ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
ELECTRÒNICA I INFORMÀTICA LA SALLE

PROJECTE FI DE CARRERA

ENGINYERIA EN INFORMÀTICA

Statecharts modelling of a robot's behavior

ALUMNE

Silvia Mur Blanch

PROFESSOR PONENT

Maria Antònia Mozota Coloma

ACTA DE L'EXAMEN
DEL PROJECTE FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. Silvia Mur Blanch

va exposar el seu Projecte de Fi de Carrera, el qual va tractar sobre el tema següent:

Statecharts modelling of a robot's behavior

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Projecte amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

«Try not.
Do. Or do not.
There is no try.»

In loving memory of my father,
Josep Mur Tolrà.

Acknowledgments

To Hans Vangheluwe, for giving me the subject of my Master Thesis and the chance to do it under his supervision at the Modelling, Design and Simulation Lab at the School of Computer Science of McGill University. Thanks for being a constant inspiration and a fascinating speaker, for your ideas, your patience, and your support. It still amazes me how you know so much about everything, and I hope that I will eventually find a job that I can be as passionate about as you are about yours!

To Maria Antònia Mozota, the best teacher I had at La Salle Universitat Ramón Llull, for accepting to be my Master Thesis supervisor. Thanks for being critic and sincere with me, and for pushing me to improve my work.

To my mother and sister and the rest of my family for their unconditional support.

To Reehan, my MSDL mate and mentor, who taught me pretty much all I know about Statecharts, Petri Nets, LaTeX, McGill University, Montréal, eating places, and many more useful things. I was blessed to meet you, and I would have been lost without you!

To Yanwar, very especially, for being so kind as to buy the robot for me and send it to Spain. I don't think I'll ever be able to repay that favor in full, my friend.

To my other McGill mates Javier, Anna, Kyle, Xiaoxi, Alex, Charles, Riry, Massi, and my flatmates Jess, Samuel, and Anders, for being my small family in Montréal.

To Paty, Chep, Marcel, Josep, Txema, and Miquel, my friends, classmates, and project mates, without whom I would never have made it through my Master courses. I owe it all to you guys, and I miss our "company meetings" but, most of all, our "coffee breaks".

To José Julio, Ana, Gabi, Emili, Javi, Albert, Sergio, Erika, Miriam, Christian, David R., Isra, Marc C., Marc F., and many other colleagues, who have eventually helped me improve my programming skills and enjoy my working hours... and non-working hours too.

To Raül, who helped me make it into McGill University as a Visiting Student.

To Marta, David G., Marc T., Sandra, Albert S., and Nadia, just because.

To David Vernet, Xavi Canaleta, and Joan Camps, my friends and teachers, from whom I've learnt a lot, and not only about programming.

And, finally, to Marta B. and Montse for putting pressure on me every Wednesday, and pushing me to finish writing my Master Thesis.

Thanks to you all!

Abstract

Statecharts are a visual modelling formalism especially suited for the specification and design of complex reactive systems. This formalism is studied and used in the modelling of a computer-controlled game character behavior, precisely the artificial intelligence of a tank. As well, a statecharts modelled simulation environment for testing is developed. The statecharts modelled artificial intelligence (AI) is later simplified and adapted to be used in the modelling of a robot's behavior. A high level Python interface is used to control the robot by means of Bluetooth serial port communication. Finally, a diverse set of statecharts modelled behaviors is developed along with different Python applications, including a graphical interface remote control for the robot.

Summary

The present Master Thesis was developed at the Modelling, Design and Simulation Lab [1] (MSDL) at the School of Computer Science of McGill University, in Montréal, Canada, in the course of a 7 month period (from January to July 2008), and under the supervision of Professor Doctor Hans Vangheluwe.

Being a research project, the Master Thesis wasn't tied any predefined development requirements nor a very meticulous and strict time planning. The main study subject of this research project were David Harel's *Statecharts*, an event-based visual modelling formalism for describing complex reactive systems. Statecharts are an enhancement of finite state automata, and provide features such as state clustering, orthogonality, and history. We've specifically focused on the statecharts modelling of AIs.

Video game AIs in particular have exponentially grown in complexity over the past few years, but writing consistent, re-usable and efficient AI code has become hard. The starting point of this research project was the proposal that modelling video game AI at an appropriate abstraction level using an appropriate modelling language/formalism has many advantages. The modelling formalism chosen to demonstrate our approach was statecharts.

In order to learn about visual modelling formalisms in general and statecharts in particular, we attended course COMP 763B: Modelling and Simulation Based Design, taught by Prof. Doctor Vangheluwe during the Winter semester at the School of Computer Science of McGill University. We did the class assignments and the personal course project, the subject of which was *Statecharts modelling of TankWars*.

The first part of this Master Thesis is focused on the statecharts modelling of computer-controlled game character behaviors. An AI is designed and split into several components at different levels of abstraction, each of which is modelled in statecharts with AToM^3, a custom modelling tool developed at the MSDL, in close collaboration with the Universidad Autónoma de Madrid (UAM). The statecharts models are later compiled into Python. A simple TankWars simulation application is developed in Python. We use a *controller* class to link the AI components with each other and with the GUI.

In the second part of this Master Thesis we introduce MSDL's robot: iRobot Create, a programmable robot that's based on the world famous vacuuming robot, the Roomba. Create is small, flat, and very sturdy, it has over 30 built-in sensors and provides the possibility of attaching all sorts of electronics or hardware peripherals thanks to its Command Module. The previous statecharts modelled AI components are used as the base to develop statecharts modelled behaviors for the robot. A wireless Bluetooth serial port connection is used to communicate with the robot, as well as a high-level Python interface called *PyRobot*. A series of Python applications are developed combining statecharts and *PyRobot*, such as a virtual iRobot Create remote control. Ultimately, we achieve to model an autonomous behavior for the robot, through the use of its built-in sensors.

Contents

List of Figures

1 Introduction

In this introductory chapter we will expound the motivations behind this Master Thesis, as well as the main objectives that we aim to achieve with it. Also, we will discuss other aspects that had to be considered prior to the execution of the Master Thesis, such as the selection of a programming language, a modelling formalism, the tools required to develop our practical work, and a programmable robot. A brief study of all our options is provided in each case, along with a justification to each of our choices.

1.1 Motivation

On a personal level, the main motivation behind this Master Thesis is the opportunity to do it abroad. After very intense and difficult years of going to classes, working on numerous projects, and taking exams, the Master Thesis is the last step left, so it seems like the best and only chance to live a vital experience before diving into the working life. It's the perfect opportunity to get to know different work methodologies, explore unknown fields in the world of computer science and, maybe, discover a vocation. Doing the Master Thesis at a foreign university will definitely provide the chance of meeting many brilliant and interesting people, such as teachers and students from all over the world.

We consider that a Master Thesis should consist of a theoretical research component and a practical development component. It was important that the main subject of the Master Thesis was something that we had never seen before, but at the same time it has to be something which foundations were known to us. Therefore, as a necessary part of the overall experience, I conceded that the subject of this project should be chosen by its supervisor at the School of Computer Science of McGill University: Prof. Dr. Hans Vangheluwe, director of the Modelling, Simulation and Design Lab (MSDL) [1]. Prof. Dr. Vangheluwe kindly invited me to join his lab as a visiting student for a period of 6 months; I will as well attend a course on modelling and simulation based design, in order to be able to work on a project related to game character behavior modelling.

The motivation behind this project as expressed by its conceiver, Prof. Dr. Vangheluwe, is to program a robot's behavior by using a modelling formalism, desirably with the final purpose of bringing an indefinite number of robots face to face in a Robot Wars competition. The main idea derives from the paper *Model-based design of Computer-controlled game character behavior* [2], in which Professor Vangheluwe himself and other authors proposed a way of easily developing a game character's AI through the use of modelling formalisms; specifically, statecharts. The authors used the EA Tank Wars [3] game environment to demonstrate their approach by modelling a tank pilot's AI.

The AI architecture was divided into several layers at different levels of abstraction, with several

components per layer as well. The components were originally designed using the class diagrams formalism, were every component consists of a set of attributes or variables, a set of methods and a statecharts defined behavior. The components are supposed to be orthogonal, thus being able to interact with each other through event broadcasting. The class diagram models were later developed using AToM^3 [4] but, by the time the *Model-based design of Computer-controlled game character behavior* paper was written, AToM^3's statecharts compiler for class diagrams formalism did not support orthogonality, nor composite or history states and, therefore, the final models had to be simplified and ended up barely resembling the original designs at all. These models were developed with the sole purpose of illustrating the paper's approach with a working example of a statecharts modelled tank pilot's AI, which was successfully inserted into the EA Tank Wars main game loop. In that occasion, the class diagrams were compiled into C++.

However, the main motivation behind the paper was not to demonstrate that it was possible to run a statecharts modelled AI into the EA Tank Wars environment, but to demonstrate that AI's could be statecharts modelled. Therefore, the paper's approach is taken up again with the purpose of focusing on exploiting statecharts's features as much as possible. We have decided not to use the EA Tank Wars environment anymore, so we are no longer attached to any programming language in particular. The class diagrams formalism is ruled out in favor of the statecharts formalism, mainly because AToM^3's StateCharts Compiler (SCC) [5] does support all of statecharts's features. The AI components will be modelled according to the original designs and later compiled into Python. A simple Python's Tkinter environment is to be developed in order to test the compiled statecharts components.

Ultimately, our tank pilot's modelled AI can be adapted, with considerably slight modifications, to match any other character AI in games of the same or similar genres. As well, our AI architecture and components can be simplified and adapted to be loaded into a robot, with the final purpose of having two or several robots "battling" each other in a Robot Wars competition. Whereas robot AI programming is neither an original nor a new concept, robot AI modelling, as approached by this project, is quite an innovation. The advantage of modelling, as opposed to programming, is that it is done at a higher level of abstraction and, therefore, the modeller does not need to know any programming languages or even have programming notions at all; instead, the modeller can focus on describing the logic of the model in a way that very much resembles natural language, by means of a state/event based formalism such as statecharts.

During the first meeting with Prof. Vangheluwe, the project time line has been outlined and the main milestones have been defined. Also, I've been briefly introduced to the statecharts modelling formalism, Prof. Vangheluwe's model-based design approach to the EA Tank Wars competition, and the lab's robot, iRobot Create. It's particularly motivating to face the challenge of this project at this point, since modelling formalisms, AI programming, and robots are all unknown subjects to me, which seems to make a perfect combination for an interesting and complete Master Thesis.

