

- Add, remove, and modify geometric transformation nodes (*i.e.*, translations, rotations and full transformations). These do not provide any visual output themselves but are useful to manage the simulation because the pose of other nodes such as meshes of planes might depend on them.
- Add, remove, and activate robotic joints,
- Add, remove, scale and move three-dimensional meshes, planes and dynamically modify their textures.
- Remove any type of node, move them (from an absolute position or with respect to its parent or any other node) and consult the position of any node (again, with respect to any arbitrary reference frame).
- Add, consult and remove node attributes.

Besides updating the world according to the representation, using RCIS world files and its management interface we can simulate differential platforms, two-dimensional lasers, RGB and RGBD cameras, inertial sensors and GPS devices. Moreover, providing realistic physics simulation is not relevant for this purpose because the world model is supposed to be maintained by the robot and there is no need to simulate the changes using any other engine.

Among all the known general-purpose open source robotics simulators, Gazebo [Koenig and Howard, 2004] is probably the most used and mature one. It is capable of simulating medium-sized populations of robots and arbitrary objects in three-dimensional worlds. It also generates sensor feedback and physical interaction between objects. However, Gazebo is strongly focused on providing realistic physics simulations and its interface for maintaining world models in real time is limited. MORSE [Echeverria et al., 2011] is another robotics simulator with strong focus on realistic physics. Despite they are perfectly valid general-purpose simulators, USAR-Sim [Carpin et al., 2007], Simbad [Hugues and Bredeche, 2006], Darwin2k [Leger, 1999] or the iCub simulator [Tikhonoff et al., 2008] have the same limitations as Gazebo or Morse when used for Cognitive Subtraction: they are not designed to provide simulated sensors for world model representations maintained by third-party software.

10.1.2 Adjustment between input and expected data

Adjusting the simulated measurements to better fit the actual inputs acquired from the sensors of the robot is also necessary. Firstly, this adjustment might provide valuable information about

how to correct small perceptive errors in the model of the environment (*e.g.*, if the floor is closer than expected and the head of the robot is fixed it is likely that the floor has been inaccurately modeled). Secondly, it helps preventing false positives in the change detection step if the modeling algorithms are not tightly synchronized with this step, especially when the robot or the objects in the environment move at a considerable speed. Thus, before comparing the two inputs (point clouds in our case), the one corresponding to the simulated sensor is adapted to better fit the actual input.

In order to perform this step many different techniques can be used. They can be divided in those that are based on a Bayesian formulation and those which do not. Among Bayesian-based approaches Annealed Markov Chain Monte Carlo (AMCMC) [Hastings, 1970] and Annealed Particle Filters (APF) [Pitt and Shephard, 1999] can be highlighted. Iterative Closest Point (ICP) [Besl and McKay, 1992] is, by far, the most-used non-Bayesian approach. APF and AMCMC work by sampling the state space according to arbitrarily shaped probability distribution functions with decreasing variance with respect to the previous sample, while ICP iteratively looks for better transformations by performing an approximated gradient descent. Although ICP is very successful for point cloud registration, Bayesian approaches are better suited for the problem that this paper faces because they can sample the state space according to the uncertainty robots have about their environment. This is a very important question because in our case —detecting unexpected percepts— it is useful to shape the probability density functions according to the movement of the robot, which is the main source of uncertainty (*e.g.*, if the robot moves forwards we only want to sample the dimensions affected by such movement). Unfortunately, it is not possible to do such thing using ICP. Therefore, we propose using an annealed particle filter to look for the transformation that makes the two point clouds least divergent while taking into account the belief distribution about the state space. The remaining of this section describes the implementation of the annealed particle filter used to adapt the synthetic input generated using the simulator.

State space and transition model

The state space sampled by the particle filter used to find the best-fitting transformation is composed of a three-dimensional translation and rotation:

$$x_t = (t_x, t_y, t_z, r_x, r_y, r_z)^T \quad (10.2)$$

The transition model is a six-dimensional Gaussian with zero-mean and a variance depending on the movement of the robot (its angular and linear velocity).

$$P(x_t | x_{t-1}, u_t) \approx N(0, \Sigma(u_t)) \quad (10.3)$$

where $\Sigma(u_t)$ is the covariance matrix of the normal distribution used in equation 10.3:

$$\Sigma = \text{diag}(\sigma_{t_x}, \sigma_{t_y}, \sigma_{t_z}, \sigma_{r_x}, \sigma_{r_y}, \sigma_{r_z}) \quad (10.4)$$

The mean is zero because the simulation is continuously updated to reflect the latest model the robot has, so it is expected to find the optimum transformation near to zero translation and rotation. Even though the transformation will be very small, it is necessary to deal with it, especially with the rotation, because even small rotational errors can lead to considerable differences depending on the distance to the points rotated.

The values of $\Sigma(u_t)$ depend on the movement of the robot (u_t). In the experiments performed during this work, σ_{t_y} and σ_{r_z} have very low values because our robot can not move upwards or downwards nor rotate the roll angle of its camera. On the other hand, $(\sigma_{t_x}, \sigma_{t_z})$ and $(\sigma_{r_y}, \sigma_{r_x})$ are proportional to the amount of movement of the robot platform and its articulated head, respectively. At the expense of a relatively low increase of computational time, by iteratively decreasing the variance of the sampling process, annealed particle filters are able to maximize the observation while avoiding local maxima.

It must be highlighted that, unlike other methods such as ICP, particle filtering allow us to specify how uncertainty is distributed among the different variables of the state space. This way the sampling is focused on the most-likely areas of the state space.

Observation model

The observation model is designed to minimize the sum of the minimum distances from each of the points of the virtual point cloud to the input point cloud, see equation 10.5:

$$P(z_t|x_t) \propto 1 / \sum_{i=1}^{K_r} \operatorname{argmin}_{j \in \{1:K_v\}} d(R(i), M \cdot V(j)) \quad (10.5)$$

where K_r and K_v are the number of points in the input and virtual point clouds, and $R(i)$ and $V(j)$ are i -th and j -th points of the real and virtual point clouds, respectively. M is the transformation matrix that x_t encodes.

10.1.3 Detection of outliers

The last step toward Cognitive Subtraction is the detection of incoherences between the virtual and actual measurements. Methods for detecting unexpected inputs have an extremely wide range of applications, mainly because they can be used to enable robots to autonomously respond to unexpected events. Here, it is proposed to detect unexpected percepts with a different purpose, to guide the attention of the robot when modeling the world: instead of detecting changes in a scene at different times, the goal is to detect changes between what robots expect to perceive —according to the information available about the environment— and what they actually perceive.

The techniques used to detect changes in a scene at different times are generally referred to as *change* or *novelty detection* techniques. Examples of these can be found in [Drews et al., 2010], where Gaussian mixture models are used to detect changes in point clouds, or [Gaborski et al., 2004] where it is presented a feature-based novelty detection technique using images. The works presented in [M. Markou and S. Singh, 2003a] and [M. Markou and S. Singh, 2003b] provide thorough reviews of statistical and neural network-based general-purpose novelty detection techniques. The algorithms used for this purpose heavily depend on the type of data the sensors provide and the information that robots have and replicate in the simulator. Therefore, for the sake of the application presented in this paper it is assumed that the obtained measurements are point clouds.

Once expected measurement acquisition and measure adjustment have been solved, both point clouds (the virtual and the one from the actual sensor) should be very similar, only varying significantly in the presence of objects that have not been modeled yet and modeled objects that

have been moved or removed from the scenario. Assuming both point clouds are correctly aligned after the execution of the algorithm presented in section 10.1.2, this last step is the most straightforward one. Its result is an euclidean clusterization [Rusu et al., 2009] of a copy of the input cloud where all the points that have a close correspondence in the aligned virtual point cloud have been removed. Most novelty detection techniques in literature are unnecessarily sophisticated for Cognitive Subtraction, especially when working with point clouds, because—ignoring non-correctly modeled elements—the actual and synthetic point clouds are very similar. This means that simpler (and generally faster) techniques can be used. The algorithm used to detect unexpected clusters within the input point cloud is described as pseudo-code in algorithm 11. Moreover, additional ad-hoc improvements can be easily included before the euclidean clusterization, e.g. if we want robots to ignore the elements outside a specific region (e.g., the room in which the robot is located or the table in which we want to find objects), all the points outside the boundaries of the such region can be removed (see line 9 of algorithm 11). Each of the remaining clusters is then considered unexpected input.

Algorithm 11 Cognitive Subtraction for point clouds.

Require: I : The input cloud.

Require: V : The synthetic cloud.

Require: u_t : The measured linear and angular velocities of the camera.

Require: h : The threshold used to determine the acceptable distance range.

```

1:  $V_t := particleFilterOptimization(I, V, u_t)$ 
2:  $O := \emptyset$ 
3: forall  $p \in I$  do
4:    $p' := closestPointFromCloud(p, V_t)$ 
5:   if  $distance(p, p') > h$  then
6:      $O := O \cup p$ 
7:   end if
8: end forall
9:  $O_f = regionOfInterestFilter(O)$ 
10:  $C = euclideanClustering(O_f, h)$ 
11: return  $C$ 
```

10.1.4 Implementation considerations of cognitive subtraction for point clouds

There are two main efficiency issues to take into account when implementing Cognitive Subtraction with point clouds. The first one is the algorithm used to find the minimum distance from an arbitrary point to a point cloud C , used in line number 4 of algorithm 11. It can be computed creating a kD-tree with the reference cloud and use the tree to find the closest point p' of the cloud C to the point p . The minimum distance is then easily computed as the euclidean distance between p and p' .

The other issue with considerable impact in efficiency is the resolution of the point clouds. On one hand, it is desirable to reduce resolution in order to accelerate the point cloud alignment. This can be done temporarily, with a copy of the clouds, just to estimate the transformation that best aligns the clouds. In fact, even if it is done permanently, when robots see something that

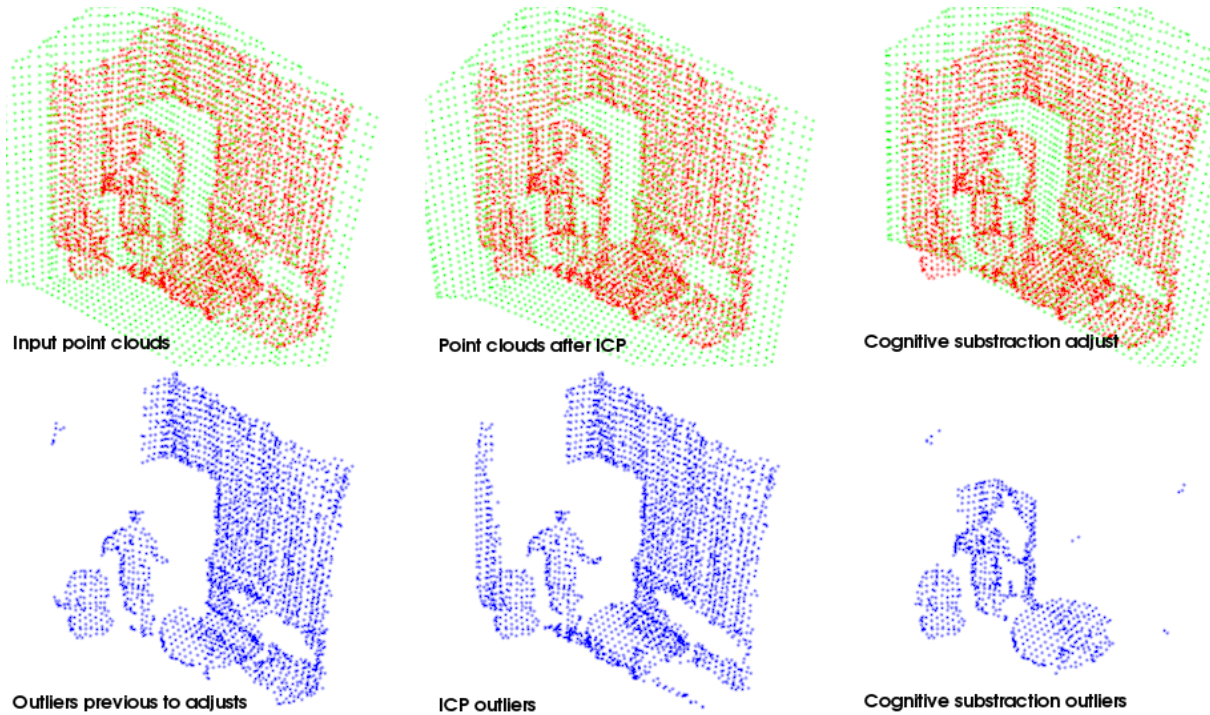


Figure 10.2: Cognitive Subtraction example 1: in the presence of considerable errors, the adjustment reduces the probability of false positives. In the room there is a table, a chair and a person holding a screen divider.

might be of interest, they can direct their gaze and approach to it in order to gather more precise information. On the other hand, the discrete nature of point clouds might have a bad influence on the accuracy of the alignment algorithms. If the resolution is too low, the algorithm employed to minimize the difference between the two point clouds might get stuck in local minimums easily. Instead of using high resolutions in both point clouds, the problem can be avoided by reducing resolution in only one of the point clouds. In order to decrease the necessary computational resources it is therefore recommended to reduce the resolution of the point cloud that is going to be transformed during the sampling process (*i.e.*, the one acquired using the simulator). This way, a kD-tree —or any other structure that may accelerate the search of the closest point of a point cloud to an arbitrary point— can be built for the static cloud, making the filter much more efficient.

Finally, it is worth mentioning that, to avoid false positives in the points corresponding to the borders of the depth map of the input sensor caused by the fitting, the simulated sensor should have a broader field of view.

10.2 Results and conclusions

In order to compare the results obtained using the proposed Annealed Particle Filter with Iterative Closest Points (see [Rusu et al., 2009, Chapter 5] for a brief introduction to ICP) we provide screenshots of the output of several execution of the algorithms using different input point clouds (see figures 10.2-10.5).

The results drawn from the experiments demonstrate the benefits usefulness of the idea

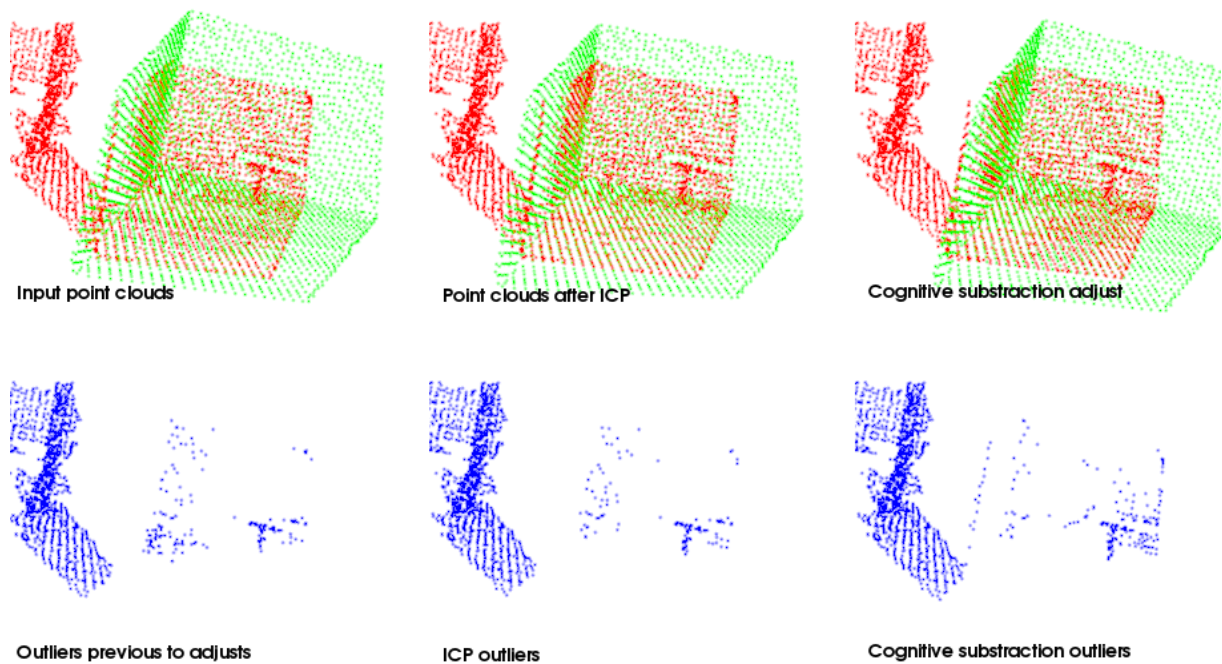


Figure 10.3: Cognitive Subtraction example 2: when the robot has only modeled a room and it is required to focus on the elements within it, the outliers located outside the room should be filtered. In the input cloud of the example the robot sees part of a table and an adjacent room.

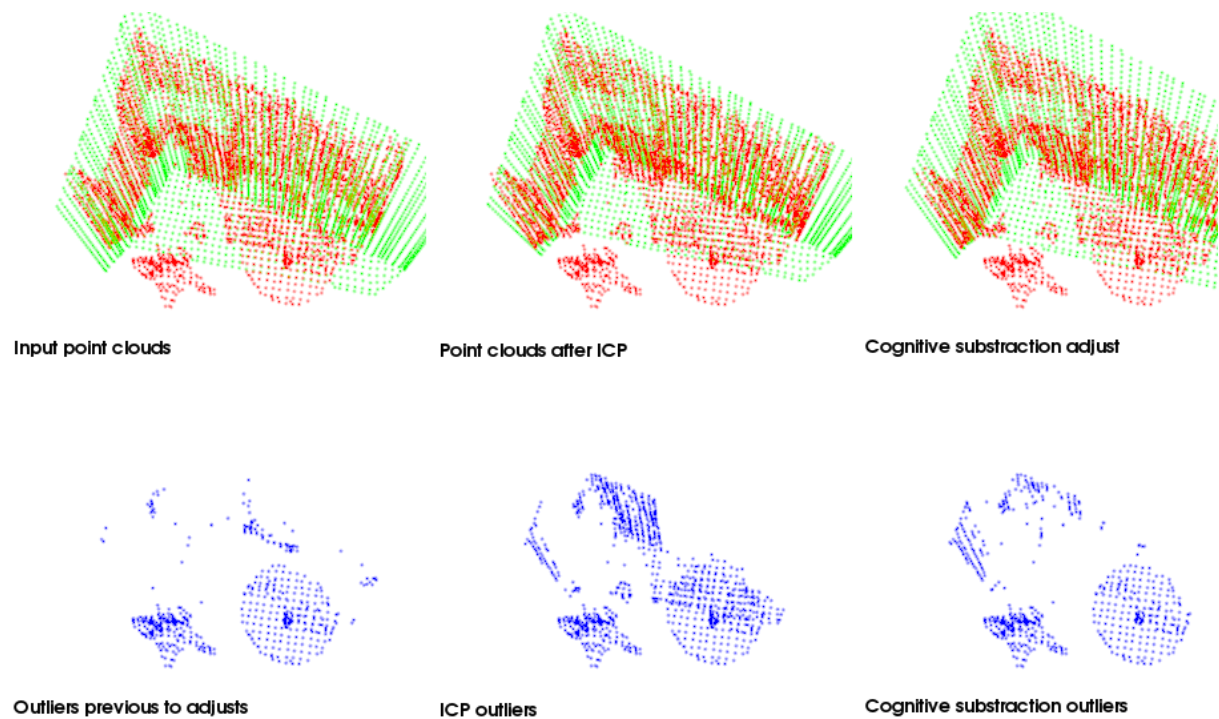


Figure 10.4: Cognitive Subtraction example 3: the annealed particle filter used to fit both point clouds should be run with a variance appropriate for the uncertainty of the model. Otherwise, it can adjust the clouds to much, resulting in false negatives.

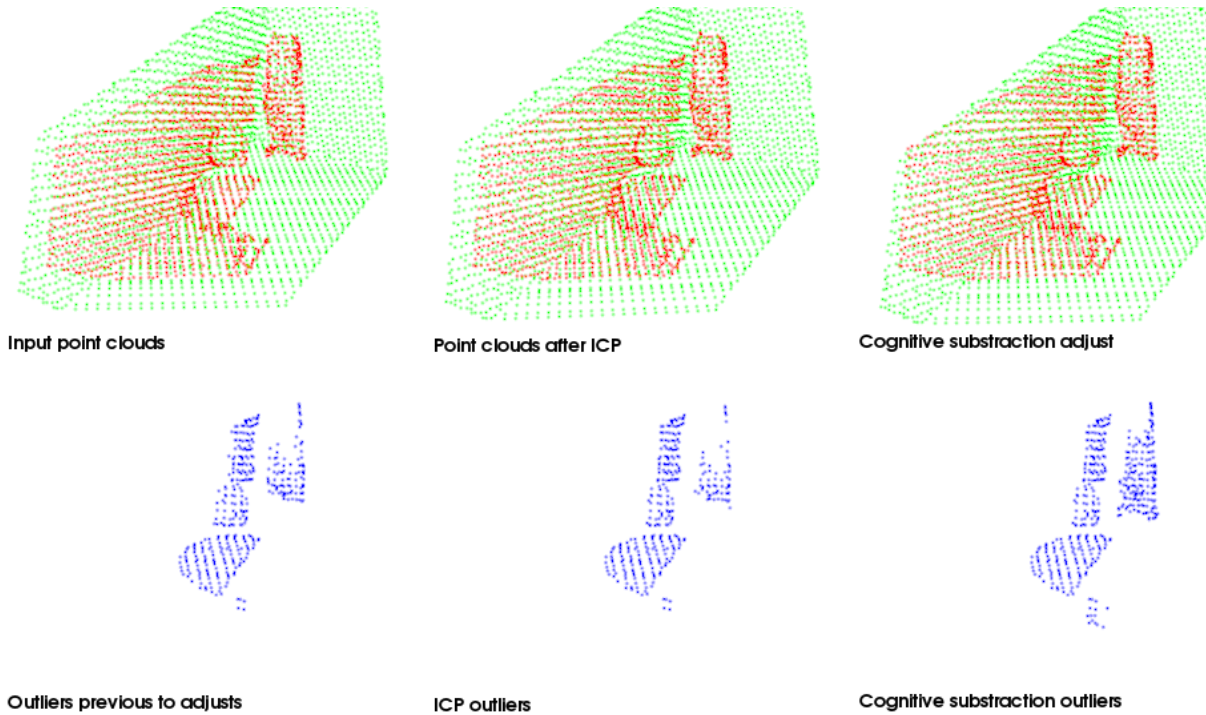


Figure 10.5: Cognitive Subtraction example 4: On the other hand, when the initial model is good, the adjustment may have little or no effect.

of Cognitive Subtraction, a novel technique designed to guide robot perception by identifying percepts that can not be explained by their current world model. Executing this technique iteratively can help robots to segment the elements of the environment and incrementally build world model representations directing the attention of the robots toward the unexpected percepts.

The work described in this chapter is used in the experiment of chapter 18 to detect the tables once the room in which the robot is located has been modeled. Additional demonstration videos can also be found in <http://ljmanso.com/thesis>. In order to enable everyone to test the concept, a free open source implementation of the algorithm has been made publicly available in the following GitHub repository: <https://github.com/ljmanso/pointCloudCS>.

Part III

Experiments

Chapter 11

Experiments map

Part III of the thesis is devoted towards explaining how the techniques covered in the text can be used to solve different useful problems related to the perception of structured environments, specifically of indoor environments.

Besides Active Grammar-based Modeling, which represents the major contribution of the thesis, there are many secondary but valuable contributions that will be presented along the experiments. The following list summarizes the experiments of the thesis with their contributions:

- **RoboComp DSLs:** The experiment provides empirical evidence of the impact of using of Domain Specific Languages (DSLs) in the context of component-oriented programming for robots. Specifically, it shows how using DSLs speeds-up different common tasks when programming robots using component-oriented programming and how it reduces the lines of code programmers have to write.
- **Deterministic indoor room modeling:** In this experiment it is described how a robot can create discriminative models of environments made of small rectangular rooms with obstacles using attention mechanisms. The main contribution of the experiment is a Hough-like method for detecting rectangles in two-dimensional grid maps.
- **A graph grammar for indoor environment mapping:** In this experiment it is proposed a first approximation to AGM. The goal of the robot in the experiment is to model a similar environment to the one in the previous experiment. The difference in this case is the use of grammar rules (hard-coded in C++) for action selection and introducing context-aware restrictions.
- **Yaw estimation:** Estimating the yaw of the robot in indoor environments is an important issue. This experiment presents an annealed particle filter for this purpose. The main contribution of the experiment is the observation model for the filter.
- **Distance estimation:** Estimating distances to planes is another important problem to solve, since it is necessary to model the height of the robot (estimating the distance from its camera to the floor) and the distances to the walls. This experiment presents another annealed particle filter for this purpose. As well as in the previous experiment, the main contribution is the observation model for the filter.

- **Room modeling:** This experiment, the first using AGM, shows how it can be used in combination with the filters of the two previous experiments to model a room. Additionally, it describes a software implementation of AGM and the results obtained using the architecture.
- **Cognitive subtraction:** Cognitive subtraction consists on detecting those parts of the environment which have not been appropriately modeled or not modeled at all. In the approach presented in the experiment, the information available about the environment is introduced in a robotic simulator and then the sensory input provided by the sensors of the robot is compared with the one provided by simulated ones. It can be used to detect which world elements can not be explained by the model the robot has and vice versa, which elements of the model are not supported by sensor data. In this experiment we use a combination of a special-purpose robotics simulator and an annealed particle filter.
- **A graph grammar for complex indoor environments:** This is the final experiment. In it, the robot uses many of the techniques covered in the thesis to actively perceive a room and the elements it contains (tables, obstacles, mugs).

Chapter 12

RoboComp DSLs

12.1 Goal

The goal of this experiment is to provide empirical evidence supporting the claims made regarding Domain-Specific Languages in section 3.4 and, more specifically, in section 5. Therefore, this section presents a real case study in which it is presented several common tasks to do when using component-oriented programming and how developers perform with and without these tools.

These tasks are situated in the following context: starting with a speech synthesis component, the goal is to communicate it with another component that controls a robotic mouth. The final configuration should be able to synthesize text and to move the mouth synchronously. Figure 12.4 shows all the components involved as seen from the RCManager deployment application.

As introduced in section 4, before making RoboComp DSL-based, to complete these changes it was required to make several changes in the source code manually as well as a good background on low-level component-oriented programming. In this section we will show how the use of DSLs reduces the time to introduce these changes, avoids undesirable errors in the code and improves user experience. The steps required to perform all the modifications are:

- Modify the Component Description file (CDSL) to include a new proxy to MouthComp.
- Include a new parameter in the Parameters Definition file of SpeechComp to specify how the connection to MouthComp will be done.
- Add a new entry in the Deployment Definition file to include the MouthComp component.
- Optionally, add a new method in the interface of SpeechComp to activate/deactivate in run-time the mouth synchronization.

12.2 CDSL

The first step to introduce an additional proxy to the Speech component is to change its CDSL file (i.e., the file describing the generic properties of the component). This can be done using the RoboComp DSL Editor (see figure 12.1). After re-generating the generic code, the specific

classes will automatically have access to the new proxy (no changes in the specific classes are needed for it).

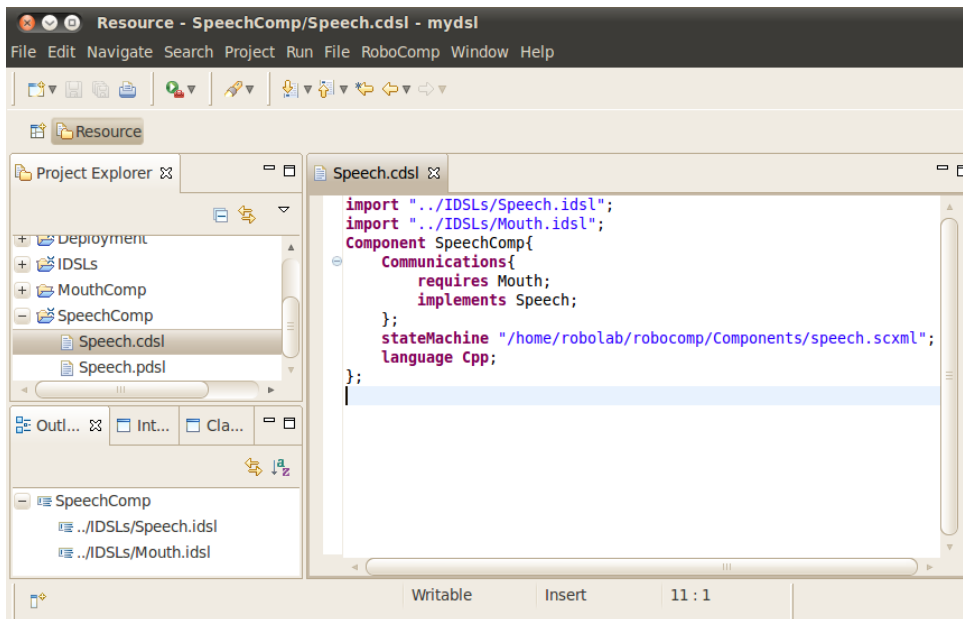


Figure 12.1: Screenshot of the RoboComp DSL Editor while modifying the CDSL model in the case study.

12.3 PDSL

After modifying the CDSL file, and only if the connection to the Mouth component is going to be optional, the PDSL file must be updated to include the corresponding variable. In this case, we created a new boolean parameter named `mouthSynchronization` as shown in figure 12.2.

12.4 DDSL

Once the code has been re-generated and the SpeechComp configuration parameters have been configured, the component is ready to work. However, since it depends on the execution of MouthComp, this component must be previously executed. Deployment can be done manually, but it is a tedious and time-consuming task when the component network is composed of more than a few components. The RoboComp DSL Editor can be used to specify different deployment scenarios. In this case the network is composed by only three components: SpeechComp, MouthComp and JointMotorComp (which is a dependence of the MouthComp component). Figures 12.3 and 12.4 provide screenshots of the RoboComp DSL Editor while modifying the previous DDSL file and the RCManager tool, respectively.

RCManager, the component manager of RoboComp (described in section 6.1), takes the DDSL file as input and manages the execution of the components and the dependencies among them. RCManager reads the deployment file and generates a visual graph where components are shown as nodes and component dependencies as edges. Moreover, the deployment file can

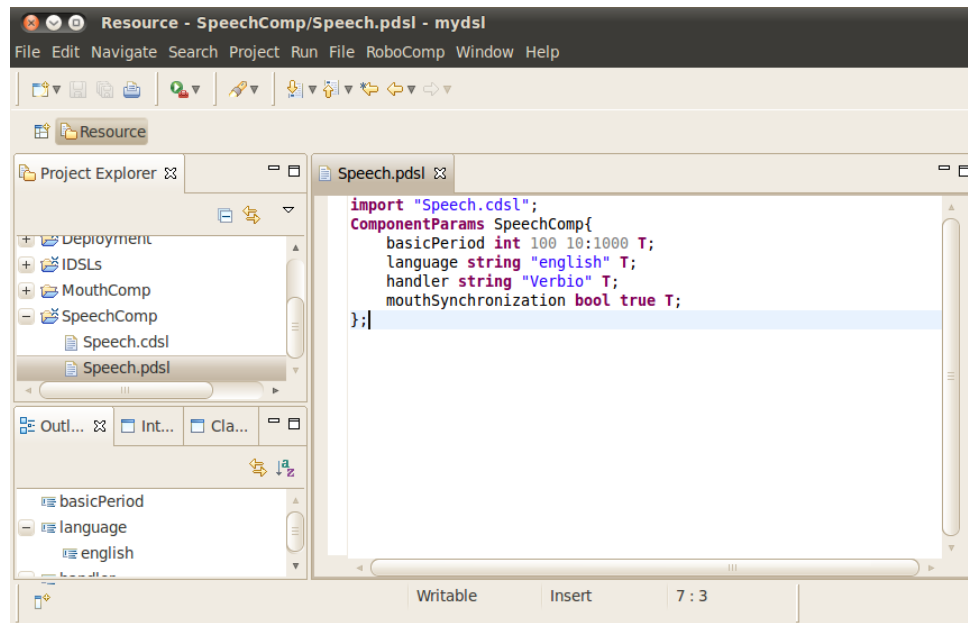


Figure 12.2: Screenshot of the RoboComp DSL Editor while modifying the PDSL model in the case study.

be generated or modified graphically using this tool. When a new host is added, RCManager checks for the availability of any known component in that machine and shows them in a list. Deployment configuration can also be created visually by dragging the components shown in this list on the graph view tab (see figure 12.4). Similarly, dependences may be assigned by dragging depending nodes over the nodes corresponding to their dependences.

In order to run a component, the user only has to double-click the node of the component to be run, or click in the “run” button of the pop-up window that appears when right-clicking nodes.

12.5 IDSL

Even though it may not be actually necessary, the developer might want to include a new method in the SpeechComp interface in order to activate/deactivate the synchronization with the robotic mouth on-line. In this case, there are three steps to take: **a)** use the RoboComp DSL Editor to include the line corresponding to the new method (see figure 12.5); **b)** re-generate the IDSL file in order to obtain the new IDL implementation; **c)** re-generate the SpeechComp CDSL to include the new function in the generic code. It is very important to note that this is the only step in which users need to modify the specific code manually. It is worth noting that it was chosen to avoid modifying specific code automatically in order to avoid damaging code written by the developer (as opposed to generic code, which is supposed to never be modified by developers, specific code is).

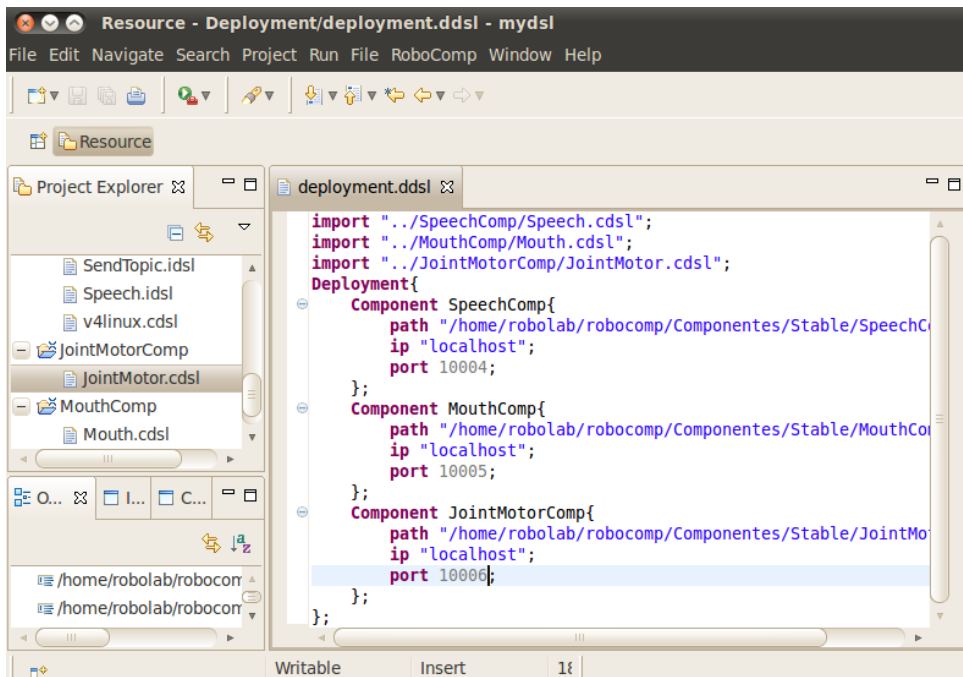


Figure 12.3: Screenshot of the RoboComp DSL Editor while modifying the DDSL model in the case study.

12.6 Conclusions

The experiment was conducted in order to provide empirical evidences supporting the claims made regarding Domain-Specific Languages. Twelve roboticists—who were already familiar with Ice and the RoboComp framework—were asked to perform five of the most common repetitive tasks they have to face when developing and managing robotics components. In particular, they were asked to make the following changes and operations with an existing robotics component:

- **a)** to create a new proxy for a given interface,
- **b)** to make the component provide a new additional interface,
- **c)** to include a new method in the previous interface,
- **d)** to include a new library and a new class in the component project and
- **e)** to deploy a small component network.

In order to be able to evaluate the benefits of the proposed approach, the experiments were performed twice: first, using the DSL technology and then, without it. For all the experiments, two variables were measured: the time spent performing the task and the lines of code written (with the exception of the last task, deploying a component network, where instead of the lines of code written it were measured the number of commands executed).

The data obtained from the time spent and the lines of code written is displayed in figures 12.6 and 12.7, respectively. Measurement data is represented as boxplots containing all measurements ranging between the first and third quartiles. The location of the median of the

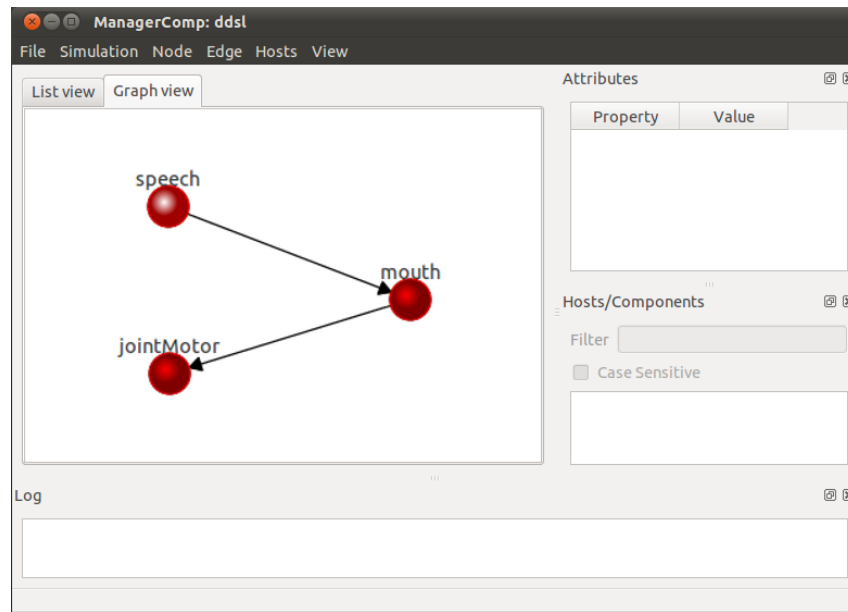


Figure 12.4: Screenshot of the RCManger tool. The three components involved in the component network are shown. Once the user requests to run the Speech component, its dependencies are automatically satisfied and the component is run.

measurements is indicated by a red line crossing the rectangle vertically. Measurements outside the box are considered outliers and are drawn using green diamonds.

Figure 12.6 shows the results regarding the time spent in the experiments. It is worth mentioning that the only time taken into account was the one in which the subject was typing code, not thinking. Since users need less time to think when using DSLs (*i.e.*, there is no need to think which code pieces should they change) this plays against the use of DSLs. In spite of this, it can be seen how using the DSL approach the subjects of the experiment achieved shorter times for all of the experiments performed.

Figure 12.7 shows the results regarding the lines of code written while performing the experiments. As happened with time, the figure shows that, using the DSL approach, fewer lines of code were written for all of the experiments performed. This is not a surprise, since small changes in the DSLs might involve many changes in the generated code. The only experiment in which no considerable improvements were achieved—a few seconds but almost no improvements in lines of code—was the experiment in which subjects had to include new libraries in the project. This is because RoboComp was already making use of CMake features for the operations involved in the experiment, so the initial number of lines to modify was already low.

This experiment has presented four DSLs based on the MDA approach used to improve the lifecycle of robotics components: **a)** for specifying general component properties (CDSL); **b)** for defining component interfaces (IDSL); **c)** for the deployment of components (DDSL); and **d)** to define and initialize component parameters (PDSL).

CDSL allows to create and modify components, potentially even using different programming languages. It reduces the workload, so developers can center their effort in the implementation of the behaviour of the components they are developing instead of middleware-related code.

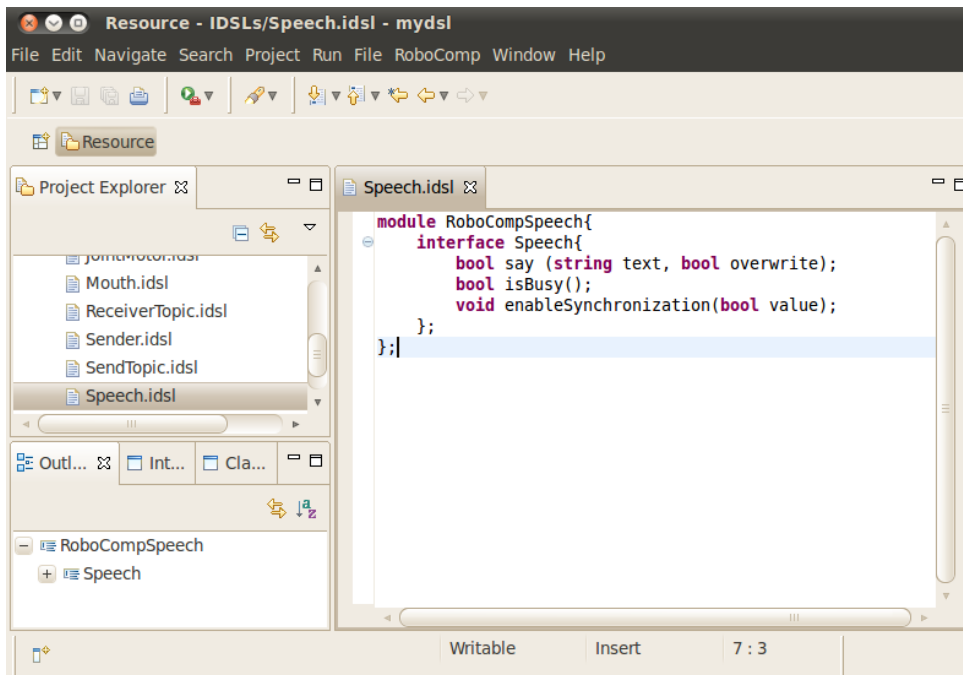


Figure 12.5: Screenshot of the RoboComp DSL Editor while modifying the IDSL model in the case study.

IDSL is a first step to make RoboComp independent not only of the architecture and the programming language, but also of the middleware. It provides the most commonly-used features of other interfaces definition languages such as ZeroC Slice or DDS. Additionally, an implementation of the communication patterns proposed in [ScGutierrez et al., 2010] in IDSL is underway (see [Gutiérrez et al., 2013]).

DDSL enables the automatic deployment and monitoring of the state of networks of components. It is used to describe networks of components and their dependencies, as well as where and how should they be executed. This allows RCManager, the component manager of RoboComp, to deploy networks of components with just a mouse click.

Finally, PDSL provides a means to store the configuration parameters of the components in a structured and less error-prone way (checking the types and the possible values of the parameters).

These DSLs have been used on the RoboComp framework to improve its flexibility, scalability and maintenance; making it possible to create and manage components in a higher level of abstraction. These DSLs have been developed as textual models in order to reduce the development time, although they can be used in conjunction with visual models.

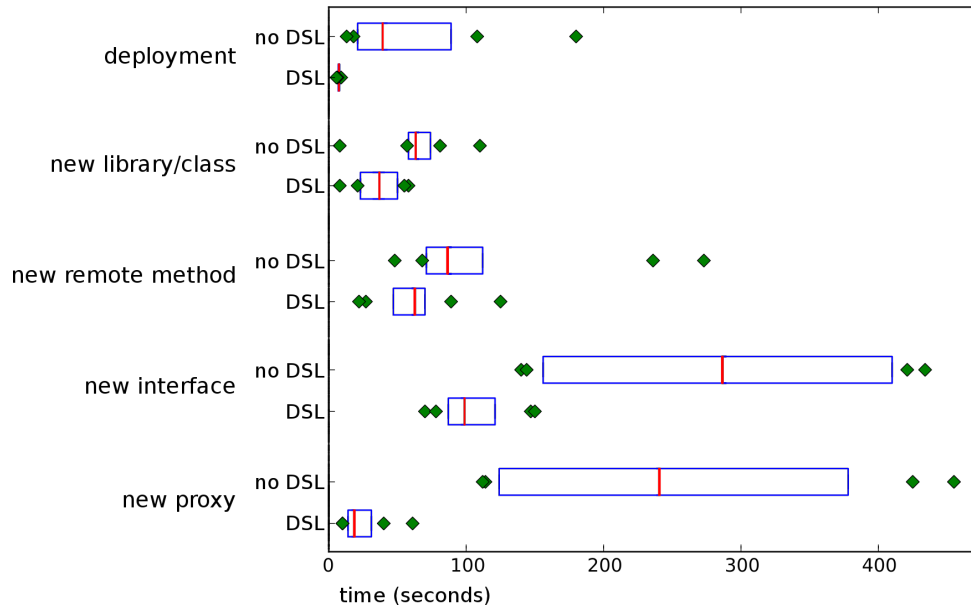


Figure 12.6: Graph showing the corresponding boxplots for the times associated with the five experiments. For each experiment it is shown the time spent using the DSL technology (DSL) and without it (no DSL).

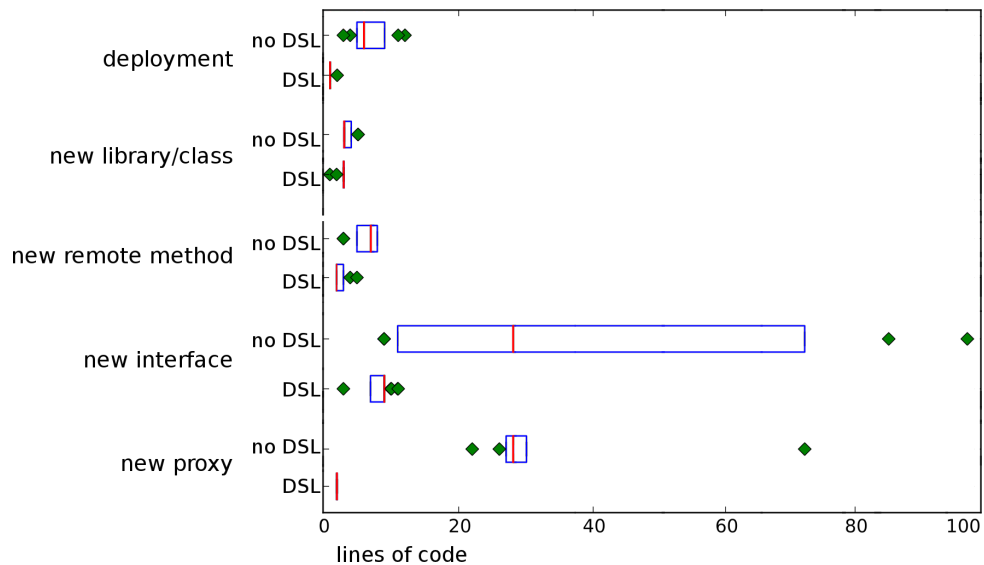


Figure 12.7: Graph showing the corresponding boxplots for the lines of code associated with the five experiments. For each experiment it is shown the time spent using the DSL technology (DSL) and without it (no DSL).

Chapter 13

Deterministic indoor room modeling

13.1 Goal

This experiment presents the first attempt towards creating a useful hybrid mapping system. The goal was to map indoor environments composed of several rooms connected through doors. Rooms were assumed to be approximately rectangular and could contain obstacles on the floor (see figure 13.1.

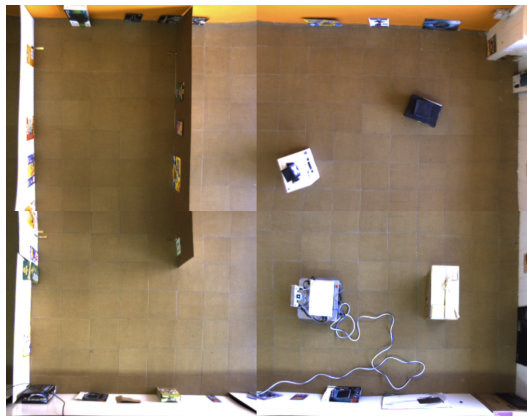


Figure 13.1: Overhead view of the real scene of the experiment of figure 13.4.

13.2 Solution

The solution proposed is to store the perceived points during room exploration in a three-dimensional probabilistic occupancy grid, where the probability associated to each cell is the probability of that area being occupied. Once the robot leaves a room, the current grid is associated to its room symbol and a new grid is created along the new room symbol.

The use of stereo vision for this experiment made necessary to address several sensor-specific issues:

- the certainty of the position of the detected points decreases as the distance increases,
- there could be errors in the parametrization of the stereo pair.

In addition, as usual, certainty increases as a region is perceived over time in the same position, so stable regions produce higher occupancy values than unstable ones. Cells with high occupancy certainty are used for detecting a room model fitting the set of perceived regions to a rectangle. Once the model of the current room is considered stable, its corresponding symbol in the topological map is filled with its size and orientation. The most important contribution of the research conducted for the experiment is a variation of the Hough transform used to detect rectangles.