1.2 Objectives

If we focus rigorously on the academic purpose of this project, there are three main objectives to it. However, it seems fair to admit that these objectives have not been established by us, but our supervisor, Prof. Dr. Hans Vangheluwe. And even though we've made those objectives ours, we should as well point out which are the personal objectives behind the execution of this Master Thesis.

1.2.1 Project objectives

These are numbered in order of importance and execution.

1. *Develop further work in the statecharts modelling of Tank Wars.* The theoretical research work behind the paper *Model-based design of Computer-controlled game character behavior*, presented at the *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems* (MoDELS), provides a very detailed definition of a layered artificial intelligence, more specifically, that of a computer-controller tank. In order to illustrate their approach with the implementation of a practical example, the paper's authors wisely chose to use an existing environment: EA Tank Wars, which is a framework provided by video games company Electronic Arts for their homonym student AI competition. In this competition, each contestant must program and submit an AI, with absolute freedom of choice of the programming paradigm and language. If the authors of [2] could insert their code -synthesized from the statecharts model/s of their AI's components- into the EA Tank Wars game main loop, they would successfully demonstrate their approach. However, due to the limitations of the class diagrams compiler of AToM^3, which is the modelling tool used by the authors of [2], they could not fully put their theory into practice. That's why, before even considering further objectives for this Master Thesis, Prof. Dr. Vangheluwe's purpose is to really prove that AI modelling is possible, as proposed by him and his colleagues in their paper, with a proper practical example. We will rule out the use of any framework and focus on faithfully capturing the logic of the statecharts models instead, by developing a simple simulation application.
2. *Model a robot's behavior using statecharts.* Artificial intelligence design has experienced an exponential improvement over the past decade, almost exclusively thanks to the huge growth of the video game industry. Nowadays, video games and AIs are two closely related concepts: the first can not be conceived separately from the second. The rapid and constant development of computer systems's speed and calculation capacity has obviously played a very important role in the achievement of many goals in the AI field. But of course AIs have a lot more applications other than video games. Whereas the purpose of our first objective is to develop a statecharts modelled game character behavior, our intention with the second objective is to go a bit further and dive into a more tangible application of AI modelling. We will adapt the statecharts model/s of the game character's AI, and use them to develop a statecharts modelled behavior for the MSDL's own iRobot Create robot [6]. We should be able to get the robot to behave completely autonomously while showing signs of its so-called artificial intelligence by properly reacting to external events.
3. *Create "Robot wars" as the result of combining objectives 1 and 2.* We are insecure about the real chances of achieving this objective, not only because we doubt our capacity to implement such a complex hardware project but, first and foremost, because of the very limited time we have to develop this Master Thesis. Although we consider this particular objective to be definitely out of the scope of this Master Thesis's aspirations, and this will become clearer as the project develops, "Robot wars" is actually the real goal we aim to achieve eventually, not at a personal level, but as members of the Modelling, Simulation and Design Lab at the School of Computer Science of McGill University. So we might not be the ones attaining this goal but, in the end, someone else at the MSDL will, and that's what really matters.

1.2.2 Personal objectives

During our years as Computer Science Engineering students, we had been constantly taught by our teachers that the most valuable quality of an engineer is the ability to find solutions and apply them; the ability to face a problem for the first time and be able to break it down in a set of smaller problems, later analyzing each of them and searching for similarities with previously solved problems. An engineer is expected to be creative and capable of finding a way through in an unknown environment. Everything that seems to go against the engineer, just adds to the challenge of solving the problem.

With the execution of this Master Thesis I aim to prove myself that I am a worthy engineer, capable of facing challenges and overcoming them by my own means. I could as well have chosen a simpler subject, or one that I could develop and finish in a much shorter period of time... but there was no challenge in that. I chose to take the chance of doing a Master Thesis while enjoying the experience of a lifetime. My personal objectives go beyond the purely academic:

1. *Internationalize.* The number one objective is to success abroad; I wanted to do my Master Thesis at a foreign university, but not just any foreign university, Europe wasn't far enough for me. I was determined to spend some months doing an internship at a North American university, as a visiting student. What are the graduate courses and the course assignments like? What kind of courses are taught and how are they taught? I want to know all these things, and I want to meet computer engineers from all over the world. McGill University provides a very international study environment, there are students from virtually every country in the world. I'm looking forward to exchange knowledge, experiences, and points of view with people of very different cultures.
2. *Do a research project.* Most North American universities are devoted to research, whereas most Spanish universities, or Enginyeria i Arquitectura La Salle in particular, are not. When it comes to Master Theses, most graduate students choose to develop practical projects with very little research components. This is mainly because doing research requires a much longer development time, but also because our university's research departments can't provide every graduate student with an interesting research project. I wanted my Master Thesis to be mainly a research project, involving little practical development, and I wanted to do research on a subject that was completely unknown to me, so that I would have to start from scratch.
3. *Write the Master Thesis report in English.* Even though this didn't seem as difficult a challenge at the beginning, I've just realized it is probably going to be the toughest of all the challenges I will face during the execution of this Master Thesis. Not only must I write the report in English, I must use a fluent, rich, and technical vocabulary. And, what's more, the report is going to be entirely written using LaTeX.

So, now that my personal objectives have been put forward, why McGill University? Why the Modelling, Simulation and Design Lab? McGill University It is among the top 12 universities in the world, and the top research university in Canada. My stay at the MSDL as a visiting student has been planned one year in advance. Before Prof. Dr. Hans Vangheluwe invited me to become a member of the MSDL, I had applied for several universities of North America, being McGill my number one option. Besides, I knew Montréal and, disregarding its extremely cold Winter, I consider it a perfect city to live in. But the fact that made me prefer McGill over the other universities in the first place was the *gr@m*, the Games Research Lab [7] at the School of Computer Science of McGill University. Video games are the reason why I decided

to study Computer Science Engineering, and I expect to work in that field some day. I became very interested about the *gr@m* so I contacted Prof. Dr. Vangheluwe, who is one of the faculty members associated with the lab, in order to apply for a visiting student internship there. Prof. Dr. Vangheluwe offered me to become a member of the MSDL instead of the *gr@m*, and as soon as I accepted he sent me a formal invitation. We met briefly in Barcelona some months later, in August 2007, when he was attending a congress in the city. He introduced me to the projects that had already been developed at the MSDL, the ones that were currently under development, and the project that I would be working on when I joined the lab 6 months later. As we said goodbye, Prof. Dr. Vangheluwe mentioned -referring to the temperature- "The next time we meet there will be as many degrees, only below zero".

1.3 Options and decision-making

1.3.1 Choosing a modelling formalism

Prior to begin working on this Master Thesis, we didn't know what modelling formalisms were. In fact, we did know at least one modelling formalism: Finite State Automata, but we did not know that they were considered so. During the first five project development months, a course on modelling formalisms was attended: COMP 763B, Modelling and Simulation Based Design [8], given by Prof. Dr. Hans Vangheluwe at the School of Computer Science of McGill university. In this course, we learnt mainly about Petri Nets, Statecharts, DEVS (Discrete Event System Specification), meta-modelling and model transformation.

According to the characteristics of the environment that we were initially going to model for (Tank Wars), we were looking for a modelling formalism that satisfied a series of features:

- *State/event based.*
- *Autonomous/reactive behavior.* Our components must react to both internal and external events. Events trigger transitions, and transitions cause our components to change their current states. The only way that components of the same system can interact with each other is through event broadcasting.
- *Modularity.* It's tantamount to easy assembly, flexible arrangement, and reusability. The formalism we choose must provide us with the possibility of modular modelling.
- *Notion of time.* Seconds, minutes, and hours must keep their meaning in the modelling formalism. We might be interested in using the passing of time to trigger transitions in our models.
- *Well known.* There should be a wide availability of educational material and simulators/code generators for our modelling formalism.

Both Petri Nets and Statecharts seemed quite appropriate for modelling reactive systems, but only statecharts met all our requirements, and as well provide:

- *Composition (or Clustering)* introduces the XOR (exclusive-or) decomposition of states, which captures the property that, being in a state, the system must be in only one of its composite components.
- *Orthogonality (or Concurrency)* introduces the AND decomposition of states, which captures the property that, being in a state, the system must be in all of its orthogonal components.

- *History.* In statecharts syntax, entering the history state is tantamount to entering the most recently visited state.

However, we must confess, there was no possible choice of a modelling formalism because we knew from the very beginning that we were going to use Statecharts, since we were conditioned by the work previously done by Prof. Dr. Vangheluwe et. al. for the paper *Model-based design of Computer-controlled game character behavior.*

1.3.2 Choosing a modelling tool

Very commonly in the context of Computer Science projects, the choice of a formalism or programming language is closely linked to the choice of a programming platform, framework or compiler. In a similar way, we could have been conditioned in our choice of a modelling formalism by the modelling tools availability, but it actually was quite the opposite.

The Modelling, Simulation and Design Lab's pride, its most popular and celebrated project is AToM^3, A Tool for Multi-formalism and Meta-Modelling. It supports multiple formalisms such as Entity-Relationship, GPSS, Deterministic Finite state Automata, Non-Deterministic Finite state Automata, Petri Nets, Data Flow Diagrams and Statecharts. It also provides code generation and generation of executable simulators based on the operational semantics of formalisms, as well as many other features. AToM^3 is freeware, and its use is widespread all over the many labs of the School of Computer Science of McGill, other than MSDL. So, in a way, the use of AToM^3 for this project was a must. Nonetheless, these are some of the tools that we could have used instead:

- *VMTS* (Visual Modelling and Transformation System) [9] is a modelling tool developed by Department of Automation and Applied Informatics of the Budapest University of Technology and Economics. It's developed in CSharp language, and therefore Microsoft .NET Framework is required for its use. It's also mandatory to install SQLServer Express, which can be annoying for some users. It supports multiple modelling formalisms, as well as meta-modelling and model transformation.
- *Telelogic STATEMATE* [10], or *STATEMATE system*, is a tool that David Harel, the creator of the statecharts formalism himself, helped design and build at I-Logix, Inc., and it's now distributed by IBM. In addition to supporting the modelling effort using statecharts, it provides powerful tools for inspecting and analyzing the resulting models, via model execution, dynamic testing, and code synthesis. STATEMATE statecharts semantics was specifically designed for it (for further information see [11]).
- *Telelogic RHAPSODY* [12], also developed by David Harel et. al., and as well distributed by IBM, is a UML based software development tool, designed to support complete model-based iterative life-cycle. RHAPSODY statecharts semantics was specifically designed for it as well (for further information see [13]).
- *AnyLogic* by XJ Technologies boasts about being "the only tool that supports all the most common simulation methodologies in place today: System Dynamics, Process-centric (AKA Discrete Event), and Agent Based modelling". The object-oriented model design paradigm supported by *AnyLogic* provides for modular, hierarchical, and incremental construction of large models. *AnyLogic* is developed in Java and based on the Eclipse framework, which makes it 100% multi-platform.