In a similar way, when a door is detected a new symbol is created for it and linked to the current room. When a detected room is crossed the new room symbol is also linked to the door. Doors are detected as holes in the rectangular model of the room —parts of the walls in which the occupancy grid is low— and therefore, can only be detected once the room model is constructed. The size of the doors is lower bounded so that small textureless parts of the environment are not mistaken with doors.

13.3 Room modeling

Since rooms are assumed to be rectangular and its walls perpendicular to the floor, the problem of modeling a room from a set of regions can be treated as a rectangle detection problem—ignoring the height of the points. Several rectangle detection techniques can be found in the literature [Lagunovsky and Ablameyko, 1999, Lin and Nevatia, 1998, Tao et al., 2002]. Most of them are based on a search in the 2D point space (for instance, a search in the edge representation of an image) using line primitives. These methods are computationally expensive and can be very sensitive to noisy data. In order to solve the modeling problem in an efficient way, we propose a new rectangle detection technique based on a search in the parameter space using a variation of the Hough Transform [Rosenfeld, 1969, Duda and Hart, 1972].

For line detection, several variations of the Hough Transform have been proposed [Matas et al., 2000, Palmer et al., 1994]. The idea of extending the Hough Transform for rectangle detection is not new. [Zhu et al., 2003] proposes a *Rectangular Hough Transform* used to detect the center and orientation of a rectangle with known dimensions. [Jung and Schramm, 2004] proposes a *Windowed Hough Transform* that consists of searching rectangle patterns in the Hough space of every window of suitable dimensions of an image.

Our approach for rectangle detection uses a 3D version of the Hough Transform that facilitates the detection of segments instead of lines. This allows considering only those points that belong to the contour of a rectangle in the detection process. The Hough space is parameterized by (θ, d, p) , being θ and d the parameters of the line representation ($d = x \cos(\theta) + y \sin(\theta)$) and $|p|$ the length of a segment in the line. For computing p it is assumed that one of the extreme points of its associated segment is initially fixed and situated at a distance of 0 to the perpendicular line passing through the origin. Under this assumption, being (x, y) the other extreme point of the segment, its *signed* length p can be computed as:

$$p = x \cos(\theta + \pi/2) + y \sin(\theta + \pi/2) \quad (13.1)$$

Using this representation, any point (x, y) contributes to those points (θ, d, p) in the Hough space that verifies:

$$d = x \cos(\theta) + y \sin(\theta) \quad (13.2)$$

and

$$p \geq x \cos(\theta + \pi/2) + y \sin(\theta + \pi/2) \quad (13.3)$$

Equation 13.2 represents every line intersecting the point as in the original Hough Transform. The additional condition expressed by equation 13.3 limits the point contribution to those line segments containing the point. This allows computing the total number of points included in a given segment. For instance, given a segment with extreme points $V_i = (x_i, y_i)$ and $V_j = (x_j, y_j)$ and being H the 3D Hough space, the number of points that belongs to the segment, which is denoted as $H_{i \leftrightarrow j}$, can be computed as:

$$H_{i \leftrightarrow j} = |H(\theta_{i \leftrightarrow j}, d_{i \leftrightarrow j}, p_i) - H(\theta_{i \leftrightarrow j}, d_{i \leftrightarrow j}, p_j)| \quad (13.4)$$

where $\theta_{i \leftrightarrow j}$ and $d_{i \leftrightarrow j}$ are the parameters of the common line to both points and p_i and p_j the *signed* lengths of the two segments with non-fixed extreme points V_i and V_j , respectively, according to equation 13.1.

Since a rectangle is composed of four segments, the 3D Hough space parameterized by (θ, d, p) allows computing the total number of points included in the contour of the rectangle. Thus, considering a rectangle expressed by its four vertices $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$, $V_3 = (x_3, y_3)$ and $V_4 = (x_4, y_4)$ (see figure 13.2), the number of points of its contour, denoted as H_r , can be computed as:

$$H_r = H_{1 \leftrightarrow 2} + H_{2 \leftrightarrow 3} + H_{3 \leftrightarrow 4} + H_{4 \leftrightarrow 1} \quad (13.5)$$

Considering the restrictions about the segments of the rectangle and using the equation 13.4, each $H_{i \leftrightarrow j}$ of the expression 13.5 can be rewritten as follows:

$$H_{1 \leftrightarrow 2} = |H(\alpha, d_{1 \leftrightarrow 2}, d_{4 \leftrightarrow 1}) - H(\alpha, d_{1 \leftrightarrow 2}, d_{2 \leftrightarrow 3})| \quad (13.6)$$

$$H_{2 \leftrightarrow 3} = |H(\alpha + \pi/2, d_{2 \leftrightarrow 3}, d_{1 \leftrightarrow 2}) - H(\alpha + \pi/2, d_{2 \leftrightarrow 3}, d_{3 \leftrightarrow 4})| \quad (13.7)$$

$$H_{3 \leftrightarrow 4} = |H(\alpha, d_{3 \leftrightarrow 4}, d_{2 \leftrightarrow 3}) - H(\alpha, d_{3 \leftrightarrow 4}, d_{4 \leftrightarrow 1})| \quad (13.8)$$

$$H_{4 \leftrightarrow 1} = |H(\alpha + \pi/2, d_{4 \leftrightarrow 1}, d_{3 \leftrightarrow 4}) - H(\alpha + \pi/2, d_{4 \leftrightarrow 1}, d_{1 \leftrightarrow 2})| \quad (13.9)$$

being α the orientation of the rectangle as expressed in figure 13.2 and $d_{i \leftrightarrow j}$ the normal distance of the origin to the straight line defined by the points V_i and V_j .

Since H_r expresses the number of points in a rectangle r defined by $(\alpha, d_{1 \leftrightarrow 2}, d_{2 \leftrightarrow 3}, d_{3 \leftrightarrow 4}, d_{4 \leftrightarrow 1})$, the problem of obtaining the best rectangle given a set of points can be solved by finding the combination of $(\alpha, d_{1 \leftrightarrow 2}, d_{2 \leftrightarrow 3}, d_{3 \leftrightarrow 4}, d_{4 \leftrightarrow 1})$ that maximizes H_r . This parametrization of the rectangle can be transformed into a more practical representation defined by the five-tuple (α, x_c, y_c, w, h) , being (x_c, y_c) the central point of the rectangle and w and h its dimensions. This transformation can be achieved using the following expressions:

$$x_c = \frac{d_{1 \leftrightarrow 2} + d_{3 \leftrightarrow 4}}{2} \cos(\alpha) - \frac{d_{2 \leftrightarrow 3} + d_{4 \leftrightarrow 1}}{2} \sin(\alpha) \quad (13.10)$$

$$y_c = \frac{d_{1 \leftrightarrow 2} + d_{3 \leftrightarrow 4}}{2} \sin(\alpha) + \frac{d_{2 \leftrightarrow 3} + d_{4 \leftrightarrow 1}}{2} \cos(\alpha) \quad (13.11)$$

$$w = d_{2 \leftrightarrow 3} - d_{4 \leftrightarrow 1} \quad (13.12)$$

$$h = d_{3 \leftrightarrow 4} - d_{1 \leftrightarrow 2} \quad (13.13)$$

In order to compute H_r , the parameter space H is discretized assuming the rank $[-\pi/2, \pi/2]$ for θ and $[d_{min}, d_{max}]$ for d and p , being d_{min} and d_{max} the minimum and maximum distance, respectively, between a line and the origin. The sampling step of each parameter is chosen according to the required accuracy. Figure 13.2 shows an example of rectangle representation in the discretized parameter space. Each pair of parallel segments of the rectangle is represented in the corresponding orientation plane of the discrete Hough space: $H(\alpha_d)$ for one pair of segments and $H((\alpha + \pi/2)_d)$ for the other one, being α_d and $(\alpha + \pi/2)_d$ the discrete values associated to α (the rectangle orientation) and $(\alpha + \pi/2)$, respectively. For each orientation plane, it is represented how many points contribute to each cell (d_d, p_d) , i.e. how many points belong to every segment of the corresponding orientation. A high histogram contribution is represented in the figure with a dark gray level, while a low contribution is depicted with an almost white color. As it can be observed, the maximum contributions are found in parallel segments with displacements of w_d and h_d , which are the discrete values associated to the rectangle dimensions.

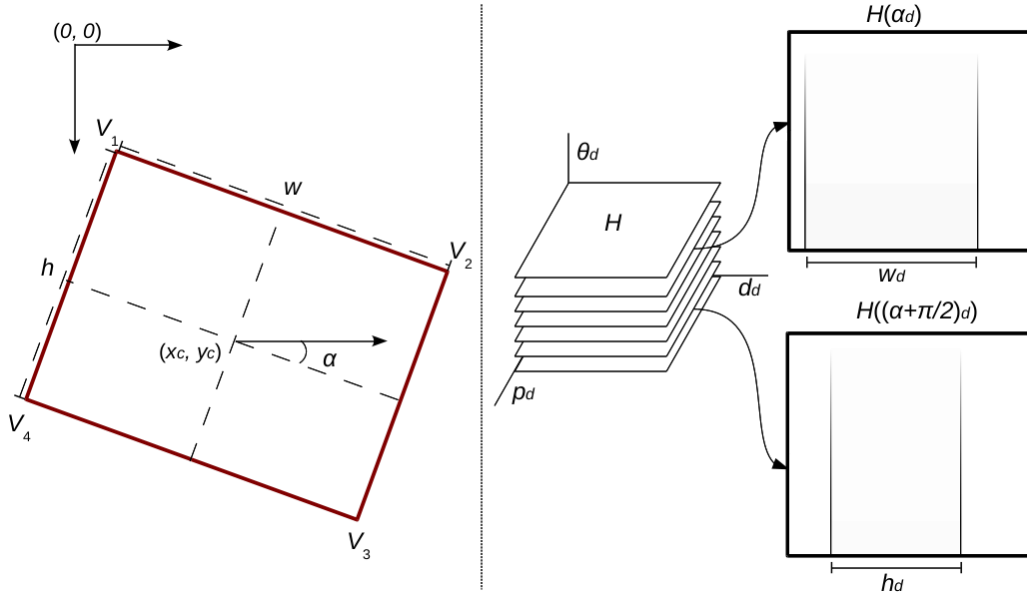


Figure 13.2: Rectangle detection using the proposed 3D variation of the Hough Transform (see the text for further explanation)

This rectangle detection technique is used to obtain a room model that fits the points stored in the 3D occupancy grid. Walls are considered to have a maximum height and, therefore, only points situated at a certain rank of height in the grid are used for detecting the model. Assuming that this rank is in the interval $[0, Z_{wall}]$ and being G the 3D occupancy grid and τ the minimum occupancy value to considered a non empty region of the environment, the proposed method for room modeling can be summarized in the following steps:

1. Initialize all the cells of the discrete Hough space H to 0.
2. For each cell, $G(x_d, y_d, z_d)$, such that $G(x_d, y_d, z_d).occupancy > \tau$ and $z_d \in [0, Z_{wall}]$:

Compute the real coordinates (x, y) associated to the cell indexes (x_d, y_d) .

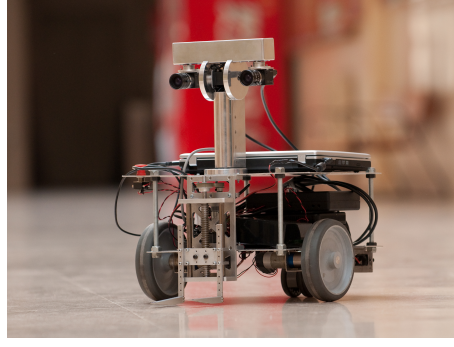


Figure 13.3: The RobEx platform.

For $\theta_d = \theta_{dMin} \dots \theta_{dMax}$:

- (a) Compute the real value θ associated to θ_d .
 - (b) Compute $d = x \cos(\theta) + y \sin(\theta)$.
 - (c) Compute the discrete value d_d associated to d .
 - (d) Compute $p = x \cos(\theta + \pi/2) + y \sin(\theta + \pi/2)$.
 - (e) Compute the discrete value p_d associated to p .
 - (f) For $p'_d = p_d \dots d_{dMax}$: increment $H(\theta_d, d_d, p'_d)$ by 1.
3. Compute $\arg \max_{\alpha, d_{1 \leftrightarrow 2}, d_{2 \leftrightarrow 3}, d_{3 \leftrightarrow 4}, d_{4 \leftrightarrow 1}} H_r(\alpha, d_{1 \leftrightarrow 2}, d_{2 \leftrightarrow 3}, d_{3 \leftrightarrow 4}, d_{4 \leftrightarrow 1})$.
 4. Obtain the rectangle $r = (\alpha, x_c, y_c, w, h)$ using equations 13.10, 13.11, 13.12 and 13.13.

13.4 Results

We carried out different experiments in real indoor environments. They were designed to demonstrate the modeling ability of a mobile robot equipped with a stereo vision head. In particular, we used RobEx (figure 13.3, see [Mateos et al., 2010]), an open-hardware robotic platform. For the experiments, RobEx was equipped with a 4-dof stereo vision head providing: a neck-movement followed by a common tilt and two independent camera pan movements.

In the first experiment, we tested the ability of the robot to model a room with several objects on the floor. Figure 13.4 shows the result of this experiment for the modeling of the room at the right side of figure 13.1. Figure 13.4a shows the set of perceived regions after an autonomous exploration of the room. These regions are associated to cells of the occupancy grid with a high certainty degree. From this set of points, the models of the room and the door are detected (figure 13.4b). The metric representation of the resulting models is shown in figure 13.4c. Using this room model, the occupancy grid is updated to cover only the local space inside the room. In addition, the positions of the points near to walls are adjusted according to the detected model. Figure 13.4d shows the result of this adjustment.

In the second experiment, the robot modeled an environment composed by two rooms communicated by a door (figure 13.7). Figures 13.5 and 13.6 show the evolution of the modeling process during the environment exploration. Specifically, for the modeling of the first room, the results of the robot behavior can be observed in figures 13.5a to 13.5d: (a) an initial model

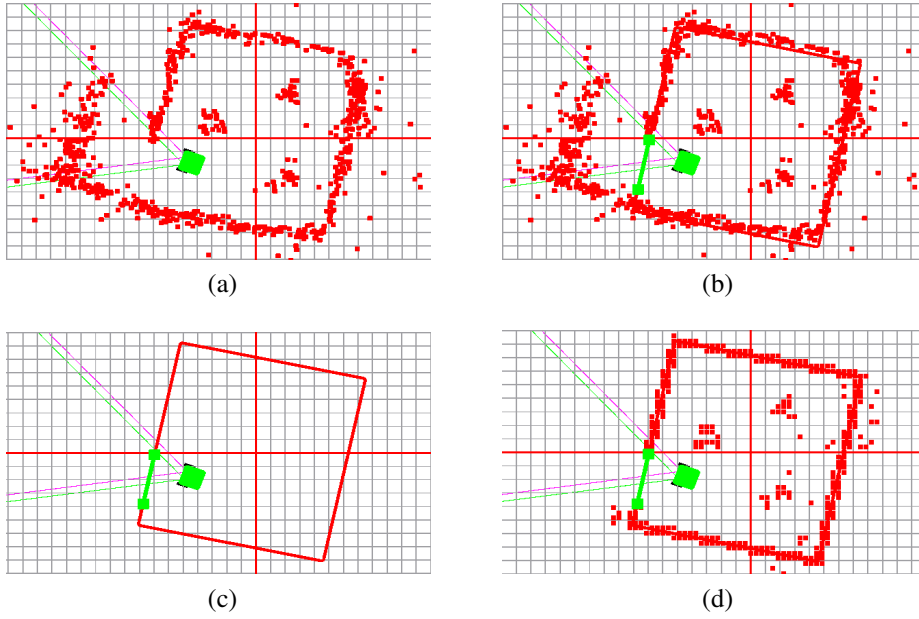


Figure 13.4: Room modeling process: (a) 2D view of the perceived regions during the exploration; (b) room and door models detection from the set of points in (a); (c) metric representation of the room; (d) state of the occupancy grid after fixating the room model in (c). The figure corresponds to the environment shown in 13.1.

is created fixating attention on frontal visual regions while the robot base turns around; (b) the robot verifies the room model by keeping attention on high uncertainty zones while approaching them. This allows correcting the model parameters as well as discarding the false door on the right of figure (a); (c) the door is verified and its parameters are adjusted by fixating attention on regions situated outside of the room; (d) the robot gets out of the room to start exploring the second room. Once the robot gets into the new room, the modeling process is repeated for detecting a new model as it is shown in figures 13.6a to 13.6c: (a) initial model creation; (b) room model verifying stage; (c) door verifying stage. Finally (figure 13.6d) the deviation between the two room models is corrected applying the common door restriction.

The last experiment tested the ability of the robot to model a more complex configuration of the environment. The environment of the experiment was composed by three intercommunicated rooms with different dimensions (figure 13.9). Figure 13.8 shows the modeling results during the exploration of each room. After creating the final model of every room, the deviation of the new model is corrected according to the door connecting the room to an existing model. Finally, it is obtained an environment representation that corresponds to a great extent with the real scene. The error in the dimensions of the first room (see figure 13.8e) is due to the discretization of the Hough space. This kind of errors are considered in the representation of the model as part of its uncertainty. It is important to note that this uncertainty is local to the model. Thus, as it is shown in figure 13.8, there is no error propagation in the creation of remaining models composing the final representation of the environment.

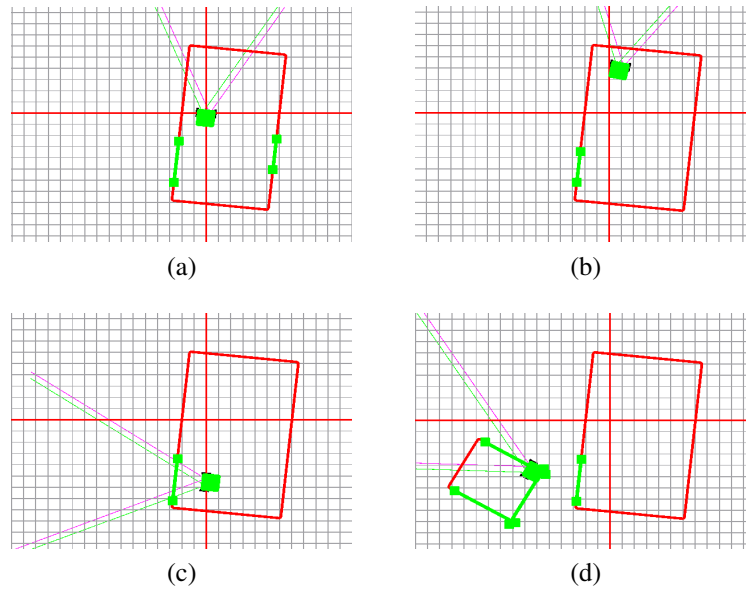


Figure 13.5: Modeling of the first room during the exploration of the environment of figure 13.7.

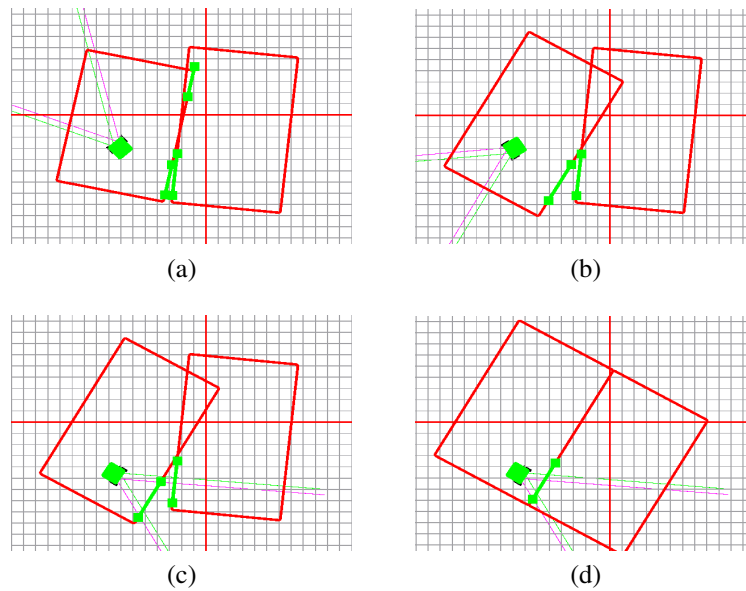


Figure 13.6: Modeling of the second room during the exploration of the environment of figure 13.7.

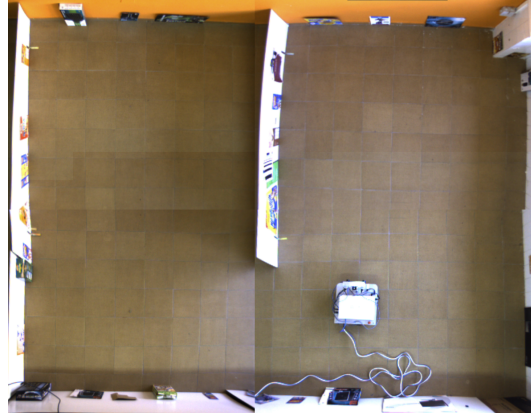


Figure 13.7: Overhead view of the environment of the experiment of figures 13.5 and 13.6.

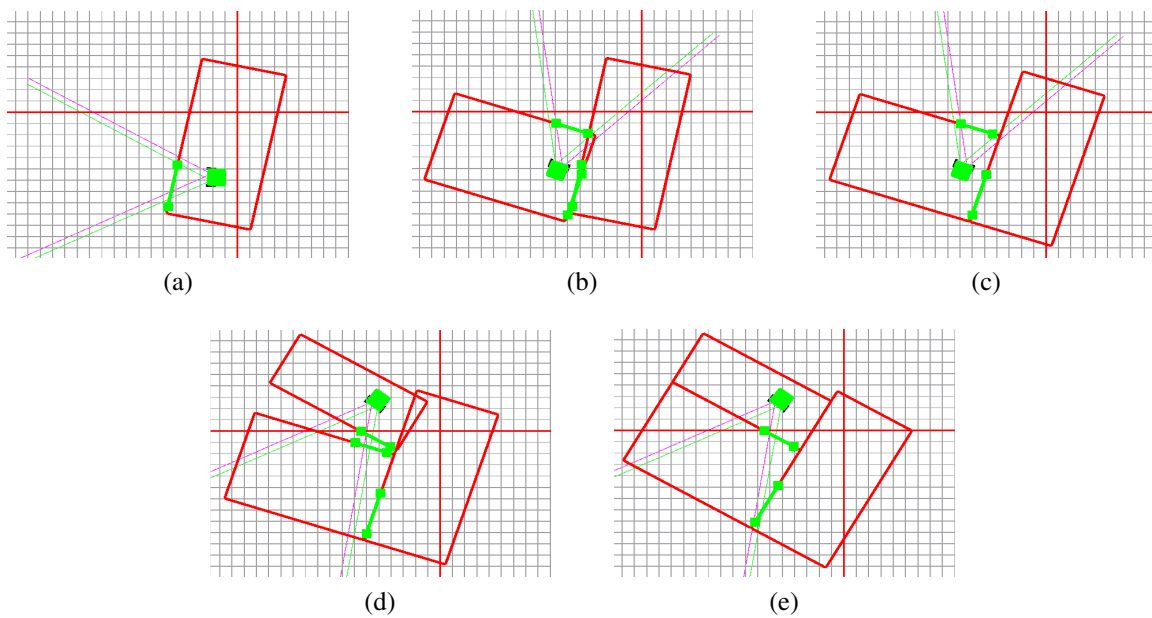


Figure 13.8: Modeling process during an autonomous exploration of the scene of figure 13.9.

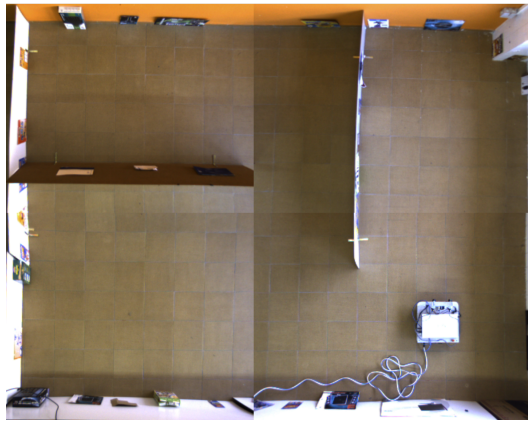


Figure 13.9: Overhead view of the real scene of the experiment of figure 13.8.

13.5 Conclusions

The main objective of the robot in this experiment was to model a real environment composed of simple rectangular rooms connected to each other, where some rooms had obstacles inside. It was also intended as an introduction to building hybrid representations and to think about how would these influence other problems such as localization.

The experiment is similar to [Thrun, 1998] in the sense that the resulting model is a hybrid representation composed of small room models. However, in this case, the model is built online, not once the exploration has been completely finished. Moreover, the main difference is that the models constructed using this system can be used not only as graphs of grid maps but also as graphs of symbols attributed with a high-order model (each room is modeled as a rectangle with its corresponding size and angle attributes), which is extraordinarily faster for planning in comparison to plain grid maps.

The method proposed in this experiment has two limitations worth mentioning. First, the system can only work with rectangular rooms. This is considered a minor limitation because the goal of the experiment was to enable robots to work with high-order models such as rectangles (instead of low-level occupancy grids only). However, the main limitation of the approach presented in the experiment is that once each room has been modeled, there is no principled criteria to trigger the process again in case there is a considerable error between the modeled room and the actual one. It would be desirable to know how well the data acquired by sensors fits the model the robot currently has and act accordingly, modifying the model or, at least, reinitializing it.

Despite not directly related to this particular experiment but with structured models in general, it was detected that the control system of robots is generally rather complicated. There are too many exceptional cases to consider and some of them depend on the context or the past experience of the robots. This non-desirable complexity increases the probability of introducing software errors overlooking details and is very time-consuming. This issue was one of the main motivations that led to AGM (chapter 9), which proposes using grammar rules to guide the robot and avoid the problems of the previous approaches.

Chapter 14

A graph grammar for indoor environment mapping

14.1 Experiment description

This chapter describes a small experiment, run before developing AGM, that was proposed in order to evaluate the concept of grammar-based perception and acquire information about needs and advantages their implementation. Without AGM, grammars were hard-coded and their explicit description was only used to help developers to organize how the system should work and how to implement the rules. The specific goal of the experiment was to model a real environment such as the one described in the experiment seen on chapter 13, that is, an environment composed of simple rooms with obstacles (see figure 14.1).

Since grammars were only used conceptually, no planning was actually performed yet (*i.e.*, there was no grammar-to-PDDL translation) and models were assumed valid (because models were verified using planners). The only feature that could be implemented as proposed by AGM at that point was the inclusion of context-aware restrictions, however, they also had to be implemented manually.

The experiment made use of a larger language for grammar definition than the one described in chapter 9. It was presented in [Manso et al., 2012] and its full implementation is left for future works. The most relevant difference is that when describing grammars it was also valid to specify negative subpatterns in the LHS of the rules (*i.e.*, patterns that could not be found in order to apply a rule). Those patterns are denoted by filling nodes in gray color.

The remaining of the chapter describes the symbols and grammar rules used, as well as the results drawn from the experiment.

14.2 Symbols

When using grammars in this context, the first step is to define the entities —symbols— the grammar will work with. These will not only depend on the domain of the problem to solve but also on the designer of the grammar since, to some extent, this is an arbitrary decision. For this experiment it was proposed to use symbols for rooms, doors and obstacles.

r Used for rooms.

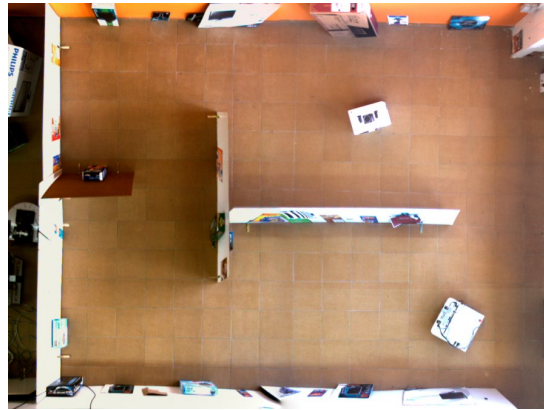


Figure 14.1: Overhead view of the environment used for the experiment. It is composed of four rectangular rooms connected in a loop. The robot is the squared object in the right-bottom part of the image. Objects are placed in the environment in order to prove the robustness of the approach.

- d** Used for doors. Doors will link two different rooms.
- o** Used for obstacles. Obstacles will be located within rooms.

Regarding attributes, rooms and obstacles carried information about their position (x and y) and their dimensions (*width* and *height*). Moreover, rooms also had an *active* attribute used to localize the robot in the graph; true for the room in which the robot was located and false otherwise. As seen in table 14.1, the formalism presented in [Manso et al., 2012] treated attributes as part of the grammar: there were attribute-based preconditions and effects. This was used in order to enable or disable the application of the different rules. However it is more elegant to avoid dealing with attributes in grammar rules and limit these descriptions to strictly grammatic issues. It must be noted that these restrictions can be introduced using grammar rules that check for the existence of accessory symbols that should or should not exist if a specific rule is to be executed.

14.3 Grammar rules

The next step is to specify the grammar rules. In most cases this is a cyclic process (*i.e.*, the symbols may depend on the rules and vice versa), and sometimes it will be necessary to introduce new symbols. The rules defined for the experiment—which are graphically represented in table 14.1—can be informally described as:

Rule 0 specifies that the start symbol can be transformed into a room. Since this is the only production with the start symbol in its left hand side, the start symbol *can only* be transformed into a room. See table 14.1a for a graphical description of the rule.

Rule 1 describes how the discovery of a new room transforms models. It connects a room to an adjacent room. See table 14.1b for a graphical description.

Rule 2 is similar to the rule number 1, but applicable to objects instead of rooms. Obstacles can be linked to rooms. See table 14.1c for a graphical description of the rule.

Rule 3 is used for loop closing. Specifically, for the case in which the robot realizes that two rooms previously modeled as disconnected are actually adjacent. In this case, a new door is included in the model without including a new room. See table 14.1d for a graphical description of the rule.

Rule 4 describes how would affect the graph the event of finding out that two rooms that were thought to be different are actually the same. This can happen when the robot closes a loop without recognizing that the new room it entered is already known. Both rooms collapse and all the ingoing and outgoing links are redirected from r_1 and r_2 to the new room r_3 . See table 14.1e for a graphical description of the rule.

14.4 Experiment results

Figure 14.1 shows an overhead view of the environment used for the experiment. It was composed of four rectangular rooms, so the system could execute the different production rules and eventually reach a map closure. The grammar described in section 14.3 was implemented in an autonomous robot in a similar way to how it was done for the experiment presented in chapter 13. The main difference is that in this case, development was guided by the grammar, so the different possible model modifications were clear when the rules were programmed. Moreover, despite the implemented system did not use PDDL mappings, it was used an extremely simple *behavior selection* algorithm that made the robot able to explore the environment: checking for all the potentially applicable rules including new symbols and selecting one of their associated actions randomly. Since the use of grammars does not affect the efficiency of the mapping algorithms, the results obtained were similar to those of the experiment of chapter 13.



The experiment started with a world model containing only the start symbol. According to the previously mentioned behavior selector the robot adopts the "explore room" behavior, expecting to run rule 0 (see table 14.1). Figure 14.2a shows the metric reconstruction of the map after the behavior successfully models the room. Once the robot accomplishes the task, the robot could potentially apply rules 1 and 2. Thus, it interleaves their associated behaviors in order to discover the next room. By repeating this process (*i.e.*, computing the candidate behaviors and interleaving them) the robot gets to the point in which all rooms and obstacles are modeled. This is shown in figure 14.2b. Figure 14.2c shows the final graph model. As it can be depicted in figure 14.2b, the metric reconstruction shows overlapping errors caused by odometry and sensor uncertainty. These errors were canceled by performing a stochastic gradient descent search in a parameter space defined by the size of the rooms and the positions of the doors (see [Bachiller et al., 2011]). The minimization was weighted by the uncertainty of the models, as in standard non-linear graph optimization. This improved the overall result and allowed the robot to perform robust loop-closings. The resulting metric reconstruction is shown in figure 14.3a.

14.5 Conclusions


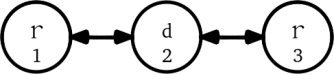
The results achieved where the expected and the use of grammar-based perceptual systems seemed promising. Having an explicit description of the rules of the grammar helped during the

Table 14.1: Grammar rules


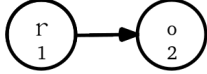
(a) Rule 0

| | | |
|-----------------------------------------------------------------------------------|---------------|-----------------------------------------------------------------------------------|
|  | \Rightarrow |  |
| Conditions: | | |
| Operations: $[r_2.active \leftarrow true]$ | | |
| Behavior: “explore room”. | | |


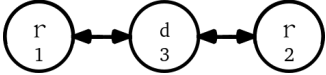
(b) Rule 1

| | | |
|-----------------------------------------------------------------------------------|---------------|------------------------------------------------------------------------------------|
|  | \Rightarrow |  |
| Conditions: $[r_1.active = true]$ | | |
| Operations: $[r_1.active \leftarrow false]; [r_3.active \leftarrow true]$ | | |
| Behavior: ”explore room“. | | |

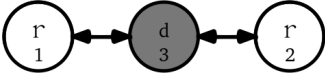

(c) Rule 2

| | | |
|-------------------------------------------------------------------------------------|---------------|--------------------------------------------------------------------------------------|
|  | \Rightarrow |  |
| Conditions: $[r_1.active = true]$ | | |
| Operations: | | |
| Behavior: “find new room”. | | |

(d) Rule3

| | | |
|-------------------------------------------------------------------------------------|---------------|--------------------------------------------------------------------------------------|
|  | \Rightarrow |  |
| Conditions: $[r_1.active = true]$ | | |
| Operations: | | |
| Behavior: “find new room”. | | |

(e) Rule4

| | | |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------------------------------------------------------------------|
|  | \Rightarrow |  |
| Conditions: $[r_1.active = true]$ | | |
| Operations: $[r_1.active \leftarrow false]; [r_4.active \leftarrow true];$ $[\forall a \in (A_{r_1} \cup A_{r_2}) A_{r_4} \leftarrow A_{r_4} \cup a]$ | | |
| Behavior: ”explore room“. | | |

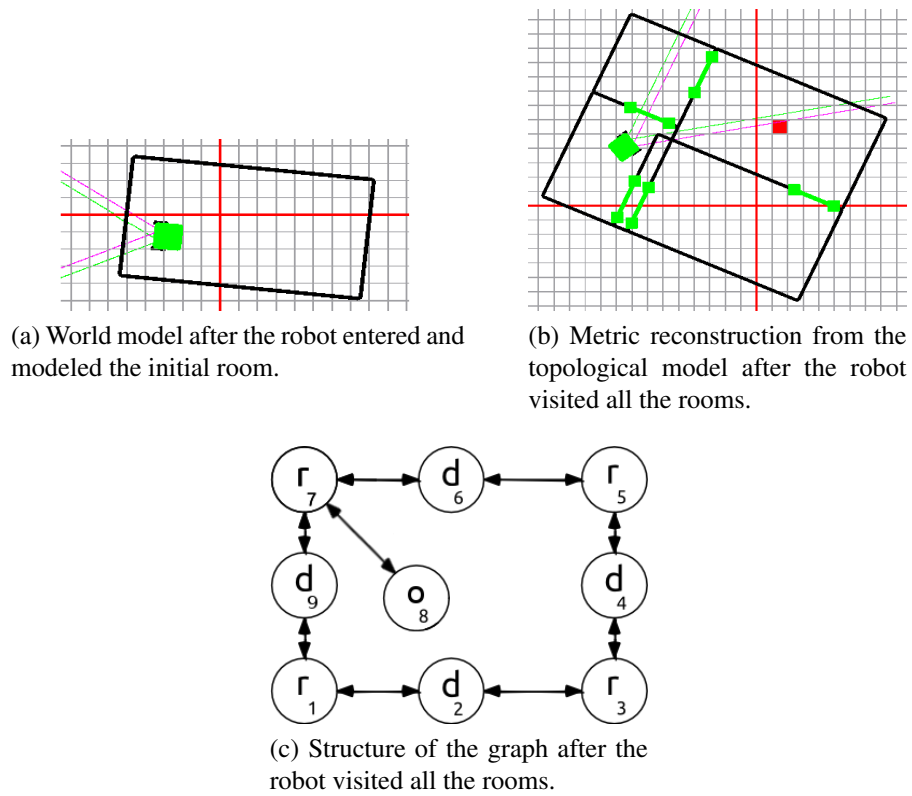


Figure 14.2: Results before optimization.

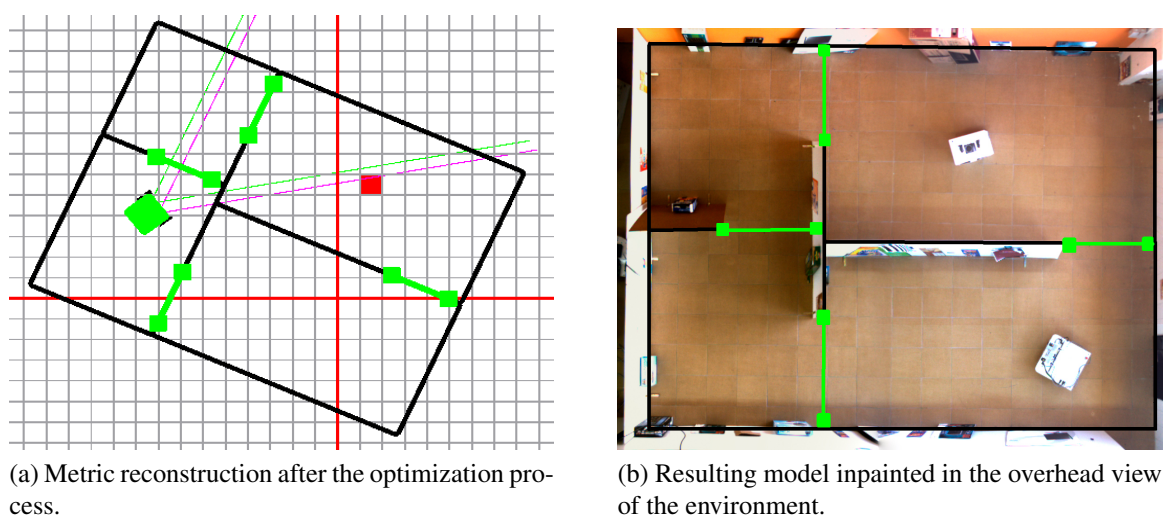


Figure 14.3: Results after optimization.

development process. However, since all the features had to be manually implemented, it provided little help regarding programming efficiency or reducing the number of programming errors.

The algorithm used for action selection was quite primitive and had to be improved. Given the absence of a proper mechanism, the changes made to the environment were assumed valid. As mentioned in chapter 9, both problems were later handled by Active Grammar-based Modeling using grammar-to-PDDL transformations. The results of this experiment were published in [Manso et al., 2012].

Chapter 15

Yaw estimation

15.1 Goal

Household robotics is one of the most attractive subfields of robotics. It is expected that these robots will have a huge number of applications in the future. Household robots operate in indoor environments, so it can be interesting to optimize their control systems for these scenarios. This experiment demonstrates how to achieve a fast and reliable method to filter the yaw angle of the odometry of indoor robots using the techniques shown in chapter 8.

In the experiment, it is used Manhattan assumption, that is, that in indoor environments what robots see is mostly composed of planar areas corresponding to perpendicular walls and other box-shaped objects aligned to them—or at least, that the robot can detect these cases or actively make them happen. This assumption was proposed in [Coughlan and Yuille, 1999], and has been successfully used in several works (see [Coughlan and Yuille, 1999], [Delage et al., 2006]).

15.2 Solution

The solution proposed to filter the yaw angle is to make use of annealed particle filtering along an algorithm used to process the RGBD images fetched by the robot, efficient enough to enable particles to be evaluated in real time.

In spite of the advances in obtaining orientation information using visual information only, it is hard to assure that the algorithms developed work as expected in daily situations. Most of these algorithms use the lines where the walls and the ceiling or the floor intersect, and this is only possible with: **1)** an appropriate point of view, **2)** good light conditions and **3)** low clutter. While the first requirement can be achieved with active vision, by relying on depth information we are able to remove the second and, to some extent, the third of these limitations.

The sensors used for the experiment are an RGBD camera and a three-axis linear accelerometer. Despite, according to the vendor of the device, the RGBD camera (an Asus Xtion Pro Live) only works in a $0.8m - 3.5m$ range, we have obtained valid measures up to a $0.55m - 6m$ range. However, these sensors have huge domestic applications and are expected to be improved in a short-time period. Moreover, the algorithm can also be run using other type of range sensors (with slight modifications) such as tilting or rotating lasers.

The observation model of the particle filter is based on a one-dimensional orientation histogram named *Vertical Surface Normal Histogram* (VSNH) generated using the depth information of the RGBD sensor. VSNH is a novel technique that allows to decrease the size of the input data of the evaluation of the particles from the number of pixels of the depth image (generally 640×480) to the size of the histogram. The size of the histogram can be adjusted depending on the desired precision: in this experiment we use a histogram of 256 bins. The rest of the chapter describes the models of the state space and the transition and observation of the particles.

15.2.1 State space and transition model

The state space used in this experiment is extremely simple, it is composed of a single scalar corresponding to the yaw of the robot:

$$X \equiv \text{"yaw"} \in \mathbb{R} \quad (15.1)$$

The transition model is driven by the odometry of the robot, specifically, using the increments of its yaw angle:

$$u_t \equiv \alpha_t - \alpha_{t-1} \quad (15.2)$$

where α_t stands for the yaw at time t as provided by the odometry.

Since robots generally provide good odometry results locally and there is no other information that might influence the state, we assume that a state only composed of the yaw can be safely considered complete. Relying on the Manhattan assumption, normal vectors can only provide information of the yaw in a modulo $\pi/2$ range (*i.e.*, there is no a priori difference between the four possible wall orientations). However, it is not considered a problem: even in the case of a total *robot kidnap*¹, it should be moved at least $\pi/4$ radians in less time than the period of the filter to confuse the robot². Therefore, the transition model is set as a univariate Gaussian distribution with the sum of the previous state and the yaw increment (estimated by the odometry) as mean. The variance, σ_{odom} , is arbitrarily set according to the accuracy of the odometry of the robot (see equation 15.3):

$$P(x_t | x_{t-1}, u_t) \approx N(x_{t-1} + u_t, \sigma_{odom}) \quad (15.3)$$

Before the observation model is explained it is necessary to describe the two preprocessing stages that allow us to generate the *VSNH* histogram that the observation model relies on (to estimate the PDF of a yaw angle given the input point cloud) and makes the filter efficient. First the acquired point cloud is used to obtain a set of oriented three-dimensional points (where the orientation of the points correspond to the surface normal corresponding to each point). Then, the obtained oriented points are used to generate the VSNH histogram. Only once the histogram has been generated the particles can be weighted.

¹A robot kidnap is a scenario in which a robot is moved so that its odometry system provides no information.

²Even assuming a relatively long time period of 0.1 seconds (10 Hz), it would have to be moved at 7.85 rad/s to be confused, that is 450 deg/s .

15.2.2 Oriented point extraction

Oriented points are regular points associated to a vector —usually the normal vector of the surface corresponding to the point. There are many different approaches to estimate normal vectors from point clouds, (see [Badino et al., 2011] and [Rusu, 2009] for updated reviews of the different approaches). All of them proceed by selecting as many subsets of points as normal vectors they provide and estimating for each of those a normal vector. The accuracy, resolution and computational efficiency of a method depends on how the subsets of points are selected and how the normal vector for each set is estimated. The position of the point is generally set as the mean of each of the selected sets of points.

Given a specific set of points, the most widely used method for estimating the normal vector associated to a point cloud subset —also the most accurate in moderate noise scenarios— is the well-known least squares method. The method can be solved in closed form by computing the eigenvector associated to the smallest eigenvalue of the covariance matrix of the points in the selected subset. Thus, the normal vector v is computed by solving equation 15.4 for v and selecting the non-degenerated solution with the smallest eigenvalue λ^3 .

$$Cv = \lambda v \quad (15.4)$$

where

$$C = \frac{1}{N} \sum_{i=1}^N ((p_i - \bar{p})(p_i - \bar{p})^T) \quad (15.5)$$

The smallest eigenvalue provides information on how close the points in each sample set are to the estimated plane. Thus, in order to ignore normals that do not correspond to a wall (a planar surface), the normal vectors whose associated eigenvalue is larger than a threshold are discarded. In order to take into account the fact that the noise of the data acquired from the sensor grows with distance, this threshold T_λ is adapted to grow with the euclidean distance to the 3D centroid associated to the point cloud of the window:

$$T_\lambda = z^2 K_\lambda, \quad (15.6)$$

where z is the z coordinate of the centroid and K_λ is an arbitrary constant experimentally adjusted for the sensor.

Regarding neighbor selection for generating the set used to estimated normal vectors, using the k -nearest neighbors (KNN) provides very accurate results (see [Badino et al., 2011]) and can be computed in short time, however, when using depth disparity maps or structured point clouds there are faster approaches that provide results almost as good as KNN and are easier to parallelize. Our approach consists on creating overlapping windows in the depth buffer and generating a normal vector for every window by including all of its points. The windows should have a size small enough to provide local information and generate a relatively high number of informative vectors, but they should also be big enough to avoid being considerably affected by noise. In order to increase the number of estimated normal vectors, the windows are overlapped so that the centers of the windows form a two-dimensional grid in which nodes are separated by half the window size. Because of the large number of depth points acquired by the depth sensor (640×480 points) and the window overlapping used, it is easy to find a balanced size; we used windows of 20×20 points.

³All normal vectors are assumed to have a negative z coordinate (*i.e.*, all normals which do not are inverted).

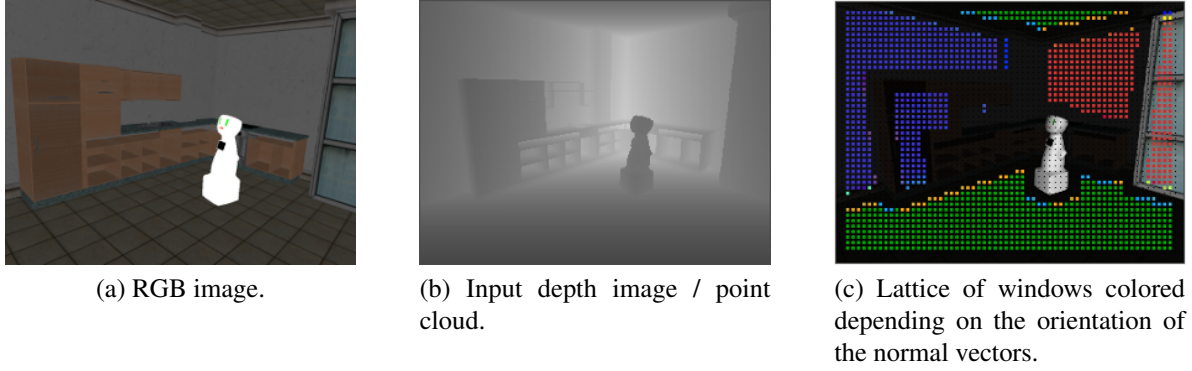


Figure 15.1: Input and result of the estimation of the oriented points.



Figure 15.2: Output of the preprocessing stage: obtaining the Vertical Surface Normal Histogram.

15.2.3 Histogram generation

The second preprocessing stage generates the histogram using the normal vectors of the oriented points computed previously. Figure 15.2 provides a graphical representation of the output of the whole preprocessing stage.

In order to insert the normal vectors in the histogram each normal vector is rotated to cancel the pitch and roll of the camera. Afterwards, their x and z coordinates are used so the yaw angle of the patch is computed as:

$$\alpha_n = \text{atan2}(n_{i.x}, n_{i.z}) \quad (15.7)$$

Figure 15.3 illustrates this step. Note that we will assume that the x axis points towards the right, the y axis points up, the z axis to the front and we use the *rule of the left hand* to determine angle signs.