AToM^3's user interface is far from eye-catching; it's easy to use but not intuitive at all; its possibilities are almost countless but, since there is no complete nor accurate documentation

for AToM^3, there's no way of knowing them; the undo command exists but it doesn't work, which too frequently means that the only way of correcting our modelling errors requires redoing the whole model from scratch; besides all this, as if it wasn't already bad enough, AToM^3 tends to crash easily. On the good side, AToM^3 only requires that Python be installed on the computer; it's a powerful, robust, and very useful tool; it works fast and it takes up very little resources. We definitely consider that we made a good choice because anyhow, in Prof. Dr. Vangheluwe's own words, *"AToM^3 crashes as much as any other modelling tool... but it's free"*.

1.3.3 Choosing a programming language

As for the programming language, the first option we considered was C++, but only because the Tank Wars environment provided by EA is programmed in C++. Since we very early ruled out the use of the Tank Wars environment in pursuit of our own much simpler environment, we also ruled out the use of C++ and chose to use Python instead. The use of Python is widespread among the School of Computer Science of McGill University, not only for research projects but class projects as well. Python is a high level programming language that supports multiple paradigms: object-oriented, imperative, and functional; its syntax is easy to learn, which emphasizes readability and reduces the cost of program maintenance. As well, Python supports modules and packages, which encourages program modularity and code reuse. Python language is interpreted instead of compiled; since there is no compilation step, the edit-test-debug cycle is incredibly fast, therefore increasing the programmer's productivity.

The choice of the programming language was also linked with the choice of the modelling tool we were going to use, and our choice was AToM^3, MSDL's modelling tool. One of AToM^3's features is SCC, a StateCharts Compiler that generates efficient source code from textural description of models. SCC's supported output languages are Java, Python, C++, and CSharp, so these were our four possible choices:

- *C++*. This programming language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates, and exception handling among other features. It is a statically typed, free-form, multi-paradigm, compiled language. Even though our programmer roots are with C++, and we're very confident in the use of this programming language, we were well aware of its lacks and the inconveniences of choosing it. Among other reasons that made us rule out C++ as our programming language, we considered the fact that it did not provide us with graphic modules for easily developing simple, user-friendly interfaces. As well, it's usually difficult to find external modules or libraries developed by others on the internet. Besides, writing low level operations in C++ is quite complicated.
- *Java*. Sun Microsystem's Java is an object-oriented, imperative programming language that derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. Soon after its release, Java became very popular for web enhancement through the use of *applets*. Apparently, Java was as good a programming language candidate for our project as CSharp and Python were, and we had no particular reason not to choose it, other than the fact that we're not Java enthusiasts at all.
- *CSharp*. Microsoft's C# is the programming language that we master the most nowadays. It is a multi-paradigm programming language that encompasses functional, imperative, generic, object-oriented, and component-oriented programming disciplines. It has

an object-oriented syntax based on C++ and is heavily influenced by other programming languages such as Delphi and Java, whereas event handling is extremely similar to Visual Basic. CSharp is easy to learn, and it allows rapid application development; taking into account the ease of importing classes or dynamic libraries into our projects, with the internet being plagued with CSharp tutorials, projects, and code snippets already written for us by someone else, plus the fact that there exist multiple CSharp frameworks for iRobot Create, this seemed to be the wisest choice for our programming language, but we ruled CSharp out too.

- *Python.* Among all these programming languages, Python was the only one of which we had never used before by the time we started working on this project. Actually, we knew very little about Python other than the fact that it was a programming language. In spite of this, that was one of the most important reasons why we chose Python; to learn yet another programming language -and, especially, to do it on our own- would be a great challenge. Along with all the features previously mentioned, Python is the language that allows most rapid application development. We could confirm this statement ourselves by writing an application first in CSharp and then in Python, and realizing not only that we had needed less time to write the Python version of it, but also that it was many code lines shorter.

1.3.4 Choosing a robot

Strictly speaking, iRobot Create was not chosen over any other robot, somehow the use of it was more an imposition rather than a choice. Should we have had to choose the most appropriate robot for this project, we would have had to ponder whether we were more interested in a robot that was specifically designed for battling, i.e. taking part in robot wars competitions, or a robot that was, above all, easy to program. Even though it was amongst the motivations of this project to ultimately be able to hold battles between robots controlled by statecharts modelled behaviors, it was not really in the scope of the project to achieve that objective, mainly because we did not have time enough for it. Therefore, it did not seem so important whether our robot would be fit for battling or not. However, we considered some alternatives to iRobot Create:

(a) Parallax's Boe-Bot. (b) Parallax's SumoBot. (c) iBOTZ's PicoBOTZ.

Figure 1.1: Programmable robot alternatives.

(a) *Board of Education Robot (BoE-Bot)* [14], by Parallax, Inc., is an educational robot that comes fully assembled and tested, so that no soldering is required. BoE-Bot has multiple I/O components, such as LEDs, a speaker, a pushbutton, photo resistors, resistors and

capacitors, infrared LEDs and receivers and whisker contact switches. It provides object detection through its infrared sensors and its whiskers, and it's very easy to program. The language used is PBASIC, a microcontroller based version of BASIC created by Parallax, Inc. These are commonly used in class projects by Prof. Gregory Dudek, director of the Mobile Robotics Lab (MRL) [15] at McGill university.

(b) *SumoBot* [16], also by Parallax, is BoE-Bot's "fighting brother". SumoBot is specifically "designed within the Northwest Robot Mini-Sumo Tournament rules", which state that the Sumo robot must locate and knock its opponent right out of the ring. SumoBot's hardware includes a black anodized aluminum chassis and scoop, servo motors, wheels, and mounting standoffs to attach multiple and diverse peripherals.

(c) *PicoBOTZ* [17] is a programmable build-yourself robot by iBOTZ. The PicoBOTZ can be programmed with up to 180 instructions. The programs are written on a PC running Windows and downloaded into the PicoBOTZ through a serial (RS232) port. It can be programmed to avoid obstacles, follow a line and react to sound, but it has a lot less sensors than iRobot Create.

Anyhow, iRobot Create has proven to be a good choice, and it offers a broad and well balanced amount of features for the best quality-price relation:

- 30 built-in sensors, including four infrared cliff sensors on the bottom to avoid tumbling down stairs, wall sensors and front bump sensors.
- 100 Open Interface Commands that can be send individually from a PC, plus the possibility of storing command scripts on the robot.
- Complete autonomy thanks to Create's Command Module, which contains an 8-bit, 18MHz microcontroller that enables full programmability of iRobot Create's motors, lights, sounds and sensor readings. Command Module programs for iRobot Create are written in C or C++, and it has four DB-9 expansion ports for attaching extra hardware and peripherals.
- Possibility of communicating with the robot wirelessly thanks to Create's Bluetooth Adapter Module (BAM).

Figure 1.2: The iRobot Create with its Command Module attached to it.

1.4 Scope

This Master Thesis is comprised of a large theoretical research part and a shorter software development part. Our focus during the research stage was on learning modelling formalisms in general, but more specifically statecharts, since this had been chosen as the formalism we'd carry our project out with. During the software development stage, we worked fast in order to create a reasonably wide and varied set of statecharts modelled behaviors, code examples, and applications for our programmable robot. Nevertheless, none of this software applications was extremely complex, but that was not our intention either. Notwithstanding the fact that it was amongst our objectives to ultimately be able to hold statecharts controlled robot wars, we quickly realized that it would involve much more developing time than we had originally planned, a time that we didn't have anyway, not to mention the enormous amount of work it would have entailed.

It is important, then, to define what's in the scope of this Master Thesis and what is not.

√ *Learn about modelling formalisms.* We will learn about them mostly by ourselves but also by attending a two months course on modelling and simulation based design at the School of Computer Science at McGill University. It's an indispensable requirement for the execution of this project.

√ *Perform further work on "Model-based design of Computer-controlled game character behavior".* This is the name of a paper by Prof. Dr. Hans Vangheluwe et. al., which provides very accurate model-based designs of the different components of a computer-controlled game character AI. The practical demonstration of the paper's approach, though, is quite poor, so our goal is to extend and improve it. We will rule out the use of class diagrams and model the components using only statecharts instead. The models will be compiled into Python for testing purposes.

χ *Create a modelling formalism of our own.* We will neither develop new features for the statecharts formalism, this has never been among this Master Thesis's objectives.

χ *Use the iRobot Create Command Module in our experiments.* This great accessory, which allows C and C++ programming of the robot and its sensors, is not going to be used in this project and, consequently, any other kind of attachable hardware or peripheral are not going to be used either, though it originally had been one of our objectives. After seriously considering it, we concluded that we would not have time enough to develop anything interesting nor noteworthy enough to be included in this Master Thesis.

√ *Build our own iRobot Create interface.* This is amongst our main intentions even though, with a single internet search, we have discovered many open source iRobot Create interfaces written in several different programming languages. One of these interfaces that we have come across is *PyRobot* [18], which is coded in Python and it provides us with exactly what we are looking for in our interface, so we might just use *PyRobot* and extend it, if we need to do so, by writing some new functions of our own.

√ *Control the robot wirelessly.* For our first experiments with the robot we will have to have it tethered to a computer through a serial cable, but ultimately we want to control the robot by means of a bluetooth connection. It shouldn't be complicated at all.

χ *Create statecharts controlled battle robots.* Our stay at McGill University is too short to dare embark on such a complex hardware development, not to mention our very little knowledge about electronics. Besides, creating fighter robots entails figuring out a way to "attack" the enemy without physically causing damage to it, as well as figuring out a way to detect the enemy "hits". Besides these two, there are many more decisions which must be taken, that revolve around hardware attachment and programming. So Robot Wars is definitely not in the scope of this Master Thesis, but it's amongst the project's future lines.

√ *Define complex behavior statecharts* which will make the most of the robot's features. Our goal is to define statecharts that resemble the Tank Wars models as much as possible.

√ *Develop our applications in Python.* It is the most popular programming language in the academic environment -or at least it is at McGill University- and our supervisor, Prof. Dr. Vangheluwe, has a personal predilection for it since, he states, it highly increases productivity. Python is completely unknown to us, but we are always eager to learn new programming languages, and we are looking forward to diving into Python, guide in hand.