The orientation cues available in indoor scenarios present a 4-folded ambiguity, so there is no logical reason to choose the normal angle of any of the four walls a room may have as the angle of reference. The cues provided by planar normal vectors also have such ambiguity, any normal vector associated to a planar surface provide positive evidence for four angles: yaw , $yaw + \pi/2$, $yaw + \pi$ and $yaw - \pi/2$. Thus, the histogram only needs to cover the angles in the range $[0, \pi/2)$. Moreover, to take this into account and increase the weight w_i of the normal vectors corresponding to the depth image patches that most likely belong to the walls, they are weighted according to the product of their eigenvalue (that provides information on how *planar* an image patch is) and the length of the normal vector after ignoring its y coordinate (that approximates to 1 as the normal vector of the patch gets perpendicular to the gravity vector):

$$\alpha_i = \text{fmod}(\pi + \text{atan2}(n_{i.x}, n_{i.z}), \frac{\pi}{2}) \quad (15.8)$$

$$w_i = \frac{\text{hypot}(n_i.x, n_i.z)}{1 + \lambda_i} \quad (15.9)$$

where n is the normal vector to include and λ its corresponding eigenvalue. After updating the histogram according to equations 15.10 and 15.11 with each of the normal vectors, the histogram is normalized so it actually describes a probability density function. For every normal vector:

$$b = \text{integer}\left(\frac{2 \alpha_i \text{len}(H)}{\pi}\right) \quad (15.10)$$

$$H[b] \leftarrow H[b] + w_i \quad (15.11)$$

where H is the histogram to be updated, and $\text{len}(H)$ its number of bins. The process is detailed in algorithm 15 and illustrated in figure 15.3.

Algorithm 12 Histogram generation algorithm:

Require: P : a vector containing $BINS_H$ \langle vector-eigenvalues \rangle tuples $\langle n_i, \lambda_i \rangle$

```

1:  $H = \text{vector}(BINS_H)$ 
2: forall  $p = \langle n, \lambda \rangle$  in  $P$  do
3:    $\alpha = \text{fmod}(\pi + \text{atan2}(n.z, n.x), \frac{\pi}{2})$ 
4:    $\text{bin} = \text{int}((\alpha * BINS_H) / \frac{\pi}{2}) \text{ modulo } BINS_H$ 
5:    $\text{weight} = \text{hypot}(n.x, n.z) / (1. + \lambda)$ 
6:    $H[\text{bin}] += \text{magnitude}$ 
7: end forall
8:  $\text{sumval} = 0$ 
9: forall  $i$  in  $(0, \dots, BINS_H - 1)$  do
10:    $\text{sumval} += H[i]$ 
11: end forall
12: forall  $i$  in  $(0, \dots, BINS_H - 1)$  do
13:    $H[i] = H[i] / \text{sumval}$ 
14: end forall
15: return  $\text{sum} / N$ 
```

15.2.4 Observation model

Particles are evaluated according to the correlation between the histogram that the particle *expects* to obtain and the one that the particles actually obtain. Its worth noting that, as seen in chapter 8.2.1.4 the weights of the particles only need to resemble $p(z_t|x_t)$ up to a scale factor, so there is no problem if the sum of the weight exceeds 1. The *expected* histogram is computed using a wrapped Cauchy distribution, as seen in algorithm 13. This distribution was selected because, among all the symmetric wrapped distributions tested, it is the one that better fits the empirical data obtained.

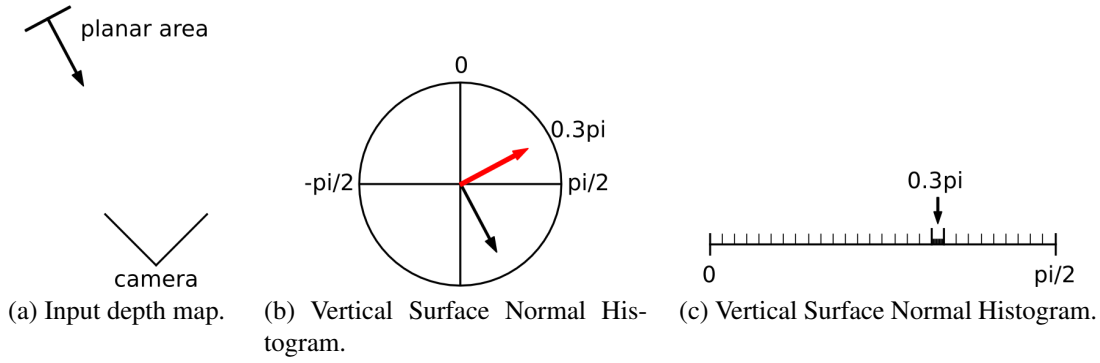


Figure 15.3: Graphical representation of the process of including a normal vector in the Vertical Surface Normal Histogram.

Once both histograms have been defined, the observation model is approximated as equation 15.12:

$$P(z_t|x_t) \propto \text{correlation}(H_{\text{expected}}, H_{\text{obtained}}) + 1, \quad (15.12)$$

where H_{expected} is the histogram expected for the particle and H_{obtained} is the histogram actually computed from the sensor. It must be noted that in order to implement the particle filter it is not actually necessary to generate the histogram for each of the particles. Thanks to the use of a symmetric wrapped distribution, any desired histogram can be obtained by selecting the appropriate portion of two consecutive two zero-mean histograms, so that the particles use only the part corresponding to their mean, requiring less computational resources (see figure 15.4).

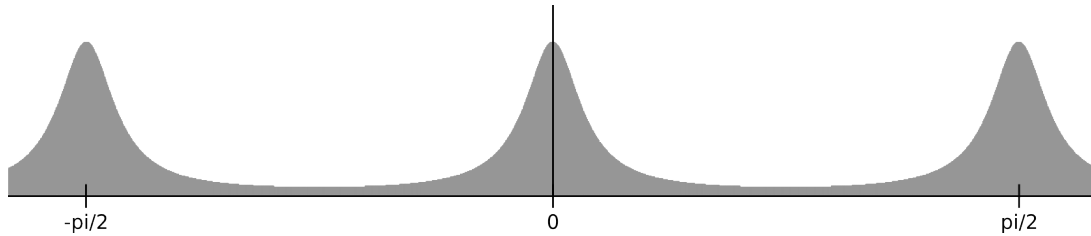


Figure 15.4: Example of the wrapped Cauchy probability density function used in the observation model of the particle filter.

15.3 Results and conclusions

Given that the error of dead reckoning systems drift over time, experimental results are provided as a plot comparing the angle provided by the particle filter with the one of a magnetometer (see figure 15.5). Despite a magnetometer does not provide accurate measurements they do not present drifts as dead reckoning does. A demonstration video can also be found in <http://ljmanso.com/thesis>.

Figures 15.6 and 15.7 provide screenshots of *cloudComp*, the component used to compute the oriented points and VSNH (figure 15.6), and of *gualzru_yaw* the component that implements the filter (15.7).

This chapter has described a useful particle filter-based approach for estimating the yaw angle of indoor robots and how to compute VSNH, a type histogram used in order to weight

Algorithm 13 Expected histogram generation algorithm:**Require:** μ : The mean of the distribution associated with the generated histogram**Require:** σ : The variance of the distribution associated with the generated histogram**Require:** $BINS_H$: Size of the histogram

```

1:  $H = \text{vector}(BINS_H)$ 
2: for  $i$  in  $(0, ..BINS_H)$  do
3:    $v = \frac{(2\pi)*(i)}{BINS_H}$ 
4:    $H[i] = \frac{\sinh(\sigma)}{2\pi(\cosh(\sigma) - \cos(v-\mu))}$ 
5: end for
6:  $sumval = 0$ 
7: for  $i$  in  $(0, ..., HISTOGRAM_{BINS} - 1)$  do
8:    $sumval += H[i]$ 
9: end for
10: for  $i$  in  $(0, ..., HISTOGRAM_{BINS} - 1)$  do
11:    $H[i] = H[i]/sumval$ 
12: end for
13: return  $H$ 

```

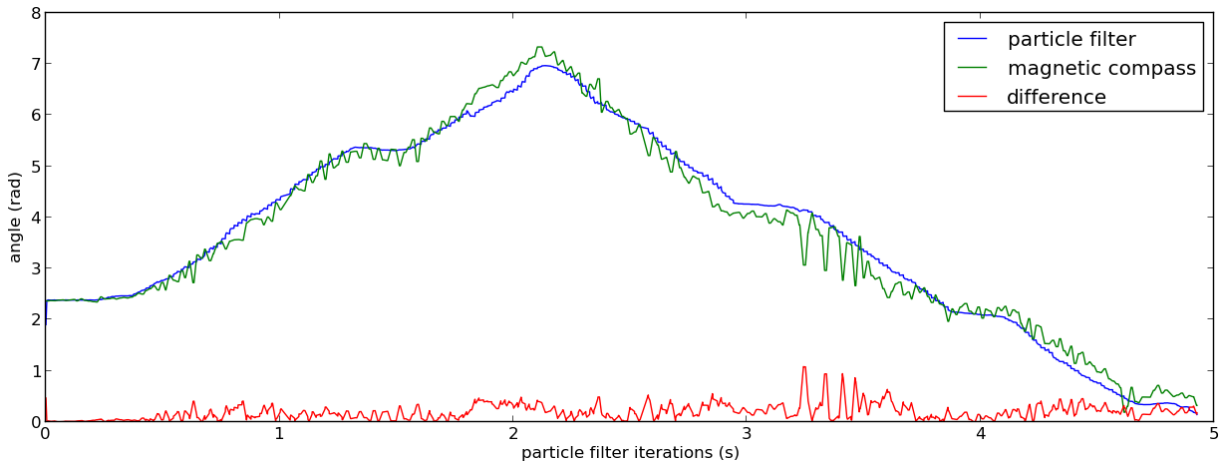


Figure 15.5: Plot depicting the angles provided by the particle filter (blue), the compass (green) and the difference between them (red). It can be observed that the angle provided by the particle filter is less noisy. Of course, these results can only be obtained in indoor environments satisfying the Manhattan assumption.

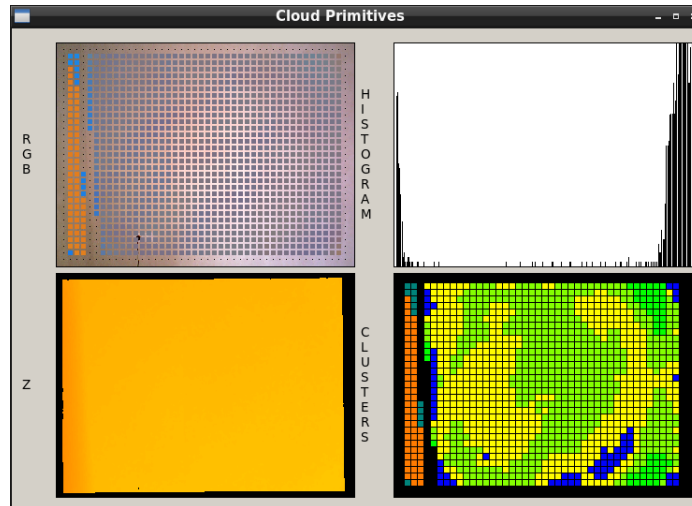


Figure 15.6: Graphical interface of the component extracting the oriented surface points: cloud-CPU. Top-left: RGBD image with the lattice of windows corresponding to the detected oriented points (colored according to the orientation). Bottom-left: depth image. Bottom-right: oriented points clustered according to their similarity. Top-right: resulting VSNH histogram.

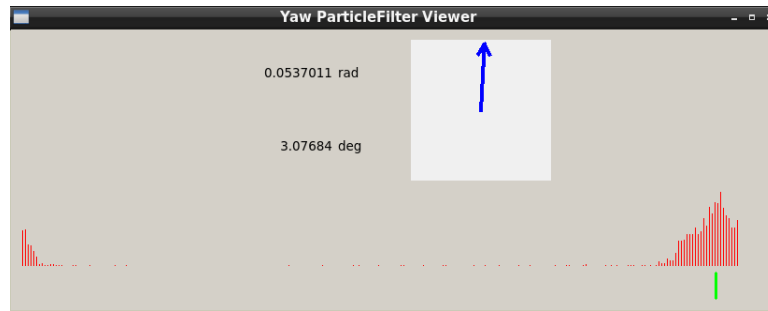


Figure 15.7: Graphical interface of the developed filter: the blue arrow indicates the orientation of the robot. The red histogram correspond to the current VSNH. The green bar under the histogram indicates the position of the best particle.

particles. The experimental results shown in figure 15.5 demonstrate that the filter is robust and can perform in real time. The work described in this experiment will also be used in further experiments.

Chapter 16

Distance estimation

16.1 Goal

Estimating the parameters of a plane from the perspective of the robot is a useful ability that indoor robots should have. It is very useful for feature-based localization and mapping purposes, since planes constitute one of the most ubiquitous features of indoor environments. The objective of this experiment is to test the validity of particle filtering to estimate the minimum distance from the RGBD sensor of the robot to a plane of known relative angle.

We will differentiate two contexts. In the first one the robot performs on-line filtering while it moves. In the second scenario it performs an initial estimation of the parameters without moving.

16.2 Solution

To achieve the objective of the experiment we propose a similar approach to the one used for the experiment shown in chapter 15: using a particle filter to estimate the state. The information in which the algorithm relies on is also the same as the one in the experiment shown in chapter 15, a set of points decorated with the normal corresponding to their local neighborhood. Additionally, the developed particle filter also makes use of a small preprocessing stage that enhances the performance of the filter.

For the first scenario, in which it is desired to achieve an initial estimation while the robot is still, it is proposed to use annealed particle filtering. On the other hand, in order to filter the relative parameters of the plane as the robot moves, it is proposed to use standard particle filtering. However, for both scenarios it is used the same state space, transition and observation models. The only difference is that in first scenario the noise induced to the particles decreases with time and the robot is still; whilst in the second scenario the robot moves and the noise only depends on such movement.

The filter—which will also be used in the experiments described in chapters 17 and 18 to model a rectangular room—receives as input three (A , B and C) of the four parameters of the plane equation:

$$Ax + By + Cz + D = 0 \quad (16.1)$$

When estimating the distance to the floor, given the pitch (r_x) and roll (r_z) of the camera, the

parameters A , B and C can be computed as:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(r_x) \\ 0 & \sin(r_x) & \cos(r_x) \end{pmatrix} \begin{pmatrix} \cos(r_z) & -\sin(r_z) & 0 \\ \sin(r_x) & \cos(r_z) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad (16.2)$$

whilst, in order to estimate the distance of the i -th wall they are computed as:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-r_x) & -\sin(-r_x) \\ 0 & \sin(-r_x) & \cos(-r_x) \end{pmatrix} \begin{pmatrix} \cos(r_{offset}) & 0 & \sin(r_{offset}) \\ 0 & 1 & 0 \\ -\sin(r_{offset}) & 0 & \cos(r_{offset}) \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad (16.3)$$

where α_{offset} stands for:

$$\alpha_{offset} = yaw - \frac{i\pi}{2} \quad (16.4)$$

16.2.1 State space and transition model

The state used in the particle filters of this experiment is the minimum distance to the plane we want to estimate, from the point of view of the robot (*i.e.*, the D parameter of the plane equation).

$$X \equiv \text{"distance"} \in \mathbb{R}. \quad (16.5)$$

The initialization of the particles is depends on the purpose of the filter. If used to estimate the distance to a totally unknown plane particles are initialized using a uniform distribution in the range of the sensor. However, if a priori information is available, such as when estimating the distance to the floor, the particles are initialized using a normal distribution.

The transition model is set as an univariate normal distribution with the sum of the previous distance and the distance increment as mean. Such increment, u_t , which represents the control of the particle filter, is estimated by the odometry. The transition model used for the walls is mathematically described in equation 16.6:

$$P(x_t|x_{t-1}, u_t) \approx N(x_{t-1} + u_t, \sigma_{odom}) \quad (16.6)$$

where the variance, σ_{odom} is arbitrarily set according to the properties of the odometry of the robot. On the other hand, in case of estimating the distance to the floor, the transition model is not influenced by the robot moving, so it can be simplified to:

$$P(x_t|x_{t-1}, u_t) \approx N(x_{t-1}, \sigma_{odom}) \quad (16.7)$$

The transition model deals only with the distance because the filter is designed to work in conjunction with previously gathered information regarding the orientation of the robot in the room (see chapter 15). Moreover, it is assumed that the walls are vertical to the floor —whose pitch and roll is previously estimated using an inertial sensor.

16.2.2 Observation model

The observation model is designed as a two-dimensional Gaussian mixture model where a kernel is introduced for each of the normal vectors detected. The dimensions correspond to the

euclidean distance from the point associated with the normal vector to the particle being evaluated and their angular distance, both being trivial to calculate. Therefore, the observation model is mathematically defined as:

$$P(z_t|x_t) \propto \sum_{i=1}^K N \left(\begin{pmatrix} \alpha_i - \alpha_{particle} \\ d_i - d_{particle} \end{pmatrix}, \mu, \sigma \right) \quad (16.8)$$

where K is the number of kernels, $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and σ depends on the usage of the filter:

- In still estimation mode: $\sigma_e * k^t$, where the variance σ_e and the annealing constant k are experimentally set.
- In regular filtering mode: σ_f , a fixed-value variance experimentally set.

16.2.3 Implementation considerations

Initially, the evaluation of a particle would require the summation of a Gaussian kernel for each of the patches extracted from the depth image. Using a 640×480 depth image, the number of patches may range from 566 (using patch windows of 32×32) to 1481 when (using a window size of 20×20). Given that the number of particles should be relatively high, the overall number of kernel summations can exceed 3×10^5 .

In order to reduce the computational resources needed to run the filter, similar patches are first clustered using K-means clustering. The restriction used to decide if a pair of patches can be clustered or not is that their angular and euclidean distance must be under a threshold, where the euclidean distance between two patches a and b is computed as the minimum of the following:

- the distance from the centroid of the patch a to the plane defined by the patch b
- the distance from the centroid of the patch b to the plane defined by the patch a .

16.3 Results and conclusions

The experimental results are provided as a plots comparing the resulting distances with their corresponding ground-truth in two different scenarios. The scenario in which the filter estimates the distance to a wall is ignored in this case because the results are similar if initialized without a good a priori. The scenarios are the following:

1. estimating the height of the camera (see figure 16.1),
2. estimating the distance to a wall while the robot moves (see figure 16.2).

The ground truth for the experiments was obtained manually for the first scenario and using a Hokuyo UTM-03LX laser rangefinder for the second.

As can be appreciated in figures 16.1 and 16.2 the results obtained demonstrate that the filter proposed can be used for a wide range of applications in indoor scenarios that do not require extreme precision. Moreover, the small error obtained should not only be attributed only to the

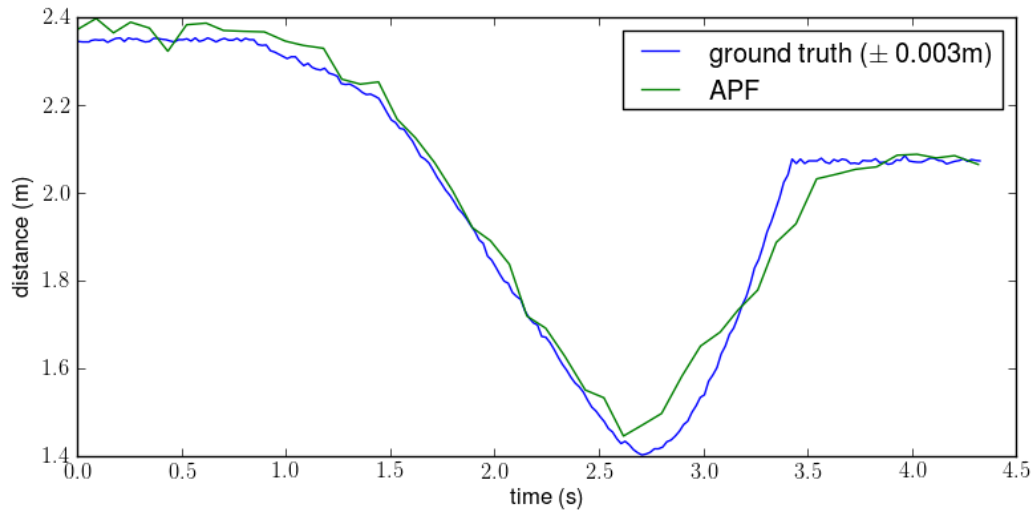


Figure 16.1: Results obtained when estimating the distance to a wall while the robot moves using the annealed particle filter described in this chapter.

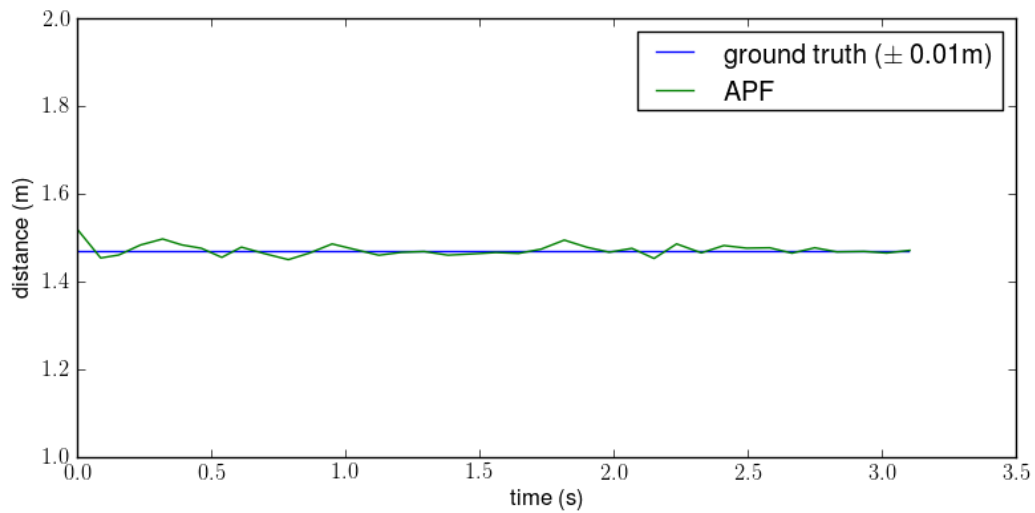


Figure 16.2: Results obtained when estimating the distance to the floor using the annealed particle filter described in this chapter.

filter but also to the input sensor, which is about 30 times cheaper than the sensor used to obtain the ground truth.

This experiment has described a useful particle filter-based approach for estimating and filtering the distance from the RGBD sensor of the robot to any wall. The experimental results shown in figures 16.1-16.2 demonstrate that the filter is robust and can perform in real time. The work described in this experiment will allow us to estimate models for the floor and walls in experiments 17 and 18. Additional demonstration videos can also be found in <http://ljmanso.com/thesis>.

Chapter 17

Room modeling

17.1 Experiment description

In the experiment presented in chapter 14, the grammar was not used at room level but to guide the whole map building process. This means that the room was seen as an atomic element which had to be perceived by using very simple behaviors along with an atomic *room detector*. This is possible in simple scenarios such as the ones used in chapter 14, but in cluttered environments this is hardly an option. In several scenarios such as cluttered wall detection or object recognition, it is necessary to perform rather complicated behaviors in order to perceive. In the case of recognizing an object such as a mug, it is possible that the robot must take a walk around the table while pointing a camera to the mug, avoiding obstacles at the same time. In AGM, these behaviors are seen as the result of the interaction of one or more agents that have the purpose of perceiving or modifying environment items or modifying their connection patterns within the world representation.

In order to achieve a more appropriate behavior and a better results this experiment studied how to implement room perception in an actual Active Grammar-based Modeling-based system. The chapter describes the AGM system used to perceive rooms—including not only the grammar but also the agents used—and the results obtained from its execution.

17.2 Symbols

This section describes the symbols used in the AGM system. Despite the attributes of the symbols are ignored in a grammar level, the attributes that the symbols will carry are also described in order to make the text easier to understand (note that the symbols with no attributes have only syntactic meaning):

startsymbol The start symbol the representation will be composed of. No attributes.

robot The symbol of the robot. No attributes.

gravity A symbol used temporarily to denote that the gravity has been modeled. Attributes: *g*.

plane A symbol used temporarily to store the ground plane once it has been modeled. Attributes: *pitch* angle, *roll* angle, *d* (distance).

floor A symbol used temporarily to store the ground plane once it has been modeled. Attributes: *pitch* angle, *roll* angle, *yaw* angle, *distance*.

notWall Temporary symbol used to denote walls that are known to exist but have not been modeled yet. No attributes.

wall Temporary symbol used to denote walls that are known and have been modeled. Attributes *d* (distance).


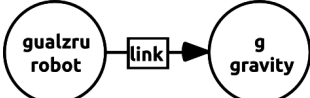
room The final symbol for the room. It has the same attributes than the *orientedfloor* symbol (*pitch*, *roll*, *yaw*, and *distance*), but *room* symbols can only exist once the four walls have been modeled. The size of room symbols can be computed using the distance of the robot to each of the walls.

17.3 Grammar rules

Since the purpose of the robot in this experiment is limited to modeling the room in which it is currently located, the grammar only cover this issue.

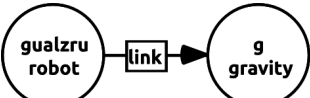

Rule 1 (shown in table 17.1) determines the first step (the only first step, since no other rule has its LHS composed only of the start symbol). It enables agents to modify the representation so that a model containing the start symbol is converted into a model containing the robot and the gravity symbols. It is used to model the gravity. Its associated behavior is “modelPitchAndRoll” (see section 17.4).

Table 17.1: Rule 1: used to model the gravity.

| | |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|  |  |
| Behavior: “modelPitchAndRoll”. | |

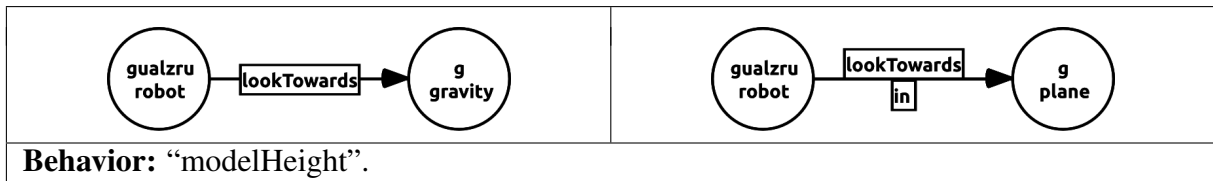
Rule 2 (see table 17.2) renames the link connecting the robot and the gravity from *linked* to *lookTowards*. It is used to denote that the robot is looking downwards. Its associated behavior is “lookUnknownFloor” (see section 17.4).

Table 17.2: Rule 2: used to denote that the robot is looking downwards.

| | |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|  |  |
| Behavior: ”lookUnknownFloor“. | |

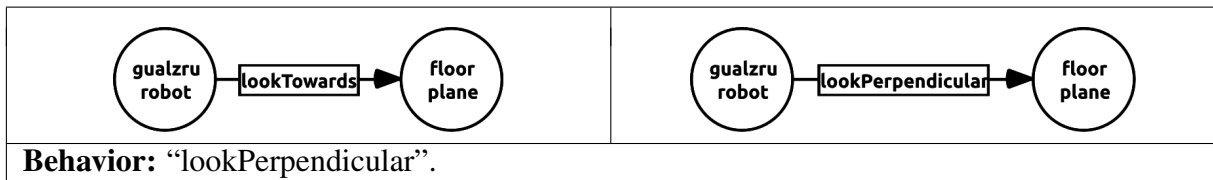
Rule 3 (see table 17.3) modifies the type of the *g* gravity symbol to plane and adds a new link from the robot symbol to it. It is used to denote that the robot has modeled the height of its camera. Its associated behavior is “modelHeight” (see section 17.4).

Table 17.3: Rule 3: used to denote that the robot has modeled the height of its camera.



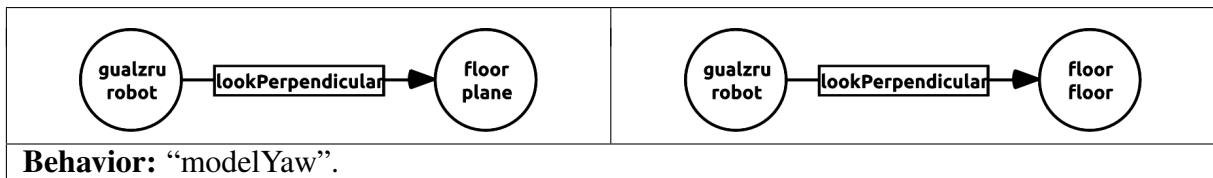
Rule 4 (see table 17.4) renames the label of the link going from the robot to the floor from “lookTowards” to “lookPerpendicular”. It is used to denote that the robot does not look towards the floor anymore but perpendicular to it. Its associated behavior is “lookPerpendicular” (see section 17.4).

Table 17.4: Rule 4: used to denote that the robot is looking perpendicular to the floor.



Rule 5 (see table 17.5) modifies the type of the node floor, from “plane” to “floor”. A *plane* is supposed to be a regular plane (containing a plane equation), whilst *floor* is supposed to be an “oriented plane” (*i.e.*, a plane with a *north*). Thus, it is used to denote that the bearing angle of the robot has been modeled. Its associated behavior is “modelYaw” (see section 17.4).

Table 17.5: Rule 5: used to denote that the bearing angle of the robot has been modeled.



The behavior of rule 6 (see table 17.6) performs two interesting modifications. First, it creates a list of four “notWall” symbols —not modeled walls— and appends the list to the existing floor symbol. Second, it removes the link connecting the robot and the floor (labeled as “lookPerpendicular”) and creates a new one (labeled as “looks”) from the robot to the first of the not modeled walls. It is used to represent that the robot looks toward the wall at *zero* angle and that there are four walls yet to model in the room in which the robot is located. Its associated behavior is “lookFirstWall” (see section 17.4).

Rule 7 (see table 17.7) moves the head of the arrow from a modeled wall to a succeeding not modeled wall. It is used to denote that the robot does not model the last wall anymore but the next not modeled wall, that is, to look to the next not modeled wall. Its associated behavior is “lookNextWall” (see section 17.4).

Rules 8 and 9 (see tables 17.8 and 17.9) modify the type of a not modeled wall to “wall”. Both rules are used to denote that a wall has been modeled. Additionally, rule 9 also changes the type of the floor from “floor” to “room”, denoting that the room has been completely modeled.

Table 17.6: Rule 6: used to look to the first wall

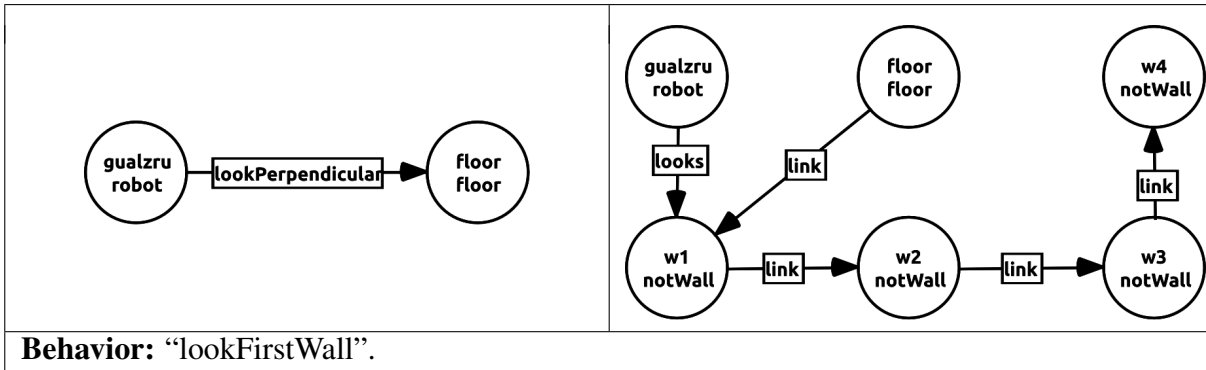
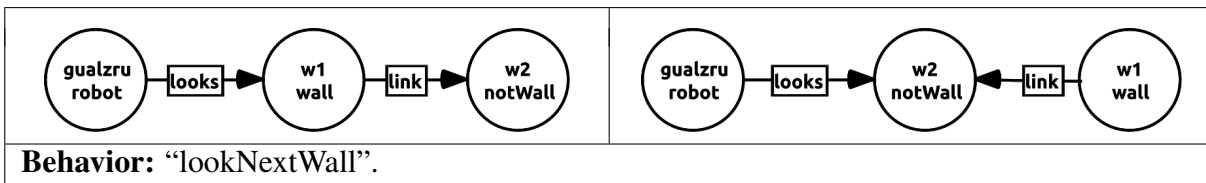
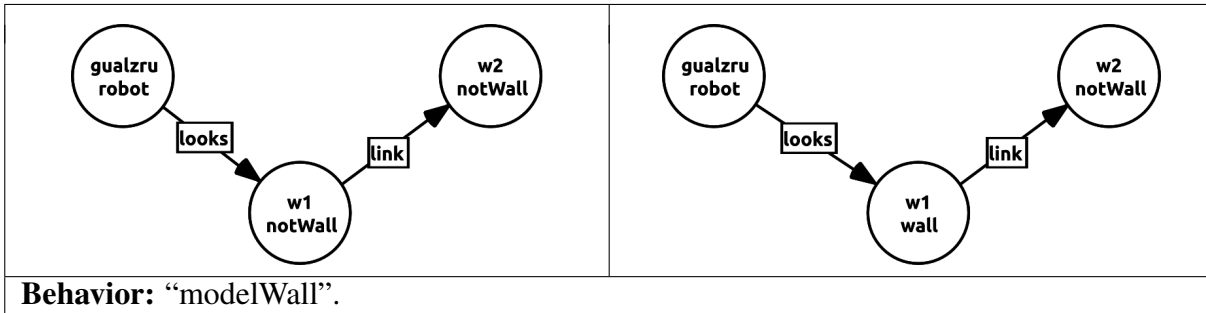


Table 17.7: Rule 7: used to denote that the robot has switched its gaze to the next not modeled wall.



The difference is that rule 8 is used to model all the walls but the last one, while rule 9 has the opposite purpose. Their associated behavior is “modelWall” (see section 17.4).

Table 17.8: Rule 8: used to denote that the robot has switched its gaze to the last not modeled wall.

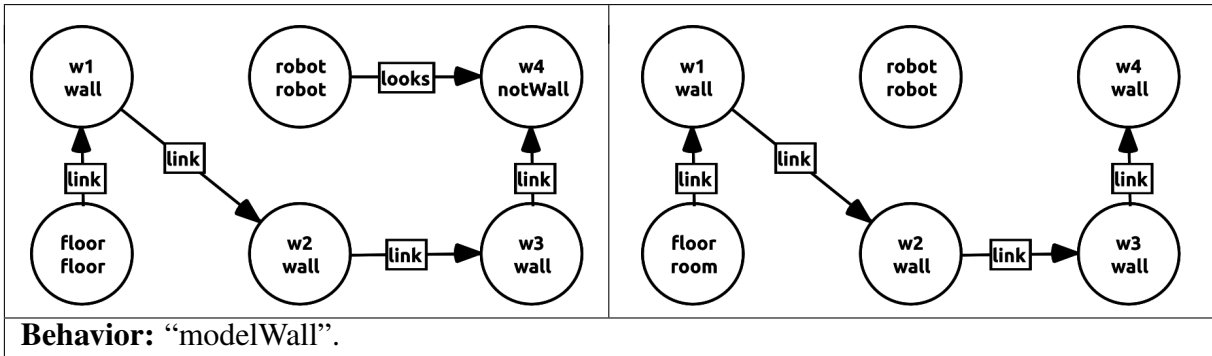


17.4 Agents and behaviors

AGM rules do not model or interact with the environment by themselves, they are “only” used to plan and verify actions and model modifications. Such actions and modifications are implemented in agents that are activated (with potentially different configurations) or deactivated depending on the context and the goals of the robots. Therefore, the AGM-based system implemented for this use case has the following agents:

pitchNRoll: In charge of modeling the pitch and roll angles of the camera of the robot. It uses a simple annealed particle filter for such purpose. Implements rule 1.

Table 17.9: Rule 9: used to denote that the robot has modeled the last wall.



yaw: In charge of modeling the yaw angle of the camera of the robot. It uses the particle filter described in chapter 15. Implements rule 5.

saccade: In charge of performing camera saccades. Implements rules 2, 4, 6 and 7. It has two modes, *downwards* to look towards the floor, *frontal* to look to the front. Its behavior is deterministic.

distance: In charge of modeling and maintaining floor and wall distances. It uses the particle filter described in chapter 16. Implements rules 3, 8 and 9. Two modes, *floor* to model the height of the camera of the robot, *wall* to model camera-to-wall distances.

Table 17.10 describes how agents are configured depending on the behavior:

Table 17.10: Behavior configuration table for this experiment.

| conf. \ agent | pitchNRoll | yaw | saccade | distance |
|--------------------------|------------|--------|-----------|----------|
| modelPitchAndRoll | active | - | - | - |
| lookUnknownFloor | - | - | downwards | - |
| modelHeight | - | - | - | floor |
| lookPerpendicular | - | - | frontal | - |
| modelYaw | - | active | - | - |
| lookFirstWall | - | active | firstWall | - |
| modelWall | - | - | - | wall |
| lookNextWall | - | active | nextWall | - |

17.5 Implementation and software components

The whole system was implemented using component-oriented programming using Robo-Comp. The executive and each of the agents were implemented as independent components. Additional components for hardware access, the processing of the acquired point cloud and the execution of platform and camera coordinated saccades are used. The whole network of components is shown in figure 17.1:

- the components implementing the hardware abstraction layer are arranged in the first column
- the components in charge of processing the point cloud (cloud) and executing the saccades (saccade) are located in the second column
- in the third column are shown the implementation of the different agents (pitch, saccade, dist, yaw) and IceStorm (part of the ZeroC Ice middleware, used to enable components to communicate using publish/subscribe).
- the three remaining components correspond to the PDDL planner (planning), the executive (executive) and —since the executive does not have a user interface— a component used to visualize in real-time the model maintained by the executive (visualizer).

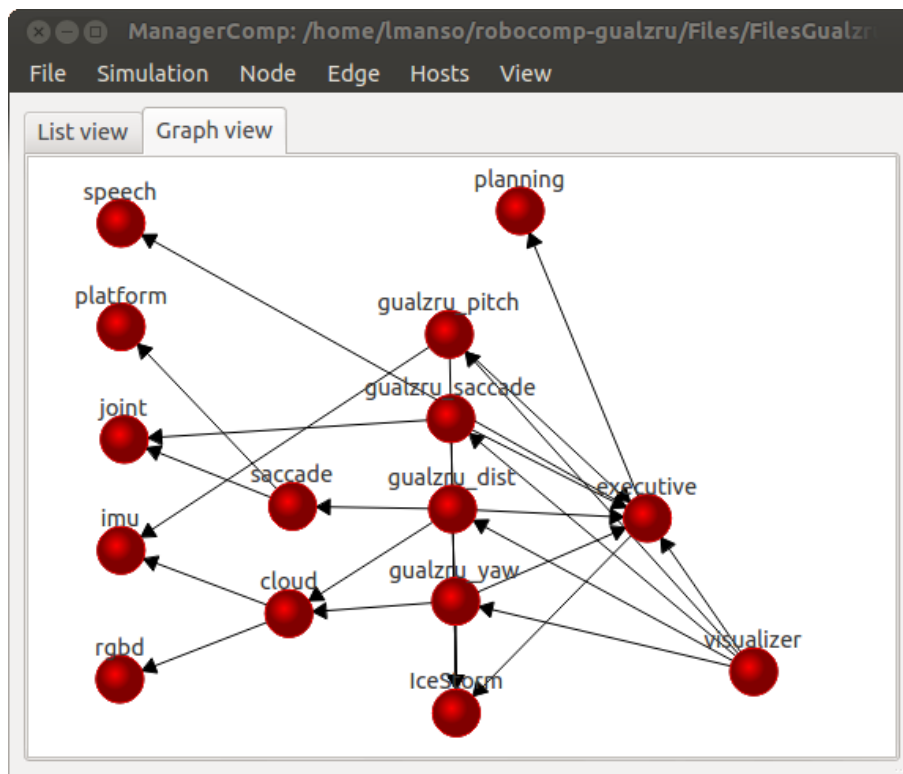


Figure 17.1: Screenshot of RCManager showing the software components used.

17.6 Experiment results and conclusions

The robot was required to model a room in order to validate the system. The environment in which the experiment was run is shown in figure 17.2. It is a room with two tables, and several obstacles: three chairs, a coat stand, a radiator and a litter bin.

The advantages of using Active Grammar-based Modeling in the experiment can be classified in two groups: those due to the use of the architecture and those due to the use of the Active Graph Grammar Language.

Table 17.11: Events and actions during the experiment.

| source of the event | event description |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| executive | The experiment begins. The executive analyzes the mission and uses the planner to get the first action of the plan with the lowest cost that would get the robot to the target state. As a result, the <i>gualzru_pitch</i> agent is activated (mode <i>modelPitchAndRoll</i>). |
| pitch | The agent <i>gualzru_pitch</i> models the pitch and roll angles of the camera, successfully triggering rule 1. The executive acts according to the plan, activating the <i>gualzru_saccade</i> agent (mode <i>LookUnknownFloor</i>). |
| saccade | The agent <i>gualzru_saccade</i> makes the robot look downwards and reflects it in the model (rule 2). The executive, activates the <i>gualzru_distance</i> agent (mode <i>modelHeight</i>). |
| distance | The agent <i>gualzru_distance</i> models the distance to the floor (rule 3). Agent <i>gualzru_saccade</i> is activated (mode <i>lookPerpendicularFloor</i>). |
| saccade | Agent <i>gualzru_saccade</i> makes the robot look to the front (rule 4). Agent <i>gualzru_yaw</i> is activated (mode <i>modelYaw</i>). |
| yaw | Agent <i>gualzru_yaw</i> models the yaw of the robot with respect an arbitrary wall of the room (rule 5). Agents <i>gualzru_yaw</i> and <i>gualzru_saccade</i> are activated (modes <i>Yaw</i> and <i>LookFirstWall</i> , respectively). |
| saccade | Agent <i>gualzru_saccade</i> makes the robot look towards the closest wall (rule 6). Agent <i>gualzru_distance</i> is activated (mode <i>ModelWall</i>). |
| distance | Agent <i>gualzru_distance</i> models the distance to the wall (rule 8). Agents <i>gualzru_yaw</i> and <i>gualzru_saccade</i> are activated (modes <i>Yaw</i> and <i>LookFirstWall</i> , respectively). |
| saccade | The agent <i>gualzru_saccade</i> makes the robot look towards the next wall (rule 7). Agent <i>gualzru_distance</i> is activated (mode <i>ModelWall</i>). These two last events are repeated three times. |
| distance | The agent <i>gualzru_distance</i> models the last wall (rule 9). The goal is achieved and the executive disables all behaviors. |

Regarding the architecture, the loosely coupled implementation of the executive, the planner, and more importantly, the agents, makes very easy to include new agents and grammar rules and reduces undesirable inter-module interactions (deadlocks or any other kind of side-effect of sharing the memory space). The implementation of these modules as RoboComp components enables users to decrease programming errors and develop faster, thanks to the use of the DSLs that RoboComp provides. Moreover, the architecture itself provides a flexible framework in which a wide variety of problems can be solved, computation is naturally distributed and code is easily reused (the executive, planner and the agents that may be useful to solve new problems).

The use of the Active Graph Grammar Language language also provides very interesting advantages. As opposed to what happened in experiment 14, describing the world grammar using AGGL made us able to automatically generate the code implementing the grammar, not only to plan the necessary action to get to the goal (perceptive or general) but also for model



Figure 17.2: Picture of the environment in which the system was tested.

checking. Code auto-generation is a very important feature, it helps developers to reduce errors and spend less time designing and integrating the grammar. Moreover, the use of a human-friendly language is a valuable means to allow people with no programming skills to take part in the development process. This is particularly important when the team developing has people from other fields different (*e.g.*, psychologists, medical practitioners).

The experiment was successfully performed. The results achieved were similar to those of experiment 14, since the accuracy is related to the modeling algorithms not to the architecture. However, this solution is much more scalable, flexible, fast to develop and easy to parallelize and understand by third-party developers.

Including a new agent optimizing the overall model (*e.g.*, refining the model by looking at the corners of the room) would have been an interesting improvement. However, these improvements lay outside the boundaries of this work and are left for future works.

Chapter 18

A graph grammar for scene understanding

18.1 Experiment description

The experiment presented in chapter 17 showed how AGM can be used to enable robots to autonomously model rooms. Since rooms have a particular known structure, in that experiment the robot did not have to find anything it did not know where to look. Behaviors only had to direct the gaze to the floor to model it and afterwards, once the orientation of the room was also modeled, to each of the walls. In this chapter we extend the experiment shown in chapter 17 and demonstrate how cognitive subtraction can be used within Active Grammar-based Modeling to guide robot perception. Specifically, the goal of the robot in this experiment is to model all the objects it finds in its environment. Tables and mugs —for which the robot can provide a high-order model— are modeled as such, the rest of objects are modeled as *obstacle* objects and the robot generates a bounding cylinder for them.

To achieve the goals, three new agents are included in the grammar. The first of them, *gualzru_unknown* corresponds to the implementation of the cognitive subtraction algorithm described in chapter 10. It is in charge of including all unexpected objects in the model. In order to work, the agent uses an instance of the RCIS simulator that is synchronized with the world model. The other two, *gualzru_tableFit* and *gualzru_mugFit* implement two simple algorithms to convert symbols corresponding to unknown objects to symbols of tables and mugs, when appropriate.

18.2 Symbols

The set of symbols managed by the grammar of chapter 17 is extended to include the new entities we want to deal with. As in the previous grammar-related experiment, a description of the attribute of the symbols is also provided (despite such attributes are not handled by the grammar):

obstacle Used for models of unknown obstacles. Attributes: geometric center and radius.

table Used for models of modeled tables. Attributes: geometric center, radius and height.

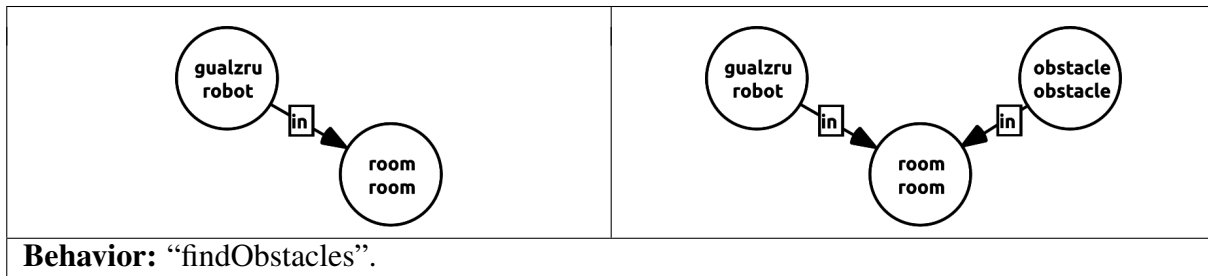
object Used for models of non-modeled objects in tables. Attributes: geometric center and radius.

mug Used to represent mugs. Attributes: extremes of the cylinder associated to the mug and radius.

18.3 Grammar rules

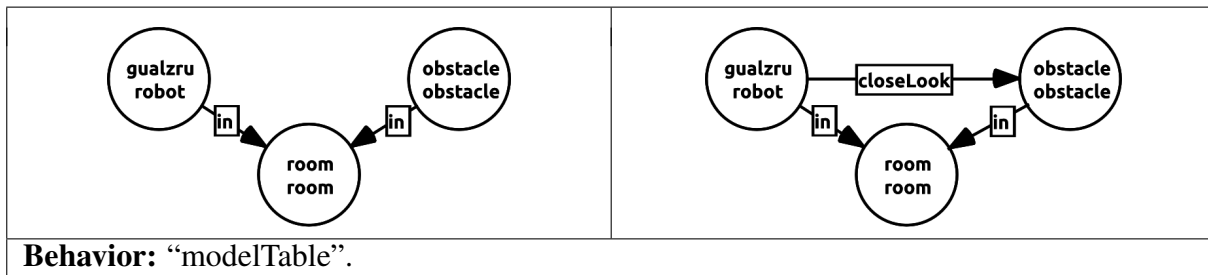
Rule 10 (shown in table 18.1) creates and links new *obstacle* symbols to the *room* symbol associated to the robot. It is used to include obstacles (unknown objects located in the floor) in the representation.

Table 18.1: Rule 10: used to include obstacles in the representation.



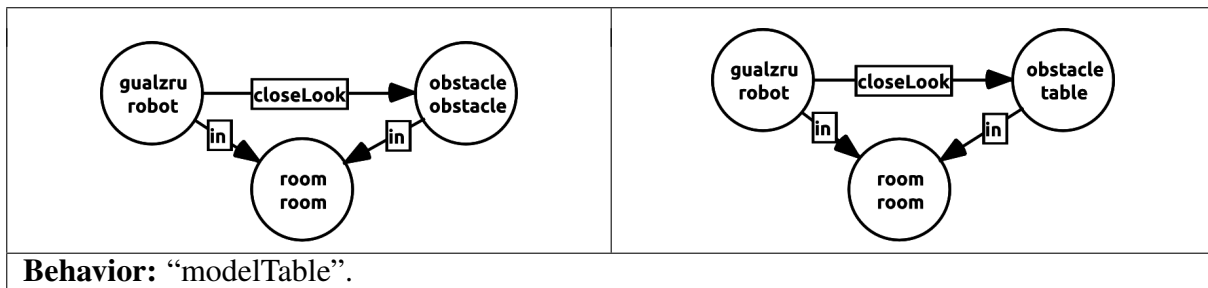
Rule 11 (shown in table 18.2) creates a new link from the robot to *obstacle* symbols. It is used to denote that the robot is looking towards a specific obstacle from a close position.