√ *Write this Master Thesis in English, and using LaTeX.* Even though we are fluent in English it is not our mother tongue, so this is probably amongst the most challenging objectives of this Master Thesis. As for LaTeX, we have never used it either but it's a requirement from Prof. Dr. Vangheluwe which we are not very enthusiastic about. Anyway, we have to admit that there are some really good things about LaTeX: we can simply focus on the content of what we write and LaTeXwill do the formatting and styling for us, plus image manipulation is way better and wiser than that of, for example, Microsoft Word.

1.5 Planning and time line

Id.	Task / subtask		Date
1	**Statecharts modelling of a computer-controlled game character behavior**		**January 08**
	1.1	Course COMP 763B: Petri Nets, assignment 1	14 Jan 08
	1.2	Course COMP 763B: Statecharts, assignment 2	21 Jan 2008
	1.3	Course COMP 763B: DEVS, assignment 3	28 Jan 08
	1.4	Course COMP 763B: Modelling languages (UPPAL, Esterel)	04 Feb 08
	1.5	Course COMP 763B: Meta-modelling, assignment 4	06 Feb 08
	1.6	Course COMP 763B: Model transformation, assignment 5	18 Feb 08
	1.7	Course COMP 763B: Rule-based modelling of model transformation	20 Feb 08
	1.8	Course COMP 763B: Reading presentation of the final course project	03 Mar 08
	1.9	Course COMP 763B: Project presentation of the final course project	28 Apr 08
2	**Get to know the iRobot Create robot**		**May 2008**
	2.1	Study iRobot Create open interface manual	01 May 08
	2.2	Tethered test experiments: serial cable connection	13 May 08
	2.3	Development/search of a Python interface to iRobot Create	14 May 08
	2.4	Wireless test experiments: serial bluetooth connection	21 May 08

3	**Statecharts modelling of a robot's behavior**		**June 2008**
	3.1	Modelling simple robot's behavior statecharts	10 Jun 08
	3.2	Modelling complex robot's behavior statecharts	20 Jun 08
	3.3	Development of *Create Remote* application	03 Jul 2008
4	**Writing Master Thesis report**		**26 Mar 08**
5	**Estimated end date**		**September 08**

The following Figure 1.3 captures the project's planning and time line in a chart.

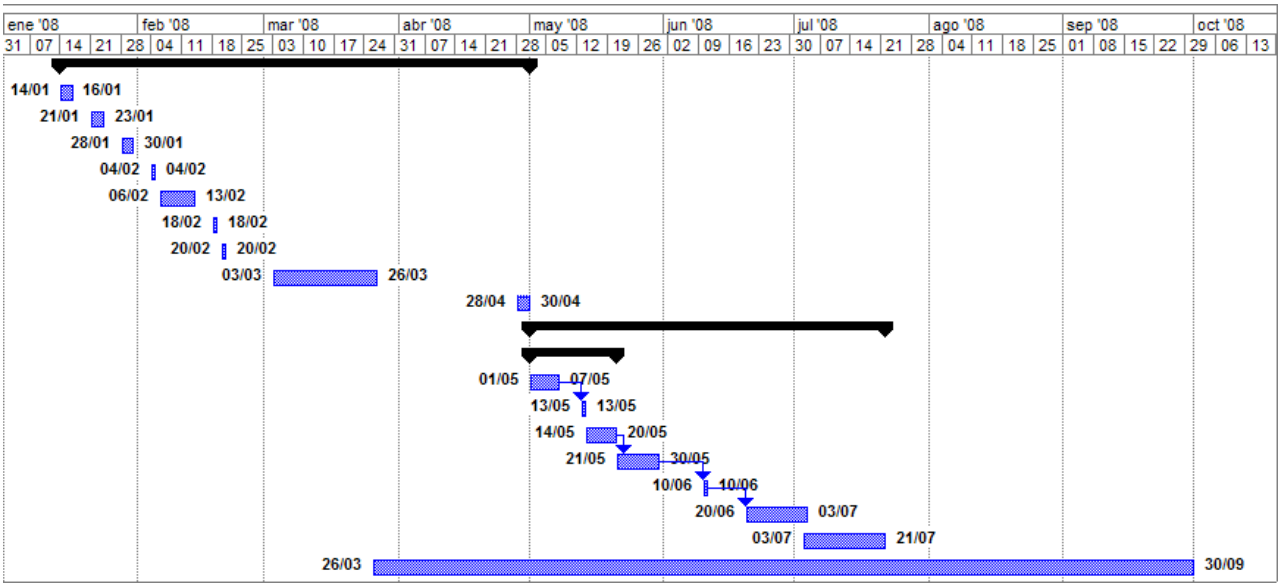


Figure 1.3: Estimated project time line.

2

Statecharts, a visual formalism

Introduction

Reactive systems are systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Reactive systems, as opposed to transformational systems, are in constant interaction with their environment, responding to external stimuli depending on their internal state. Typical examples of reactive systems are traffic control systems, programs controlling mechanical devices such as trains, a plane, or ongoing processes such as a nuclear reactor.

Modelling formalisms are extensively used to model reactive systems in order to test them prior to physically build them. Model-based testing of reactive systems is a very big subject in Computer Science nowadays, and has been for many years now. The origins of the graphical modelling formalisms are in the automata theory and the Finite State Machines (FSM). A finite state machine, or finite state automaton, is a model of behavior composed of a finite number of states, transitions between those states, and actions.

A modelling formalism is "*any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules*" [19]. Modelling languages can be graphical (a.k.a visual) or textual, but we'll only focus on the graphical ones, which "*use a diagram technique with named symbols that represent concepts, and lines that connect the symbols and represent relationships, and various other graphical annotation to represent constraints*". Some examples of visual modelling languages used in the fields of computer science, project management, and systems engineering are Behavior Trees, Petri Nets, Specification and Description Language (SDL), and Statecharts. Many of these visual modelling languages/formalisms are based on finite-state machines; statecharts specifically are an enhancement of the *Mealy machines*.

In the present chapter we'll discuss visual formalisms in general and statecharts in particular, for they are the main subject of this Master Thesis.

2.1 On visual formalisms

Visual formalisms have been used for many years to visually display information of complex and intricate nature. *Topovisual* formalisms, specifically, are those diagrammatic paradigms that are topological and relational in nature instead of geometrical, which means it's the relationships between the objects and not their shapes nor their positions that is important. Swiss mathematician Leonhard Euler is credited with creating two of the best known *topovisual* formalisms, which are graphs and *Euler circles*.

A graph is a set of objects called points, nodes, or vertices connected by links called lines or

edges. The relationship between the nodes in a set is defined by the edges connecting them and thus a graph can be *undirected* (a line from point A to point B is considered to be the same thing as a line from point B to point A) or *directed* (the two directions are counted as being distinct edges), though there are many more types or relations in the graphs field. Graphs can be modified to support a number of different kinds of nodes and edges, representing different kinds of elements and relationships.

Euler circles were modified to improve their ability to represent logical propositions by John Venn, thus creating a new formalism: the *Venn diagrams*. Although *Venn diagrams* are considered a formalism evolved from *Euler circles*, very frequently both names are used indistinctly when referring to either one of the two. In *Venn diagrams*, a set is represented by the inside region of a closed curve that divides the plane into disjoint inside and outside regions, introducing the topological notions of:

- *enclosure*, meaning "being a subset of",
- *exclusion*, meaning "being disjoint from",
- and *intersection*, meaning "having a nonempty intersection with".

Therefore, the main difference between graphs and *Euler circles/Venn diagrams* is that the first provide a way of representing a set of elements with some relations on them, whereas the latter provide a way of representing *collections* of sets with some structural relationships between them. When representing some complex systems both capabilities are needed in order to represent a number of sets that are structurally interrelated, but there are also one or more additional relationships of special nature between them. Moreover, in order to prevent certain representations from growing exponentially in size, it's desirable to identify the Cartesian product of some of the sets.

This is the motivation behind the creation of *higraphs*, a formalism that combines the capabilities of both graphs and *Euler circles/Venn diagrams* and extends them to represent the Cartesian product, finally connecting the resulting curves by edges or hyperedges.

2.1.1 *Higraphs*

To illustrate the definition of *higraphs* [20], a topovisual formalism introduced by David Harel in 1988, we'll start by taking a look at Figure 2.1, which captures a simple example of Euler circles with a special feature: instead of circles we use rounded rectangles, and from now on we'll refer to the areas they enclose as *blobs*. Each *blob* denotes a certain kind of set and the nesting of curves denotes set inclusion. Therefore, Figure 2.1 can be seen to contain several cases of inclusion, disjointness, and intersection of sets. As the next step into *higraphs* definition, we now require that every set of interest be represented by a unique *blob*, complete with its own contour. This is captured in Figure 2.2, where the result of each inclusion, disjointness, and intersection of sets is enclosed in a *blob* and labeled with a capital letter, thus creating new sets.

So in Figure 2.2, among the many depicted *blobs*, we can make C out as the set resulting from the intersection of sets A and D, whereas B is the set resulting from the difference of sets A and D. More precisely, with this "unique-contour" convention, the only real, identifiable sets are the *atomic sets*, that is, those represented by *blobs* residing on the bottom levels of the diagram, containing no enclosed *blobs* within. Any other *blob* merely denotes the compound set consisting of the union of all atomic and non-atomic sets represented by *blobs* that are totally enclosed within it. Therefore, the atomic blobs in Figure 2.2 are *B, C, E, G, H, J, K, M, O, P, R, S*, and *T*.

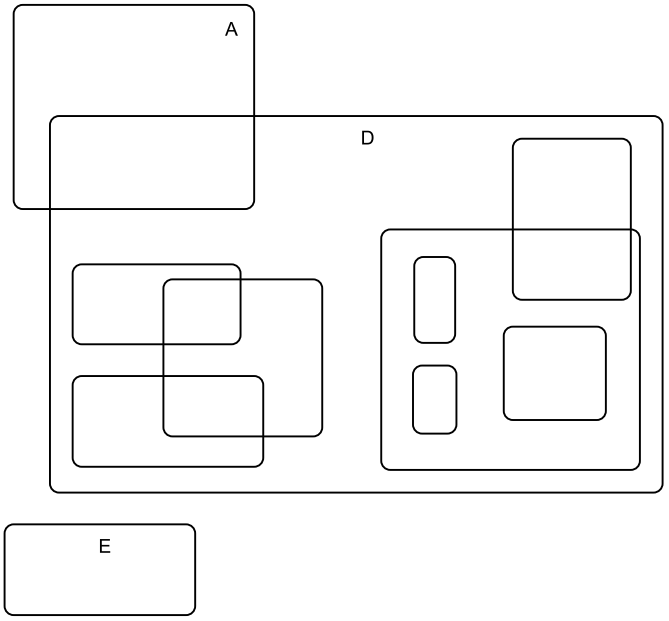


Figure 2.1: Simple *blobs* [20].

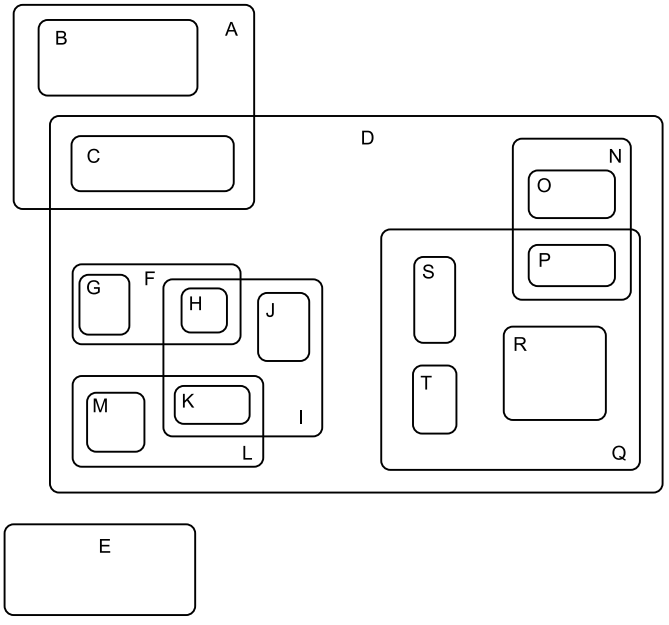


Figure 2.2: Adding unique contours for all identifiable sets [20].

Next we add the ability to represent the *Cartesian product*. Figure 2.3 shows the notation used: a partitioning by dashed lines. In this diagram, Q is no longer the union of P, R, S, and T, but, rather, the product of the union of S and T (U) with the union of P and R (W). We will call *blobs* U and W the *orthogonal components* of *blob* Q. The *Cartesian product* is unordered, so Q is really a set of unordered pairs of elements.

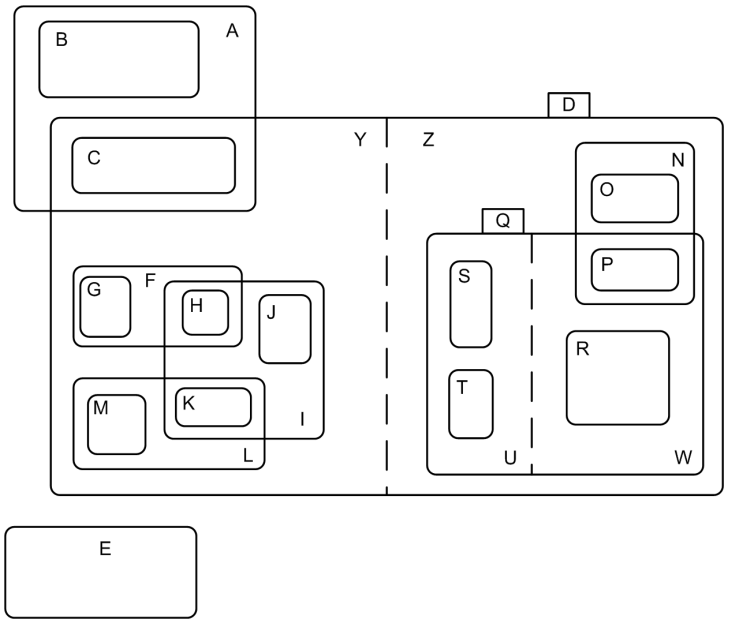


Figure 2.3: Adding Cartesian products [20].

So far, we have defined a formalism for representing sets and their structural, set-theoretic relationships; now it is time to add edges. A *higraph* is obtained by simply allowing edges, or more generally, *hyperedges*, to be attached to the contour of *any blobs*. As in graphs, edges can be directed or undirected, labeled or unlabeled, of one type or of several, etc. *Hyperedges* can connect *blobs* at different "nesting levels" of the diagram, that is, atomic and non-atomic sets indistinctly.

The computer science literature is full of uses of graphs, and it appears that many of these can benefit from the extensions offered by *higraphs*, for example *entity-relationship* diagrams. We will not delve any deeper into *higraphs* and its many applications since, at this point, we have already provided the fundamentals of statecharts, and that's what we will focus on in the following subsection.

2.1.2 Statecharts

The first definition of statecharts given by its creator, David Harel, refers to statecharts as *"essentially a higraph-based version of finite state machines and their transition diagrams"* [20]. Statecharts are a visual formalism for modelling large and complex reactive systems. As opposed to transformational systems, reactive systems are event-driven, continuously having to react to external and internal stimuli. Some examples of reactive systems are automobiles, electrical appliances such as microwaves, dishwashers or televisions, and communications networks. Like in finite state automata, statecharts use a state/event representation to describe the behavior

of complex reactive systems, as seen in Figure 2.4. An example of behavior of this state diagram would be: "when in state A, if event e1 occurs the system transfers to state B", this is called a *state transition*. However, to be useful, such a state/event approach like this must be modular, hierarchical, well structured, and it must solve the exponential blow-up problem by relaxing the requirement that all combinations of states have to be represented explicitly; as well, it should allow

- the clustering of states into a superstate,
- independence, or orthogonality,
- the definition of more general transitions than the single event-labeled arrow,
- and the refinement of states.

Statecharts fulfill all this requirements, thus constituting a visual formalism for describing states and transitions that also enables clustering, orthogonality (i.e., concurrency) and refinement, and also provides 'zoom' capabilities for moving easily between levels of abstraction [21]. In the following subsections we'll discuss the features of statecharts.

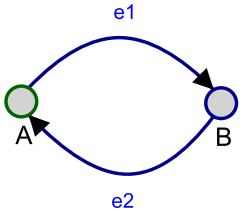


Figure 2.4: "When in state A, if event e1 occurs the system transfers to state B."

2.1.2.1 Clustering and refinement

In the statecharts graphical notation, states are represented by rounded rectangles, or *blobs* -as seen in higraphs-. Depth and hierarchy are denoted using encapsulation, and arrows are allowed to originate and terminate at any level. Thus, in the example statechart depicted in Figure 2.5, A, B, C, and D are states, with A, B, and C being *atomic states*, because neither of them encapsulates any states. Therefore, D is considered a superstate, or *composite state*, and represents the highest level of hierarchy, whereas A, B, and C represent the deepest level of hierarchy.

Arrows will be labeled with events, i.e.: "when in state B, if e2 occurs the system will transfer to state A". Optionally, arrows will also be labeled with a parenthesized condition, i.e.: "when in state A, if e1 occurs and condition P evaluates to true, the system will transfer to state C". Also, as depicted in the left side of Figure 2.5, when either in state A or state C, if e4 occurs the system will transfer to state B, hence it doesn't matter if the system is either in A or C and both states can be clustered into superstate D, and the two arrows labeled with the e4 event can be replaced by only one arrow, originating in superstate D and terminating in state B. The semantics of D is then the exclusive-or (XOR) of A and C, which means that to be in state D one must be either in A or in C, and not in both.

Refining, as opposed to clustering, consists in breaking down a composite state into a set of states (not necessarily atomic). E.g.: another approach to the right side of Figure 2.5 could have been Figure 2.6; on the left side, "when in state B, if event e2 or event e3 occur, the system will

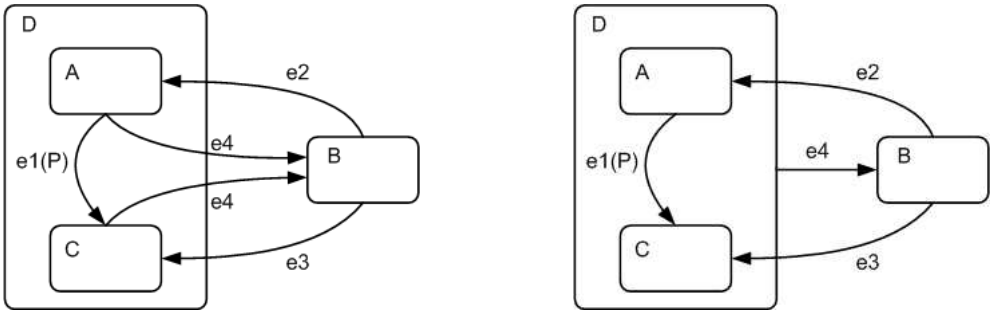


Figure 2.5: A and C can be clustered into superstate D.

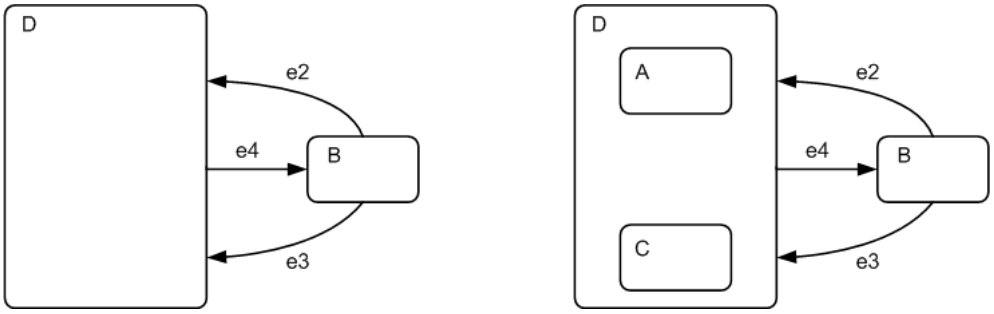


Figure 2.6: State D can be refined to consist of A and C.

transfer to state D". State D could then have been refined to consist of atomic states A and C, as depicted on the right side of Figure 2.6. Consequently, a question arises: when in state B, if event e2 or event e3 occur, which state in composite state D will the system transfer to? This introduces the notion of *default state*, which is the state that will be entered by default unless another state is specified. It is necessary to define a default state at every hierarchy level and for every set of composite states, i.e.: on the right side of Figure 2.6, D is the default state among D and B, and A is the default state among A and C. Therefore, if asked to enter the A, B, C group of atomic states the system will enter A unless otherwise specified. The graphical notation of the default state is as captured in Figure 2.7.