Table 18.2: Rule 11: used to denote that an obstacle is being seen by the robot.



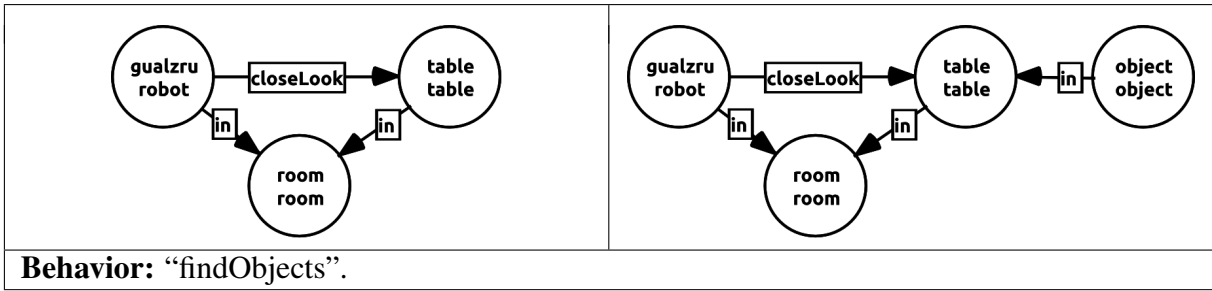
Rule 12 (shown in table 18.3) modifies the type of *obstacle* symbols to *table* symbols. It is used to denote that an obstacle has been successfully modeled as a table.

Table 18.3: Rule 12: used to denote that an obstacle has been successfully modeled as a table.



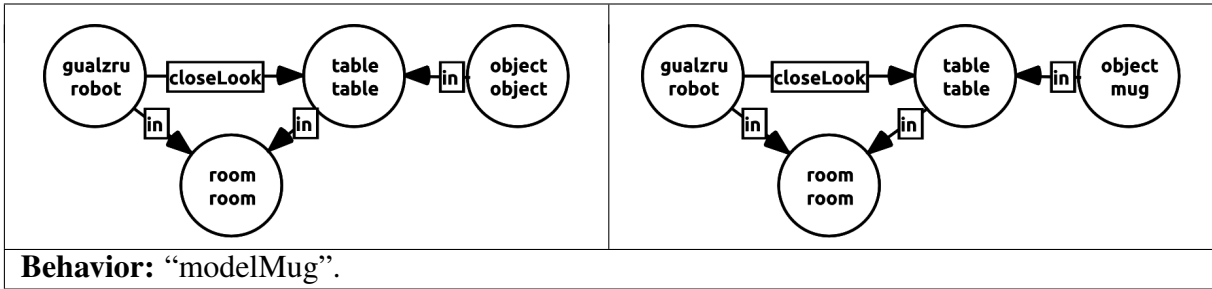
Rule 13 (shown in table 18.4) links *object* symbols to *table* symbols. It is used to include unmodeled objects on tables in the world representation.

Table 18.4: Rule 12: used to include unmodeled objects on tables in the world representation.



Rule 14 (shown in table 18.5) modifies the type of *object* symbols to *mug* symbols. It is used to denote that an object in a table has been successfully modeled as a mug.

Table 18.5: Rule 13: used to denote that an object in a table has been successfully modeled as a mug.



18.4 Agents and behaviors

In order to achieve the goals, three new agents are included in the grammar:

gualzru_unknown corresponds to the implementation of the cognitive subtraction algorithm described in chapter 10. In order to work, the agent uses an instance of the RCIS simulator that is synchronized with the world model. Includes all unexpected objects in the model. It implements rule 10 and 12 and update the attributes of *obstacle* and *object* symbols.

gualzru_tableFit implements a simple algorithm to convert symbols corresponding to unknown objects in the floor to symbols of tables. Implements rule 11.

gualzru_mugFit implements a particle filter used to convert symbols corresponding to unknown objects in tables to symbols of mugs. Implements rule 13.

Table 18.6 describes the configuration of the agents that are used in the behaviors introduced for this experiment, those agents not appearing are disabled in all of the new behaviors:

Table 18.6: Behavior configuration table for this experiment.

| conf. \ agent | yaw | saccade | unknown | table | mug |
|----------------------|--------|------------------|------------|--------|--------|
| findObstacles | active | scanRoom | roomLevel | - | - |
| approachTable | active | approachObstacle | roomLevel | - | - |
| modelTable | - | - | roomLevel | active | - |
| findObjects | - | scanTable | tableLevel | - | - |
| modelMug | - | - | tableLevel | - | active |

Regarding implementation issues, the only differences with the experiment in chapter 17 the extension of the grammar, the creation of a new mode for agent *gualzru.saccade* (mode *approachObstacle*) and the inclusion of the new agents described. The graph resulting from the components used in the system and their interconnections is shown in figure 18.1.

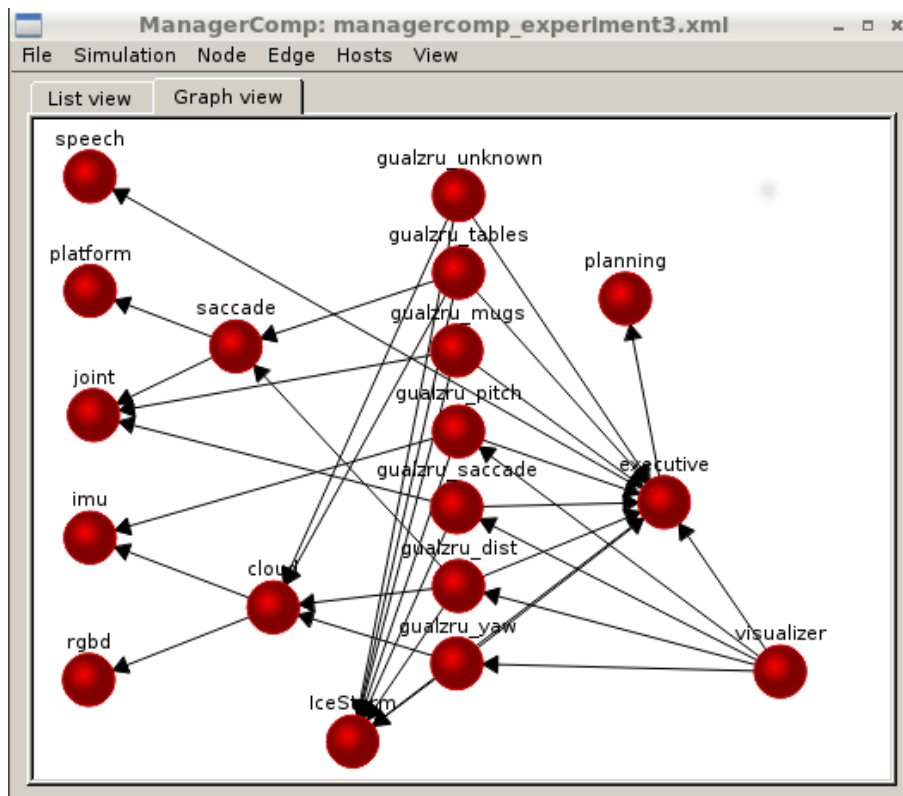


Figure 18.1: Screenshot of RCManager showing the software components used.

18.5 Experiment results and conclusions

In this case, the robot was required to model a room and the objects within it. The environment in which the experiment was run is the same that the one used in chapter 17, shown in figure 17.2. Table 18.7 shows the events that the system triggered after modeling the room (the previous ones ignored because they are similar to the ones in the experiment of room modeling).

Table 18.7: Events and actions during the experiment.

| source of the event | event description |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| unknown | The agent <i>gualzru_unknown</i> detects an obstacle (rule 10). Behavior <i>approachTable</i> is selected. |
| saccade | Agent <i>gualzru_saccade</i> makes the robot closer to the obstacle (which is actually a table) and triggers rule 11. Behavior <i>modelTable</i> is selected. |
| table | Agent <i>gualzru_table</i> models the obstacle as a table (rule 12). Behavior <i>findObjects</i> is selected. |
| unknown | Agent <i>gualzru_unknown</i> detects an object in the table (rule 13). Behavior <i>findMug</i> is selected. |
| mug | Agent <i>gualzru_mug</i> models the object as a mug (rule 14). The robot accomplishes the mission. |

Part IV

Conclusions

Chapter 19

Review and contributions

Many different contributions have been made to **RoboComp** during the development of this thesis. In particular, the author has developed some of the most important tools of the framework (RCManager, RCMonitor, RCReplay, RCInnerModelSimulator, RCInnerModelEditor), and has contributed to some extent to almost any considerable aspect of the framework, some of the most remarkable being: development of the component model, compilation system, development of the component interfaces or the development of many different components and libraries. Moreover he has also designed and implemented the InnerModel and component deployment Domain-Specific Languages and contributed to the design (not the implementation) of the rest of them. Chapter 12 provides the results obtained from an experiment that evaluates the impact of the use of model-driven engineering in the development of software components for robotics.

The thesis proposed **Active Grammar-based Modeling** to solve or mitigate several problems related to active perception. It is a concrete technology that uses grammar rules associated to active-perceptive configurations to overcome the passive nature of particle filtering and related methods. This way, robots are provided a principled way of **a)** moving to gather information about all the variables the state space is composed of and **b)** building and updating representations of dynamically changing and unknown state spaces. Two experiments (see chapters 17 and 18 support the proposal). Besides its use to plan how to perceive the environment, AGM can be used to introduce context-aware restrictions on perception, to verify model modifications and to enable covert perception. Moreover, it can also exploit context to perceive the environment properly by selecting different particle filters (or sets of them) depending on the context.

Active Graph Grammar Language is a visual domain-specific language specifically designed to facilitate the implementation AGM-based systems. The need for such language was one of the conclusions drawn from the experiments described in chapters 13 and 14: implementing grammar rules manually was error-prone and time-consuming. Describing grammar rules in a visual DSL allows to automatically transform such descriptions into PDDL, the most common language for AI planning.

Vertical Surface Normal Histogram, presented in chapter 15, is a histogram obtained as the result from processing three-dimensional cloud points. It is designed to be used in indoor environments to help robot localization. Chapter 15 presented a particle filter that makes use of this technique to estimate and filter the yaw angle of an indoor robot. Another useful particle filter for indoor environments is presented in chapter 16, in this case it is used to model and track planes.

Grammar-based Cognitive Subtraction, refers to the use of robotics simulators in AGM-based systems to detect unexpected percepts such as unknown or incorrectly modeled objects. It is used to enable robots to localize these objects and correct their model or include a new one in the representation using AGM. Chapter 10, deeply describes the problem and an algorithm to enable this kind of process in robots.

Active Grammar-based Modeling, cognitive subtraction, and the previously mentioned particle filters are used in the experiment on chapter 18 to model a room and the objects within it (obstacles, tables, and mugs). In the experiment, once the room has been modeled, the robot detects unknown objects which attract the attention of the robot. This unleashes an attentional loop in which the robot tends to satisfy its curiosity and attempts to model everything it does not expect unless it is provided with a specific mission that do not requires modeling all objects.

All the software developed during this work will be published with a Free/Libre/Open Software license in <http://ljmanso.com/thesis>.

Chapter 20

Future works

An interesting topic to continue working in the future would be **extending the Active Graph Grammar Language** in order to support negative subpatterns, quantified formulas (*i.e.*, counterparts for the quantified formulas *forall* and *exists* of PDDL) and type inheritance. It would also be interesting to improve the API of the AGM-related libraries and increase the amount of code that can be auto-generated when using AGM within RoboComp. These improvements would help enhancing the efficiency of the roboticists designing AGM-based systems.

The use of the RoboCompInnerModelSimulator (RCIS) for cognitive subtraction and prediction tasks is also interesting. Specifically, we think enabling RCIS to perform *heterogeneous simulation* would be worth researching. With heterogeneous simulation we refer to the possibility of updating each of the bodies of the simulation using potentially different simulation algorithms (probably in separated software components). Computer simulation is an extraordinarily complex topic. There are different libraries that generally provide reasonably good results, however, they require considerable computational resources and provide poor results in some cases. In order to reduce the computational resources needed for simulation and/or to improve its results, we plan to support ad-hoc simulation for specific nodes. The simulator would be specified in the simulated world definition file how each node should be simulated: static or no simulation (by default), using the Bullet physics engine, or using a third-party component providing an specially tailored simulation interface.

Regarding Active Grammar-based Modeling it would also be of great interest to perform a biggest experiment in which AGM is used to model several rooms and the entities of the model are not only created and updated (*e.g.*, moved, resized) but also removed. Using other type of polygons for rooms besides rectangles would also be interesting, since it would enable robots to operate in a wider range of environments. Despite this was not implemented in any experiment because they were only designed to demonstrate the validity of the AGM approach, it is considered a very desirable feature for the future.

Chapter 21

Publications

2013:

1. “Ursus: A robotic assistant for training of children with motor impairments”. C. Suarez-Meís, C. Echevarría, P. Nuñez, **L.J. Manso**, P. Bustos, S. Leal and C. Parra. In *Converging Clinical and Engineering Research on Neurorehabilitation Biosystems & Biorobotics*. Springer, vol. 1, pp. 249-253.

2012:

1. “Experiments in self-calibration of an autonomous mobile manipulator”. A. Sánchez, P. Núñez, **L.J. Manso**, P. Bustos In *Proc. of Workshop of Physical Agents 2012*. September 2012.
2. “Indoor scene perception for object detection and manipulation”. **L.J. Manso**, P. Bustos, P. Bachiller and J. Franco In *Proc. of Symposium on Spatial Cognition in Robotics. International Conference on Spatial Cognition*. September 2012.
3. “Graph Grammars for Active Perception”. **L.J. Manso**, P. Bustos, P. Bachiller and Marco A. Gutierrez In *Proc. of 12th International Conference on Autonomous Robot Systems and Competitions* ISSN 978-972-98603-4-8, pp 63-68. April 2012.
4. “Engaging human-to-robot attention using conversational gestures and lip-synchronization”. F. Cid, R. Cintas, **L.J. Manso**, L. Calderita, A. SÁnchez and P. Núñez. In *Journal of Physical Agents* ISSN 1888-0258. Vol. 6, No. 1, pp 3-10. March 2012.
5. “Interactive games with robotic and augmented reality technology in cognitive and motor rehabilitation”. A.B. Naranjo, C. Suárez, C. Parra, E. González, F. Böckel, A. Yuste, P. Bustos, **L. Manso**, P. Bachiller and S. Plana. In *Serious Games as Educational, Business, and Research Tools: Development and Design*. ISBN 978-1-4666-0149-9. Vol. 1, pp. 1212-1233. February 2012.

2011:

1. “Improving the life cycle of robotics components using Domain Specific Languages”. A. Romero-Garcés, **L.J. Manso**, M.A. Gutiérrez, R. Cintas and P. Bustos. In *Proc. of 2nd International Workshop on Domain-Specific Languages and models for ROBotic systems* (DSLRob’2011). September 2011.
2. “An incremental hybrid approach to indoor modeling”. P. Bachiller, M-A. Gutierrez, **L.J. Manso**, P. Bustos and P. Núñez. In *Proc. of European Conference on Mobile Robots* 2011. September 2011.
3. “A real-time synchronization algorithm between Text-To-Speech (TTS) system and Robot Mouth for Social Robotic Applications”. F. Cid, R. Cintas, **L.J. Manso**, L. Calderita, A. Sánchez and P. Núñez. In *Proc. of Workshop of Physical Agents* 2011. September 2011.
4. “Attentional Behaviors for Environment Modeling by a Mobile Robot”. P. Bachiller, P. Bustos and **L.J. Manso**. *Advances in Stereo Vision*, InTech, ISBN 978-953-307-837-3, pp 17-40. July 2011.
5. “Robust Behavior and Perception using Hierarchical State Machines: A Pallet Manipulation Experiment”. R. Cintas, **L.J. Manso**, L. Pinero, P. Bachiller and P. Bustos. In *Journal of Physical Agents*, ISSN 1888-0258. Vol. 5, No. 1, pp 35-44. March 2011.

2010:

1. “RoboComp: a Tool-based Robotics Framework”. **L.J. Manso**, P. Bachiller, P. Bustos, P. Núñez, R. Cintas and L. Calderita. In *Simulation, Modelling and Programming for Autonomous Robot*. Springer LNAI 6472, ISBN 978-3-540-89075-1, pp 251-262. November 2010.
2. “Improving a Robotics Framework with Real-Time and High-Performance Features”. J. Martínez, A. Romero-Garcés, **L.J. Manso** and P. Bustos. In *Simulation, Modelling and Programming for Autonomous Robot*. Springer LNAI 6472, ISBN 978-3-540-89075-1, pp 263-274. November 2010.
3. “RobEx: an Open-hardware Robotics Platform”. J. Mateos, Agustín Sánchez, **L.J. Manso**, P. Bachiller and P. Bustos. In *Proc. of Workshop of Physical Agents*, pp 17-24. September 2010.
4. “Visually-guided Object Manipulation by a Mobile Robot”. L. Pinero, R. Cintas, **L.J. Manso**, P. Bachiller and P. Bustos. In *Proc. of Workshop of Physical Agents*, pp 145-152. September 2010.
5. “Un Framework de Desarrollo para Robótica”. **L.J. Manso**, P. Bustos, P. Bachiller, P. Núñez, R. Cintas and L. Calderita. In *Proc. of I Jornadas Jóvenes Investigadores*, pp 33-38, ISBN 978-84-693-1707-5. Cáceres, April 2010.
6. “Multi-cue Visual Obstacle Detection for Mobile Robots”. **L.J. Manso**, P. Bustos, P. Bachiller and J. Moreno. In *Journal of Physical Agents*, ISSN 1888-0258. Vol 4, No 1, pp 3-10. January 2010.

2009:

1. "Obstacle Detection on Heterogeneous Surfaces Using Color and Geometric Cues". **L.J. Manso**, P. Bustos, P. Bachiller and J. Moreno. In *Proc. of Workshop of Physical Agents*, pp 95-101, Cáceres, September 2009.
2. "Navegación Visual en Robots Móviles". *MSc Thesis* **L.J. Manso**. Cáceres, July 2009.

2008:

1. "Attentional Selection for Action in Mobile Robots". P. Bachiller, P. Bustos, **L.J. Manso**. In *Advances in Robotics, Automation and Control*. I-Tech, pp 111-136, ISBN 78-953-7619-16-9. November 2008.

Index

- absolute representation, 57
- Active Grammar-based Modeling, 98
- Active Graph Grammar Language, 103
- aggl2pddl, 102
- AGGLEditor, 102
- AGGL, 103
- AGM, 97, 98
- ancestral sampling, 79
- Annealed Particle Filtering, 88
- APF, 88
- auxiliary particle filter, 89

- Bayesian filter, 62
- belief distribution, 60, 63
- belief space, 59
- bootstrap resampling, 77
- Branched Iterative Hierarchical Sampling, 91

- Cognitive Subtraction, 116
- complete state, 59, 60
- Component Description Specific Language, 34
- component interface, 11
- component-oriented programming, 11
- COP, 11
- corrected belief distribution, 63
- correction, 63
- CS, 116

- deliberative robots, 53
- Deployment Domain-Specific Language, 38
- deterministic representation, 59
- Domain-Specific Language, 18

- endpoint, 15

- Gibbs sampling, 83
- grammar, 98

- hardware independence, 9
- histogram filter, 65
- hybrid representation, 56

- hybrid robots, 53

- IDL, 14
- importance sampling, 75
- initial belief distribution, 63
- InnerModelDSL, 40
- Interface Definition Language, 14
- Interface Description Specific Language, 37

- Markov chain, 60
- Markov Chain Monte Carlo, 81
- Markovian assumption, 59
- MCMC, 81
- MDE, 18
- measurement probability, 62
- metric representation, 54
- Metropolis, 81
- Metropolis-Hastings, 82
- middleware, 11
- mixture model, 72
- Model-driven engineering, 18
- Monte Carlo methods, 73

- occupancy grid, 54

- Parameter Definition Specific Language, 39
- particle filtering, 78
- Partitioned sampling, 89
- predicted belief distribution, 63
- probabilistic representation, 58
- proposal distribution, 73

- Rao-Blackwellized, 88
- RCControlPanel, 47
- RCInnerModelSimulator, 49
- RCLogger, 47
- RCManager, 45
- RCMonitor, 47
- RCReplay, 46
- reactive robot, 53

- rejection sampling, [73](#)
- relative representation, [57](#)
- Reversible-jump MCMC, [87](#)
- RJMCMC, [87](#)
- RoboComp DSL Editor, [34](#)

- S/IR, [77](#)
- sampling/importance resampling, [77](#)
- simulated annealing, [85](#)
- SLAM, [56](#)
- state transition probability function, [62](#)
- state variable, [59](#)
- Subspace hierarchical particle filtering, [90](#)
- swarm robotics, [53](#)
- symbolic representation, [56](#)

- target distribution, [73](#)
- topological representations, [55](#)

Bibliography

- [Angelopoulou et al., 1992] Angelopoulou, E., Hong, T. H., and Wu, A. Y. (1992). World Model Representations for Mobile Robots. In *Proc. of the Intelligent Vehicles Symposium*, pages 293–297.
- [Arkin, 1998] Arkin, R. C. (1998). *Behavior-based robotics*. MIT Press.
- [Bachiller et al., 2011] Bachiller, P., Bustos, P., and Manso, L. J. (2011). *Stereo Vision*, chapter Attentional Behaviors for Environment Modeling by a Mobile Robot, pages 17–40. InTech.
- [Badino et al., 2011] Badino, H., Huber, D., Park, Y., and Kanade, T. (2011). Fast and accurate computation of surface normals from range images. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3084–3091. IEEE.
- [Bandouch, 2010] Bandouch, J. (2010). *Observing and Interpreting Complex Human Activities in Everyday Environments*. PhD thesis, Technical University of Munich.
- [Bauer and Wilhelm, 2006] Bauer, J. and Wilhelm, R. (2006). Abstract Interpretation of Graph Grammars. In *Proc. of Simulation and Verification of Dynamic Systems Seminar*.
- [Besl and McKay, 1992] Besl, P. J. and McKay, N. D. (1992). Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics.
- [Bishop, 2007] Bishop, M. C. (2007). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition.
- [Bousassida et al., 2010] Bousassida, I., Chassor, C., and Jmaiel, M. (2010). Graph Grammar-based Transformation for Context-aware Architectures Supporting Group Communication. *Revue des Nouvelles Technologies de l'Information*, L(19):29–42.
- [Brandao et al., 2006] Brandao, B. C., Wainer, J., and Goldenstein, S. K. (2006). Subspace Hierarchical Particle Filter. In *Brazilian Symposium on Computer Graphics and Image Processing, SIBGRAPI '06*, pages 194–204.
- [Brooks et al., 2005] Brooks, A., Kaupp, T., Makarenko, A., Oreback, A., and Williams, S. B. (2005). Towards component-based robotics. In *Proceedings of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 163–168.
- [Brooks et al., 2007] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Oreback, A. (2007). Orca: A Component Model and Repository. *Software Engineering for Experimental Robotics*, Springer, pages 231–251.