Let's have a look at some more complex examples to better understand the concepts of composite and default states. The following statecharts have been defined with a software tool called AToM^3, covered in section 2.2. As we'll see, default states are graphically represented with a green outline color, whereas the rest of states have a blue outline color. In Figure 2.8 states AB, C, and XY are at the same hierarchy level and therefore only one of them must be appointed the default state among the three; in this case, it's AB. In composite state AB, A is default among A and B, and in composite state XY, X is default among X and Y.

In the case of adding more atomic states to Figure 2.8, as captured in Figure 2.9, we would have to choose between maintaining state AB as the default, or appointing state M OR state N as the new default state among AB, C, XY, M, and N. We choose to maintain state AB as the default.

Back to Figure 2.8, if states AB and C were to be clustered into an ABC composite state, as depicted in Figure 2.10, a default state should be appointed among ABC and XY; in this case

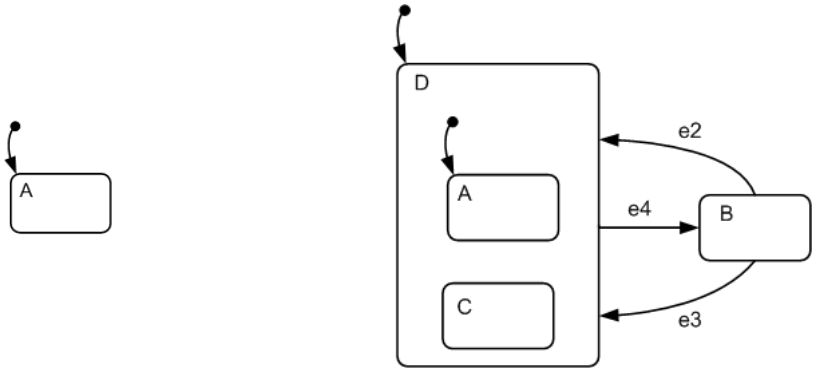


Figure 2.7: Introducing the default state.

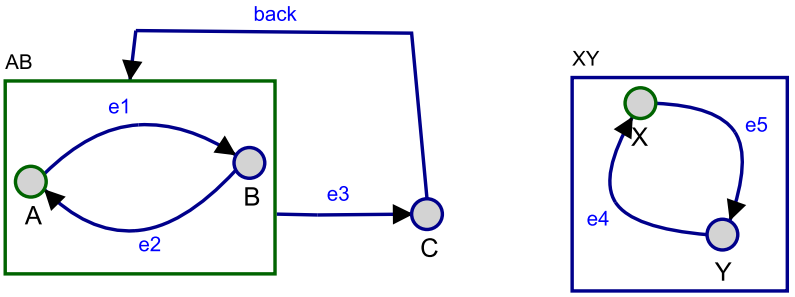


Figure 2.8: State AB is default among states AB, C, and XY.

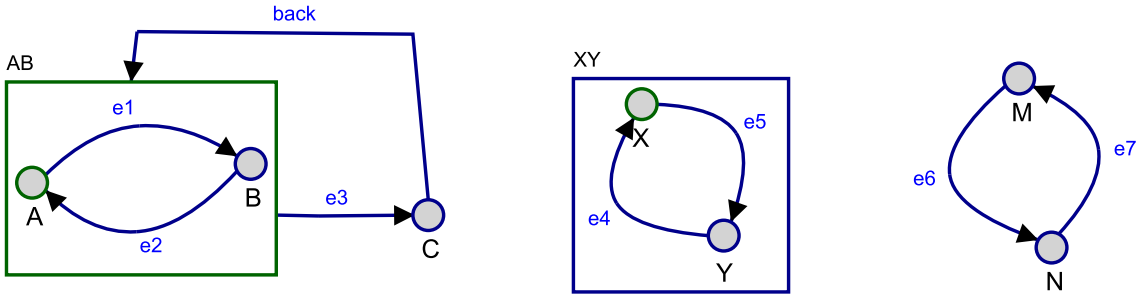


Figure 2.9: State AB is default among states AB, C, XY, M, and N.

it's XY. Composite state AB would still be the default state among AB and C, i.e.: if the system was to enter state ABC, it would enter state A, which is default among A and B.

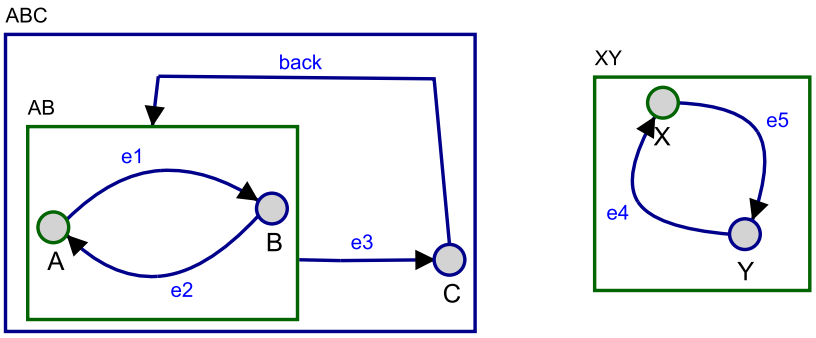


Figure 2.10: State XY is default among states ABC and XY.

Finally, retaking Figure 2.9 and clustering states XY, M, and N into composite state XYMN, as captured in Figure 2.11, again forces us to decide whether to maintain our current default state or appointing a different one. In this case, we appoint state XYMN as new default state among AB, C, and XYMN, whereas state M is appointed default state among XY, M, and N.

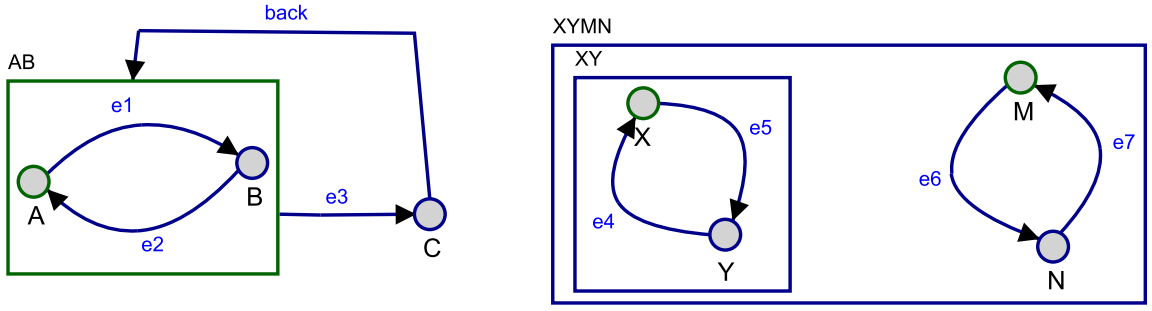


Figure 2.11: State XYMN is default among states AB, C, and XYMN.

2.1.2.2 Orthogonality: Independence and concurrency

We've introduced the XOR (exclusive-or) decomposition of states with the notions of clustering and refinement in the previous section. Orthogonality, on the other hand, introduces the AND decomposition of states, which captures the property that, being in a state, the system must be in all of its AND components. The graphical notation used to represent orthogonality in statecharts involves the physical splitting of a box into components using dashed lines, as captured in Figure 2.12; in this case we say that Y is the orthogonal product of A and D, meaning that being in state Y entails being in some combination of B or C with E, F or G. As in composite states, it's required to appoint a default state for each orthogonal component of a statechart. In Figure 2.12, B is the default state for orthogonal component A, and F is the default state for orthogonal component D. This means that, when entering state Y from the outside without providing more specific information, the combination of states (B, F) will be entered. As well, when in combined state (B, F), if event e1 occurs the system will transfer B to C and F to G simultaneously, thus resulting in the new combined state (C, G). However, if then event

e5 occurs, only the D component will be affected and the system will transfer G back to F, thus resulting in the new combined state (C, F). The first example illustrates *concurrency*: a single event, e1, causes two simultaneous transitions. The second example illustrates *independence*: when event e5 occurs it doesn't matter which state of the A component, B or C, the system is in, because the transition will have the same effect in the D component, transferring state G to state F.

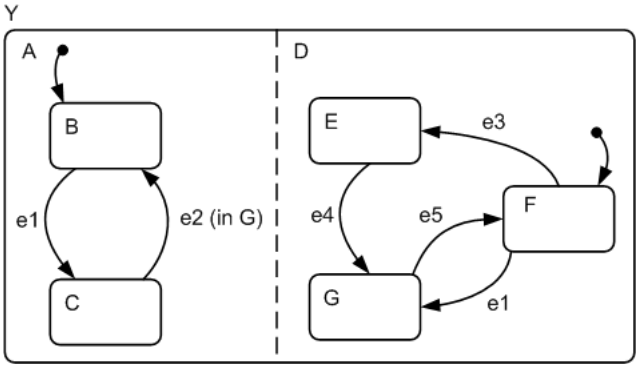


Figure 2.12: State Y is split into orthogonal components A and D.

Concerning orthogonality we can assert that, in a statechart that is composed of a set of orthogonal components, the system will be simultaneously in as many concurrent states as orthogonal components the statechart is composed of. Also, using orthogonality introduces the possibility of labeling transitions with conditions, thus creating conditioned transitions; e.g.: *e (C)*. In this example, when event e occurs, the transition will be taken only if condition C is satisfied. The condition between brackets accepts any boolean expression. We have another example in Figure 2.13: *e2 (in G)*, which means that, when event e2 occurs, the system must be in state G for the transition to be taken. Therefore, as we can observe, C will only transfer to B if orthogonal component D is in state G when event e2 occurs. But also it means that, whenever orthogonal component A is in state C, the only way to transfer it to state B is by concurrently having orthogonal component D in state G.

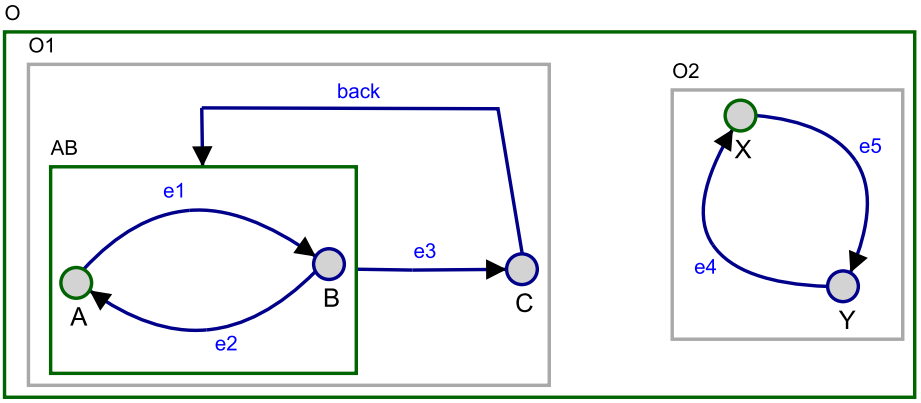


Figure 2.13: Orthogonal components O1 and O2 behave completely independently.

To better understand the concept of orthogonality we will now discuss a few more statecharts examples of higher complexity. Figure 2.13 depicts a state O consisting of two orthogonal components, O1 and O2. The default combined state of O is (A, X), being A and X the default states for O1 and O2 respectively. O1 and O2 don't have any common events and, therefore, they will behave completely independently.

The O state in Figure 2.14 consists of three orthogonal components, O1, O2, and O3. The default combined state of O in this case is (A, X, M). If event e3 occurs when the system is in combined state (A, X, M), state A will be transferred to state C in component O1, resulting in the new combined state (C, X, M). Note that, even though components O1 and O3 have event e3 in common, component O3 is not affected by the triggering of event e3 in this case, because its current state is M and not N. Also, it actually does not matter whether component O1 is in state A or B, because the resulting combined state would be the same: (C, X, M).

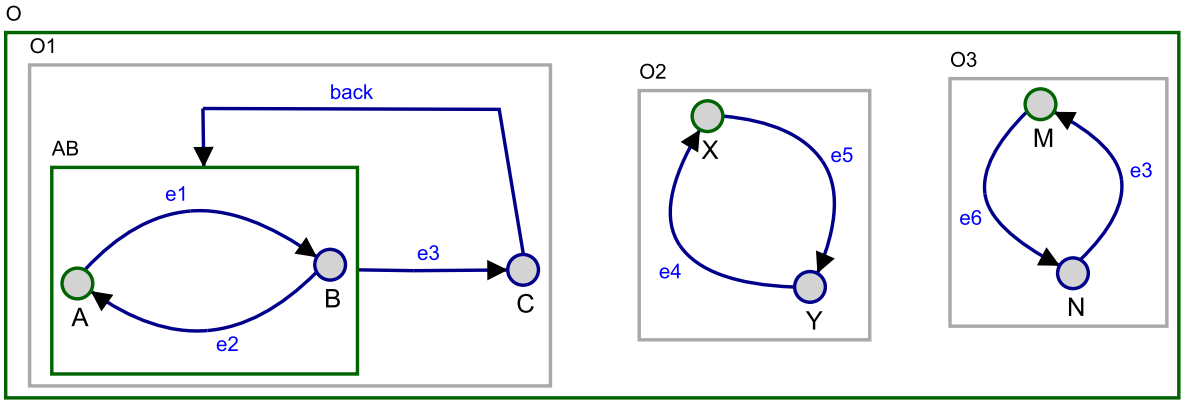


Figure 2.14: When in combined state (A, X, M), if event e3 occurs the system transfers to combined state (C, X, M).

Finally, Figure 2.15 doesn't differ very much from the previous one. In it, orthogonal components O1, O2, and O3 all have event e3 in common. The default combined state of O is also (A, X, M) but, in this case, if event e3 occurs the system will transfer AB to C in component O1 and X to Y in component O2, resulting in the new combined state (C, Y, M). If event e4 occurs then, the condition between brackets will be satisfied in component O2 (component O1 is in state C), and the system will transfer the current combined state to (C, X, M). However, if the event *back* occurs prior to e4, the current state in orthogonal component O1 will be transferred from C back to A (because of being the default state), thus resulting in combined state (A, Y, M).

2.1.2.3 History

To easily explain the notion of history in statecharts, we'll introduce Figure 2.16, which captures the behavior of an average cat. As we can observe in this statechart, a cat's daily activity basically consists of sleeping, eating and rinsing. When the cat's master comes, he will interrupt the cat's activity and start stroking it, no matter what the cat was doing at that moment. When the master goes away, the cat will want to go back to what it was doing before being stroked instead of start sleeping, which is its default activity. Hence, this is the use of the history state; as we can observe in Figure 2.16, when the *master_goes* event occurs, the system will transfer

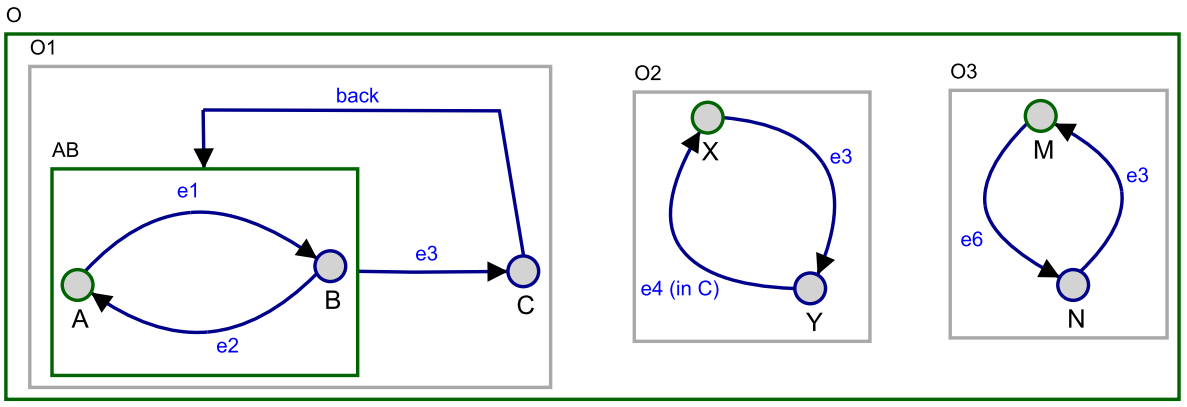


Figure 2.15: When in combined state (C, Y, M), if event e4 occurs the system transfers to combined state (A, Y, M).

state *Being stroked* to the history state, which in statecharts is graphically represented by a circled H. Entering the history state is tantamount to entering the most recently visited state, therefore, if the cat was eating before its master came and started stroking it, the cat will go back to eating when its master goes away.

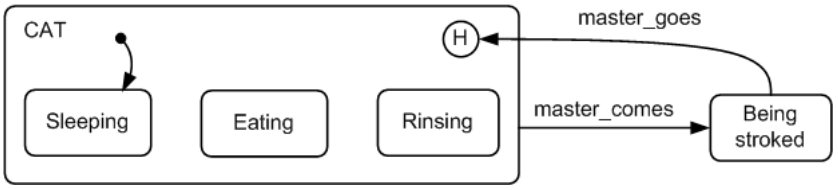


Figure 2.16: Introducing the history state.

The statecharts captured in Figure 2.17 takes us a little bit further with our definition of history. As we can observe, there are three superstates F, G, and J, which encapsulates the first two. It should be noted that an H generally means that history is applied only on the level in which it appears; thus, when the history state is entered in the statechart on the left side of Figure 2.17, it will choose only between superstates F and G. This means that if the most recently visited state was either A or B, superstate F will be chosen, thus entering state A (which is default among A and B); and if it was either C or D or E, superstate G will be chosen, thus entering state D (which is default among C, D, and E). On the other hand, when using *deep history*, or H*, history is applied all the way down to the lowest level of states. So, if we replace H by H* in the same statechart, as captured on the right side of Figure 2.17, when the history state is entered it will choose between the most recently visited state among A, B, C, D, and E, overriding both defaults.

2.1.3 Actions

What we've explained about statecharts so far is that the system changes its internal state-configuration in response to incoming or sensed events and conditions. But then, what's the use of having composite and orthogonal components in a statechart, if they can't interact with

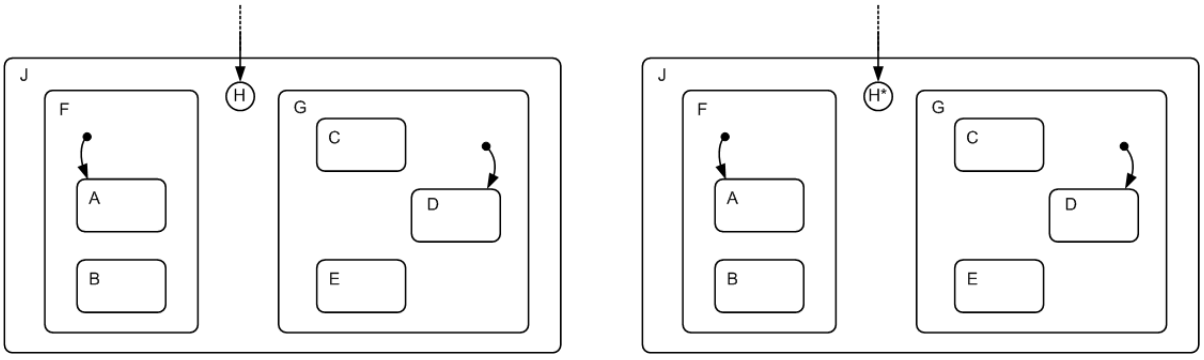


Figure 2.17: Normal history Vs. Deep history.

each other? How do components influence other components of the system? Statecharts have the ability to generate events and to change the value of conditions. These can be expressed by the notation *"e/S"* that can be attached to the label of a transition, where S is an *action* carried out by the system in response to the event e being triggered and causing the transition in question to be taken. Obviously, if the event e is received but the transition is not taken, the action S is not carried out.

Actions are split-second happenings, instantaneous occurrences that take ideally zero time, as does sending a signal or a conventional assignment statement. If we take a look at the example statechart captured in Figure 2.18 we see that, upon sensing the event e, the e transition in orthogonal component A is taken -thus changing its current state from *A1* to *A2*-, and the action S is carried out, generating S as an event that can be sensed elsewhere in the statechart. Indeed, the S transition in orthogonal component B is taken instantaneously. Note that the fact that actions take zero time doesn't mean that events e and S are triggered at the same time; event triggering is always sequential.

We will discuss actions and its applications in more detail in the following section, 2.2.

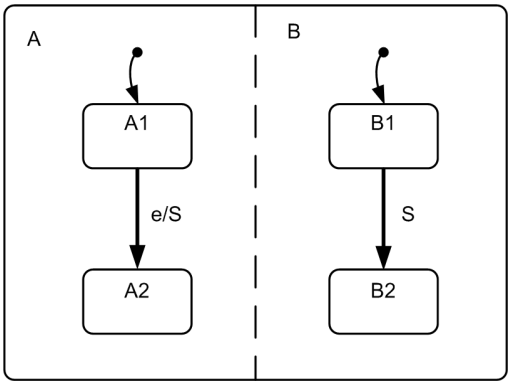


Figure 2.18: Event S is generated at orthogonal component A and triggers a transition in orthogonal component B.