- [Brooks, 1991] Brooks, R. A. (1991). Intelligence without representation. *Artificial intelligence*, 47(1):139–159.
- [Brooks et al., 1989] Brooks, R. A., Flynn, A. M., and Fast, C. (1989). Fast, cheap and out of control: A robot invasion of the solar system. *Journal of the British Interplanetary Society*, 42:478–485.
- [Brugali and Scandurra, 2009] Brugali, D. and Scandurra, P. (2009). Component-based robotic engineering. part i: Reusable building blocks. *IEEE Robotics and Automation Magazine*, 16(4):84–96.
- [Brugali and Shakhimardanov, 2010] Brugali, D. and Shakhimardanov, A. (2010). Component-based robotic engineering. part ii: Models and systems. *IEEE Robotics and Automation Magazine*, 17:100–112.
- [Bruyninckx, 2001] Bruyninckx, H. (2001). Open Robot Control Software: the OROCOS project. In *Proc. of International Conference on Intelligent Robots and Systems*, pages 2523–2528.
- [Busquets et al., 2003] Busquets, D., Sierra, C., and De Mántaras, R. L. (2003). A multiagent approach to qualitative landmark-based navigation. *Autonomous Robots*, 15(2):129–154.
- [Carlin and Louis, 2009] Carlin, B. P. and Louis, T. A. (2009). *Bayesian Methods for Data Analysis*. Chapman & Hall/CRC.
- [Carpin et al., 2007] Carpin, S., Lewis, M., Wang, J., Balakirsky, S., and Scrapper, C. (2007). Usarsim: a robot simulator for research and education. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1400–1405.
- [Cañas et al., 2007] Cañas, J. M., Ruiz-Ayucar, J., Agüero, C., and Martín, F. (2007). JDE-Neoc: Component oriented software architecture for robotics. *Journal of Physical Agents*, 1(1):1–6.
- [Cintas et al., 2011] Cintas, R., Manso, L. J., Pinero, L., Bachiller, P., and Bustos, P. (2011). Robust Behavior and Perception using Hierarchical State Machines: a Pallet Manipulation Experiment. *Journal of Physical Agents*, 5(1):35–44.
- [Cote et al., 2007] Cote, C., Brosseau, Y., Letourneau, D., Raievsky, C., and Michaud, F. (2007). Using MARIE for Mobile Robot Component Development and Integration. *Software Engineering for Experimental Robotics, Springer*, pages 211–230.
- [Cotterill et al., 2001] Cotterill, R. M. et al. (2001). Cooperation of the basal ganglia, cerebellum, sensory cerebrum and hippocampus: possible implications for cognition, consciousness, intelligence and creativity. *Progress in Neurobiology*, 64(1):1–33.
- [Coughlan and Yuille, 1999] Coughlan, J. M. and Yuille, A. L. (1999). Manhattan World: Compass Direction from a Single Image by Bayesian Inference. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 941–947. IEEE.

- [Del Pero et al., 2012] Del Pero, L., Bowdish, J., Fried, D., Kermgard, B., Hartley, E., and Barnard, K. (2012). Bayesian geometric modeling of indoor scenes. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2719–2726. IEEE.
- [Delage et al., 2006] Delage, E., Lee, H., and Ng, A. Y. (2006). A dynamic Bayesian network model for autonomous 3d reconstruction from a single indoor image. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 2418–2428. IEEE.
- [Descartes, 1641] Descartes, R. (1641). *Meditationes de prima philosophia, in qua dei existentia et animæ immortalitas demonstratur*.
- [Deutscher and Reid, 2005] Deutscher, J. and Reid, I. (2005). Articulated Body Motion Capture by Stochastic Search. *International Journal of Computer Vision*, 61(2):185–205.
- [Doucet et al., 2000] Doucet, A., De Freitas, N., Murphy, K., and Russell, S. (2000). Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 176–183. Morgan Kaufmann Publishers Inc.
- [Drews et al., 2010] Drews, P., Núñez, P., Rocha, R., Campos, M., and Dias, J. (2010). Novelty detection and 3d shape retrieval using superquadrics and multi-scale sampling for autonomous mobile robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3635–3640. IEEE.
- [Duda and Hart, 1972] Duda, R. O. and Hart, P. E. (1972). Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15:11–15.
- [Echeverria et al., 2011] Echeverria, G., Lassabe, N., Degroote, A., and Lemaignan, S. (2011). Modular open robots simulation engine: Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46–51.
- [Efron, 1982] Efron, B. (1982). The bootstrap, jackknife and other resampling plans. *Society of Industrial and Applied Mathematics*.
- [Fitzpatrick et al., 2008] Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards Long-Lived Robot Genes. *Journal of Robotics and Autonomous Systems*, 56(1):29–45.
- [Frith, 2009] Frith, C. (2009). *Making up the mind: How the brain creates our mental world*. Wiley-Blackwell.
- [Gaborski et al., 2004] Gaborski, R. S., Vaingankar, V. S., Chaoji, V., Teredesai, A., and Tentler, A. (2004). Venus: A system for novelty detection in video streams with learning. In *Proceedings of the 17th International FLAIRS Conference*, pages 1–5.
- [Gat, 1998] Gat, E. (1998). On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. MIT Press.
- [Gelfand and Smith, 1990] Gelfand, A. E. and Smith, A. F. M. (1990). Sampling-Based Approaches to Calculating Marginal Densities. *Journal of the American statistical association*, 85(410):398–409.

- [Gerkey et al., 2005] Gerkey, B., Collet, T., and MacDonald, B. (2005). Player 2.0: Toward a Practical Robot Programming Framework. In *Proc. of the Australasian Conf. on Robotics and Automation*.
- [Gerkey et al., 2003] Gerkey, B., Vaughan, R. T., and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, pages 317–323.
- [Ghallab et al., 1998] Ghallab, M., Aeronautiques, C., Isi, C. K., and Wilkins, D. (1998). PDDL: the planning domain definition language. Technical Report Technical Report CVC TR98003/DCS TR1165, New Haven, CT: Yale Center for Computational Vision and Control.
- [Gibson, 1986] Gibson, J. J. (1986). *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates.
- [Gordon et al., 1993] Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *Radar and Signal Processing, IEEE Proceedings*, volume 140, pages 107–113.
- [Green, 1995] Green, P. J. (1995). Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika*, 82.
- [Grisetti et al., 2006] Grisetti, G., Stachniss, C., and Burgard, W. (2006). Improved techniques for grid mapping with Rao-Blackwellized Particle Filters. *Robotics, IEEE Transactions on*, 23(1):34–46.
- [Gupta et al., 2010] Gupta, A., Efros, A. A., and Hebert, M. (2010). Blocks World Revisited: Image Understanding Using Qualitative Geometry and Mechanics. In *European Conference on Computer Vision(ECCV)*.
- [Gutiérrez et al., 2013] Gutiérrez, M. A., Romero-Garcés, A., Bustos, P., and Martínez, J. (2013). Progress in RoboComp. *Journal of Physical Agents*, 7(1):38–47.
- [H. and R., 2004] H., H. and R., M. (2004). Action selection and mental transformation based on a chain of forward models. In *Proceedings of the eighth international conference on simulation of behavior, SAB*, pages 213–222.
- [Han and Zhu, 2009] Han, F. and Zhu, S.-C. (2009). Bottom-Up/Top-Down Image Parsing with Attribute Grammar. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(1).
- [Hasemann, 1994] Hasemann, J.-M. (1994). A Robot Control Architecture based on Graph Grammars and Fuzzy Logic. In *Proc. of Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2123–2130.
- [Hastings, 1970] Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109.

- [He et al., 2005] He, J., Li, X., and Liu, Z. (2005). *Component-based Software Engineering: the Need to Link Methods and their Theories*, pages 70–95. Proc. of ICTAC 2005, Lecture Notes in Computer Science 3722. Springer.
- [Heckel, 2006] Heckel, R. (2006). Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198.
- [Henning and Spruiell, 2005] Henning, M. and Spruiell, M. (2005). Ice - Internet Communications Engine.
- [Henning and Spruiell, 2007] Henning, M. and Spruiell, M. (2007). Distributed Programming with Ice. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 89–110.
- [Hesslow, 2002] Hesslow, G. (2002). Conscious thought as simulation of behaviour and perception. *Trends in cognitive sciences*, 6(6):242–247.
- [Holland, 2003] Holland, O. (2003). Robots with internal models. *Journal of Consciousness Studies*, 10(4-5):77–109.
- [Holz et al., 2011] Holz, D., Schnabel, R., Droeschel, D., Stücker, J., and Behnke, S. (2011). Towards semantic scene analysis with time-of-flight cameras. In *RoboCup 2010: Robot Soccer World Cup XIV*, pages 121–132. Springer.
- [Hugues and Bredeche, 2006] Hugues, L. and Bredeche, N. (2006). Simbad: An autonomous robot simulation package for education and research. In Nolfi, S., Baldassarre, G., Calabretta, R., Hallam, J., Marocco, D., Meyer, J.-A., Miglino, O., and Parisi, D., editors, *From Animals to Animals 9*, volume 4095 of *Lecture Notes in Computer Science*, pages 831–842. Springer Berlin Heidelberg.
- [Johansen and Doucet, 2008] Johansen, A. M. and Doucet, A. (2008). A note on Auxiliary Particle Filters. *Statistics & Probability Letters*, 78(12):1498–1504.
- [Jung and Schramm, 2004] Jung, C. R. and Schramm, R. (2004). Rectangle detection based on a windowed hough transform. In *Proceedings of the XVII Brazilian Symposium on Computer Graphics and Image Processing*, pages 113–120.
- [Kalman, 1960] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME—Journal of Basic Engineering*, 82(Series D):35–45.
- [Khan et al., 2004] Khan, Z., Balch, T., and Dellaert, F. (2004). An mcmc-based particle filter for tracking multiple interacting targets. *Computer Vision-ECCV 2004*, pages 279–290.
- [Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.
- [Koenig and Howard, 2004] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154.

- [Kosaka and Kak, 1992] Kosaka, A. and Kak, A. (1992). Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. In *Intelligent Robots and Systems, 1992., Proceedings of the 1992 IEEE/RSJ International Conference on*, volume 3, pages 2177–2186.
- [Lagunovsky and Ablameyko, 1999] Lagunovsky, D. and Ablameyko, S. (1999). Straight-line-based primitive extraction in grey-scale object recognition. *Pattern Recognition Letters*, 20(10):1005–1014.
- [LaValle and Kuffner, 2001] LaValle, S. M. and Kuffner, J. J. (2001). *Algorithmic and Computational Robotics: New Directions*, chapter Rapidly-exploring Random Trees: Progress and Prospects, pages 293–308. A K Peters.
- [Leger, 1999] Leger, C. (1999). *Automated synthesis and Optimization of Robot Configurations: An Evolutionary Approach*. PhD thesis, The Robotics institute, Carnegie Mellon University.
- [Lin and Nevatia, 1998] Lin, C. and Nevatia, R. (1998). Building detection and description from a single intensity image. *Computer Vision and Image Understanding*, 72(2):101–121.
- [Lin et al., 2009] Lin, L., Wu, T., Porway, J., and Xu, Z. (2009). A Stochastic Graph Grammar for Compositional Object Representation and Recognition. *Journal of Pattern Recognition*, 42(7):1297–1307.
- [M. Markou and S. Singh, 2003a] M. Markou and S. Singh (2003a). Novelty detection: a review — part 1: statistical approaches. *Signal processing*, 83(12):2481–2497.
- [M. Markou and S. Singh, 2003b] M. Markou and S. Singh (2003b). Novelty detection: a review — part 2: neural network based approaches. *Signal processing*, 83(12):2499–2521.
- [MacCormick and Blake, 2000] MacCormick, J. and Blake, A. (2000). A probabilistic exclusion principle for tracking multiple objects. *International Journal of Computer Vision*, 39(1):57–71.
- [Manso, 2012] Manso, L. J. (2012). InnerModel*: InnerModel, InnerModelViewer, RCInnerModelEditor and RCInnerModelSimulator. RoboComp’s Wiki Tutorials.
- [Manso et al., 2010] Manso, L. J., Bachiller, P., Bustos, P., Núñez, P., Cintas, R., and Calderita, L. (2010). RoboComp: a Tool-based Robotics Framework. In *Proc. of Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*, pages 251–262.
- [Manso et al., 2012] Manso, L. J., Bustos, P., Bachiller, P., and Gutierrez, M. A. (2012). Graph Grammars for Active Perception. In *Proc. of 12th International Conference on Autonomous Robot Systems and Competitions*, pages 63–68.
- [Martínez and Caputo, 2011] Martínez, J. and Caputo, B. (2011). Towards Semi-Supervised Learning of Semantic Spatial Concepts. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 1936–1943.

- [Martínez et al., 2011] Martínez, J., Jimenez, A., Gómez, J. A., and García, I. (2011). Combining invariant features and localization techniques for visual place classification: successful experiences in the robotVision@ImageCLEF competition. *Journal of Physical Agents*, 5(1):45–54.
- [Matarić, 1990] Matarić, M. J. (1990). A Distributed Model for Mobile Robot Environment-Learning and Navigation. Technical report, Massachusetts Institute of Technology AI Lab.
- [Matas et al., 2000] Matas, J., Galambos, C., and Kittler, J. (2000). Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, 78(1):119–137.
- [Mateos et al., 2010] Mateos, J., Sánchez, A., Manso, L. J., Bachiller, P., and Bustos, P. (2010). RobEx: an open-hardware robotics platform. In *Workshop of Physical Agents*.
- [McIlroy, 1969] McIlroy, M. D. (1969). Mass produced software components. *NATO, Scientific Affairs Division*, pages 79–85.
- [Meng et al., 2006] Meng, Y., Nickerson, V. J., and Gan, J. (2006). Multi-robot aggregation strategies with limited communication. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2691–2696. IEEE.
- [Metropolis et al., 1953] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., and Teller, A. H. (1953). Equations of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21:1087–1091.
- [Miller et al., 1986] Miller, G. A., Galanter, E., and Pribram, K. H. (1986). *Plans and the Structure of Behavior*. Adams Bannister Cox.
- [Montemerlo et al., 2003] Montemerlo, M., Roy, N., and Thrun, S. (2003). Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In *Proceedings of International the International Conference on Intelligent Robots and Systems*, pages 2436–2441.
- [Montijano and Sagues, 2009] Montijano, E. and Sagues, C. (2009). Topological maps based on graphs of planar regions. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1661–1666.
- [Murtra et al., 2010] Murtra, A. C., Trulls, E., Tur, J. M. M., and Sanfeliu, A. (2010). Efficient use of 3d environment models for mobile robot simulation and localization. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 461–472. Springer.
- [Müller and Rodriguez, 1996] Müller, J. P. and Rodriguez, M. O. (1996). A constructivist approach to autonomous systems. In *In Growing Mind Symposium, Piaget*.
- [NAIST, 2010] NAIST (2010). RT-Middleware: OpenRTM-AIST. National Institute of Advanced Industrial Science and Technology (AIST).
- [Neal, 2001] Neal, R. M. (2001). Annealed importance sampling. *Statistics and Computing*, 11(2):125–139.

- [Newman, 2006] Newman, P. (2006). Moos - mission orientated operating suite. Massachusetts Institute of Technology, Dept. of Ocean Engineering.
- [Newton, M. A. and Raftery, A. E., 1994] Newton, M. A. and Raftery, A. E. (1994). Approximate Bayesian inference with the weighted likelihood bootstrap. *Journal of the Royal Statistical Society*, pages 3–48.
- [Nicolescu and Matarić, 2002] Nicolescu, M. N. and Matarić, M. J. (2002). A Hierarchical Architecture for Behavior-Based Robots. In *Proc. of Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, pages 227–233.
- [Noë, 2004] Noë, A. (2004). *Action in perception*. The MIT Press.
- [Nuske et al., 2009] Nuske, S., Roberts, J., and Wyeth, G. (2009). Robust outdoor visual localization using a three-dimensional-edge map. *Journal of Field Robotics*, 26(9):728–756.
- [Palmer et al., 1994] Palmer, P. L., Kittler, J., and Petrou, M. (1994). Using focus of attention with the hough transform for accurate line parameter estimation. *Pattern Recognition*, 27(9):1127–1134.
- [Pitt and Shephard, 1999] Pitt, M. K. and Shephard, N. (1999). Filtering via simulation: Auxiliary particle filters. *Journal of the American statistical association*, 94(446):590–599.
- [Quigley et al., 2009] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *Proc. of ICRA Workshop on Open Source Software*.
- [Quintero et al., 2011] Quintero, E., García-Olaya, A., Borrajo, D., and Fernández, F. (2011). Control of Autonomous Mobile Robots with Automated Planning. *Journal of Physical Agents*, 5(1):3–13.
- [Richtsfeld et al., 2012] Richtsfeld, A., Mörwald, T., Prankl, J., Balzer, J., Zillich, M., and Vincze, M. (2012). Towards scene understanding—object segmentation using rgbd-images. In *Computer Vision Winter Workshop (CVWW)*, volume 1.
- [Roberts, 1963] Roberts, L. G. (1963). *Machine Perception of Three-dimensional Solids*. PhD thesis, Massachusetts Institute of Technology.
- [Rodriguez et al., 2008] Rodriguez, S., S. Thomas, R. P., and Amato, N. (2008). RESAMPL: A region-sensitive adaptive motion planner. *Algorithmic Foundation of Robotics VII*, pages 285–300.
- [Romero-Garces et al., 2011] Romero-Garces, A., Manso, L. J., Gutiérrez, M. A., Cintas, R., and Bustos, P. (2011). Improving the life cycle of robotics components using Domain Specific Languages. In *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob’2011)*.
- [Rosenfeld, 1969] Rosenfeld, A. (1969). Picture processing by computer. *ACM Comput. Surv.*, 1:147–176.

- [Rubin et al., 1988] Rubin, D. B. et al. (1988). Using the SIR algorithm to simulate posterior distributions. *Bayesian statistics*, 3:395–402.
- [Rusu, 2009] Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technical University of Munich, Germany.
- [Rusu et al., 2009] Rusu, R. B., Blodow, N., Marton, Z. C., and Beetz, M. (2009). Close-range scene segmentation and reconstruction of 3d point cloud maps for mobile manipulation in domestic environments. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1–6.
- [Sanchez et al., 2012] Sanchez, A., Núñez, P., Manso, L. J., and Bustos, P. (2012). Experiments in self-calibration of an autonomous mobile manipulator. In *Proceedings of the Workshop of Physical Agents (WAF2012)*, pages 183–190.
- [ScGutierrezhlegel et al., 2010] ScGutierrezhlegel, C., Steck, A., Brugali, D., and Knoll, A. (2010). Design abstraction and processes in robotics: From code-driven to model-driven engineering. In *2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots*, pages 324–335. Springer.
- [Schlegel, 2004] Schlegel, C. (2004). *Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach*. PhD thesis, University of Ulm.
- [Schlegel, 2006] Schlegel, C. (2006). Communication Patterns as Key Towards Component-Based Robotics. *International Journal of Advanced Robotic Systems*, 3(1):49–54.
- [Schmidt et al., 1997] Schmidt, D. C., Gokhale, A., Harrison, T. H., Levine, D., and Cleeland, C. (1997). Tao: a high-performance end system architecture for real-time corba. *IEEE Communications Magazine*, 14(2).
- [Schmidt and Huston, 2002] Schmidt, D. C. and Huston, S. (2002). *C++ Network Programming Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley.
- [Septier et al., 2009] Septier, F., Pang, S. K., Carmi, A., and Godsill, S. (2009). On mcmc-based particle methods for bayesian filtering: Application to multitarget tracking. In *Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2009 3rd IEEE International Workshop on*, pages 360–363. IEEE.
- [Sheng et al., 2006] Sheng, W., Yang, Q., Tan, J., and Xi, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, 54(12):945–955.
- [Simhon and Dudek, 1998] Simhon, S. and Dudek, G. (1998). A Global Topological Map formed by Local Metric Maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1708–1714.
- [Siskind et al., 2007] Siskind, J. M., Sherman, J., Pollak, I., Harper, M. P., and Bouman, C. A. (2007). Spatial Random Tree Grammars for Modeling Hierarchical Structure in Images with Regions of Arbitrary Shape. *Transactions on Pattern Analysis and Machine Intelligence*, 29(9):1504–1519.

- [Smith and Gelfand, 1992] Smith, A. F. M. and Gelfand, A. E. (1992). Bayesian statistics without tears: a sampling-resampling perspective. *American Statistician*, pages 84–88.
- [Smith et al., 2009] Smith, B., Howard, A., McNew, J.-M., Wang, J., and Egerstedt, M. (2009). Multi-robot Deployment and Coordination with Embedded Graph Grammars. *Journal of Autonomous Robots*, 26(1):77–98.
- [Smith, 2007] Smith, K. (2007). *Bayesian methods for visual multi-object tracking with applications to human activity recognition*. PhD thesis, Lausanne, EPFL.
- [Svensson et al., 2009] Svensson, H., Morse, A., and Ziemke, T. (2009). *Neural Pathways of Embodied Simulation*, pages 95–114. Anticipatory Behavior in Adaptive Learning Systems. Springer.
- [Tao et al., 2002] Tao, W. B., Tian, J. W., and Liu, J. (2002). A new approach to extract rectangular building from aerial urban images. In *Signal Processing, 2002 6th International Conference on*, volume 1, pages 143–146.
- [Thrun, 1998] Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 9(1):21–71.
- [Thrun, 2000] Thrun, S. (2000). Monte carlo pomdps. *Advances in neural information processing systems*, 12:1064–1070.
- [Thrun, 2002] Thrun, S. (2002). Robotic Mapping: A Survey. Technical report, School of Computer Science, Carnegie Mellon University.
- [Thrun et al., 2005] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics*. MIT Press, Cambridge.
- [Tikhanoff et al., 2008] Tikhanoff, V., Cangelosi, A., Fitzpatrick, P., Metta, G., Natale, L., and Nori, F. (2008). An open-source simulator for cognitive robotics research: The prototype of the icub humanoid robot simulator.
- [Tomatis et al., 2003] Tomatis, N., Nourbakhsh, I., and Siegwart, R. (2003). Hybrid simultaneous localization and map building: a natural integration of topological and metric. *Robotics and Autonomous Systems*, 44(1):3–14.
- [Torralba et al., 2010] Torralba, A., Murphy, K. P., and Freeman, W. T. (2010). Using the forest to see the trees: exploiting context for visual object detection and localization. *Communications of the ACM*, 53(3):107–114.
- [Tu et al., 2005] Tu, Z., Chen, X., Yuille, A., and Zhu, S. C. (2005). Image parsing: Unifying segmentation, detection, and recognition. *International Journal of Computer Vision*, 63(2):113–140.
- [Tu and Zhu, 2002] Tu, Z. and Zhu, S.-C. (2002). Image Segmentation by Data-Driven Markov Chain Monte Carlo. *Transactions on Pattern Analysis and Machine Intelligence*, 24.

- [Ulam and Arkin, 2004] Ulam, P. and Arkin, R. C. (2004). When good communication go bad: communications recovery for multi-robot teams. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3727–3734. IEEE.
- [Utz et al., 2007] Utz, H., Mayer, G., Kaufmann, U., and Kraetzschmar, G. (2007). VIP: The Video Image Processing Framework Based on the MIRO Middleware. *Software Engineering for Experimental Robotics, Springer*, pages 325–344.
- [van Zwynsvoorde et al., 2000] van Zwynsvoorde, D., Simeon, T., and Alami, R. (2000). Incremental topological modeling using local Voronoï-like graphs. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and System (IROS 2000)*, volume 2, pages 897–902.
- [Vaughan et al., 2003] Vaughan, R. T., Gerkey, B., and Howard, A. (2003). On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems*, pages 2121–2427.
- [Vlassis et al., 2002] Vlassis, N., Terwijn, B., and Krose, B. (2002). Auxiliary particle filter robot localization from high-dimensional sensor observations. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 7–12. IEEE.
- [Volpe et al., 2001] Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The CLARAty Architecture for Robotic Autonomy. In *Proc. of Aerospace Conference*, pages 121–132.
- [Welch and Bishop, 1995] Welch, G. and Bishop, G. (1995). An introduction to the kalman filter. Technical Report 041, University of North Carolina at Chapel Hill, Chapel Hill, NC.
- [Wolf and Sukhatme, 2008] Wolf, D. F. and Sukhatme, G. S. (2008). Semantic mapping using mobile robots. *IEEE Transactions on Robotics*, 24(2):245–258.
- [Yan et al., 2006] Yan, F., Zhuang, Y., and Wang, W. (2006). Large-scale Topological Environmental Model Based Particle Filters for Mobile Robot Indoor Localization. In *Proc. of the IEEE Int. Conf. on Robotics and Biomimetics*, volume 0, pages 858–863.
- [ZeroC, 2010] ZeroC (2010). ZeroC Customers.
- [Zhu et al., 2000] Zhu, S.-C., Zhang, R., and Tu, Z. (2000). Integrating Bottom-Up/Top-Down for Object Recognition by Data Driven Markov Chain Monte Carlo. In *Proc. of Int. Conf. in Computer Vision and Pattern Recognition*, pages 738–745.
- [Zhu et al., 2003] Zhu, Y., Carragher, B., Mouche, F., and Potter, C. S. (2003). Automatic Particle Detection through Efficient Hough Transforms. *IEEE Trans. Med. Imaging*, 22(9).
- [Ziemke et al., 2005] Ziemke, T., Jirnhed, D.-A., and Hesslow, G. (2005). Internal simulation of perception: a minimal neuro-robotic model. *Neurocomputing*, 68:85–104.