2.2 AToM^3: A Tool for Multi-formalism Meta-Modelling

AToM^3 [4] is a tool for multi-paradigm modelling under development at the *Modelling, Simulation and Design Lab* (MSDL) [1] in the School of Computer Science of McGill University. It is developed in collaboration with the School of Computer Science of Universidad Autónoma de Madrid. The two main tasks of AToM^3 are *meta-modelling* and *model-transformation*. Meta-modelling is the modelling of a formalism used to model systems, and model-transformation is the automatic process of converting, translating, or modifying a model in a given formalism, into another model that might or might not be in the same formalism. Some of the meta-models currently available are: Entity-Relationship, GPSS, Deterministic Finite state Automata, Non-Deterministic Finite state Automata, Petri Nets, Data Flow Diagrams and Structure Charts.

The meta-modelling feature allows us to easily define new modelling formalisms and include them in AToM^3's index. We could, for example, define our own traffic system modelling formalism: we would have to describe the formalism's entities and their relationships (a series of rules that specify how the entities can be interconnected), and we would have to draw a "graphical representation" or icon of each of the entities -AToM^3 provides a very simple drawing tool for this purpose-. Once we would be done defining the meta-model and we would have compiled it, we could load the formalism on AToM^3 -its toolbar is automatically generated- and start modelling traffic systems.

When designing reactive systems we're conditioned by the modelling tool we use, even though, ideally, it shouldn't be this way. All the statecharts models captured in many of the Figures of this Master Thesis have been modelled with AToM^3. We mainly used DCharts, *"a formalism for modelling and simulation of complex reactive software systems"* which *"is based on UML statecharts and DEVS, but provides better modularity and expressiveness"*.

The DCharts formalism was created by Thomas Feng [22] for AToM^3, along with the Statechart Virtual Machine (SVM) and the StateCharts Compiler (SCC). SVM is "originally a statecharts simulator implemented in Python, but now it supports the complete DCharts semantics and the textual syntax, including the syntactic extensions", and it can be used with AToM^3 as an external utility.

SCC [5] is a "command-line tool to synthesize executable code from DCharts models. It optimizes the models and produces efficient code". SCC is distributed with SVM, but the code generated by SCC is independent of the SVM simulator. SCC is able to synthesize code in Java, C++, C# and Python.

SVM pre-defines a number of macros that can be used in DCharts modelling. When a DCharts model is compiled with SCC, these macros are translated into methods of the generated classes. Some of the macros that we will most commonly use are:

- `EVENT(ev)`. This macro is used to raise an event, which is broadcasted to the model. The event name is given by parameter `ev`. Parameter `p` can be used as a parameter or a list of parameters for the event. By default it is an empty Python list.
- `AFTER(sec)`. This macro is used to define a timed-transition, that is, an unlabeled transition which will be automatically taken `sec` seconds after the origin state is entered.
- `INSTATE(state)`. This macro checks whether the model is currently in a specific state, and it should only be used in the guards of transitions, as the DCharts formalism requires that the actions of a model cannot reflect upon the current state of the model itself.
- `PARAMS`. This macro can only be used in the guards or output of transitions. It returns

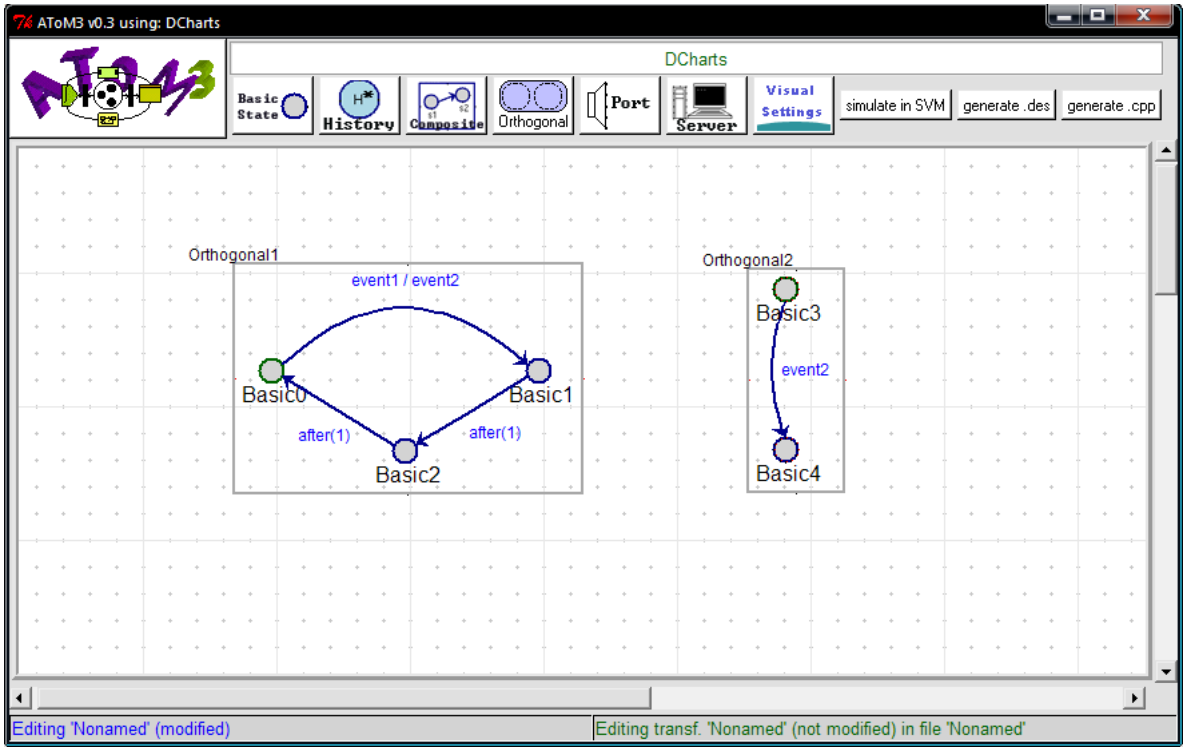


Figure 2.19: AToM^3's main application window.

the parameter of the event that triggers a transition (this parameter can be optionally provided if the event is raised externally).

- `DUMP(msg)`. This macro dumps a message to the output device (console window).

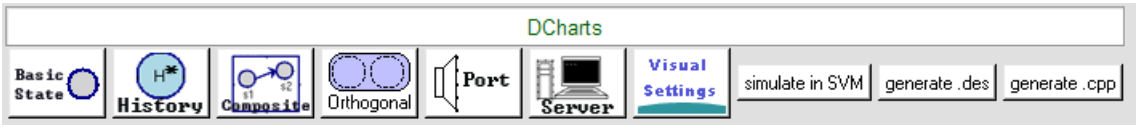


Figure 2.20: Toolbar of the DCharts formalism.

Next we're going to see some examples on how to model DCharts in AToM^3. Figure 2.19 captures AToM^3's main application window. By pressing F3 we choose a formalism to be loaded, and its corresponding toolbar appears on top of the modelling canvas. Figure 2.20 provides a closer look at the toolbar of the DCharts formalism; we can load as many formalisms as we want to. Once we've selected an entity from the toolbar, we may create instances of it by pressing the Control key and clicking the mouse right button on the modelling canvas. Entities are linked by hyperedges; these can be created by pressing the Control key and clicking the mouse left button on any of the entities on the canvas. The Control key must be held down while the hyperedge is traced; in order to link the origin entity to another -or itself-, the mouse left button must be clicked again on the destination entity.

Entities and hyperedges can be edited by selecting them from the canvas and pressing the 'E'

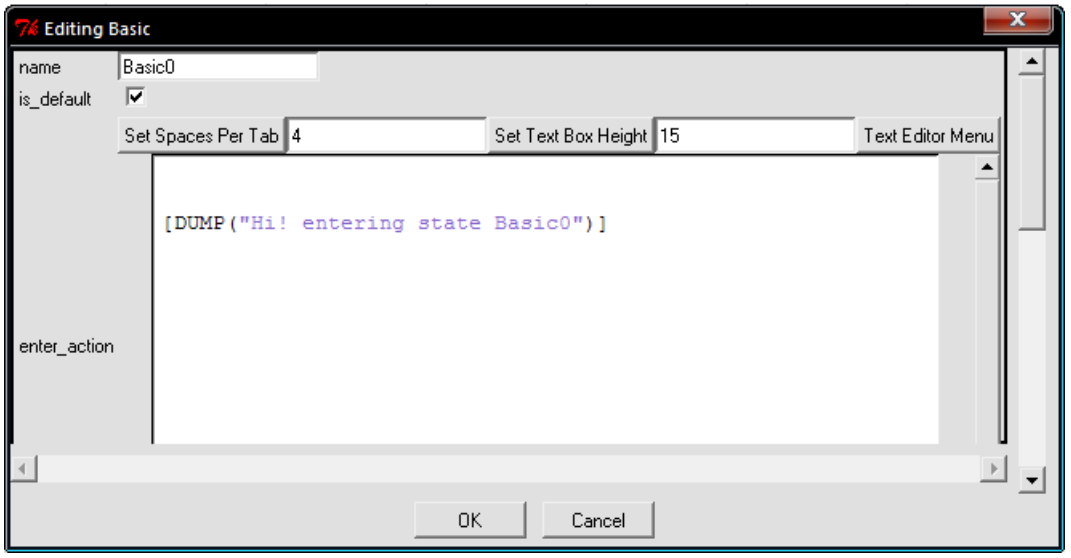


Figure 2.21: Editing a state.

key. Figure 2.21 captures the state edition window; as we can see, a checkbox is used to specify if the state being edited is a default state or not. Note that both states *Basic0* and *Basic3* -from the statechart depicted in Figure 2.19-, are default states, since they belong to different orthogonal components of the statecharts model. All states at the same nesting level can be marked as default, which is not possible according to the statecharts features and therefore AToM^3 would not compile such a model. The *enter_ action* textbox of the state edition window is where we write the code to be executed when the state in question is entered. We can code in any of the supported programming languages, but only single line statements are allowed. There is also an *exit_ action* textbox in the bottom of the state edition window.

Figure 2.22, in turn, captures the hyperedge edition window. The *trigger* textbox is where we may specify the name of the event which will trigger the transition described by the hyperedge that's being edited. As we have previously mentioned, we can use SVM's pre-defined macros to turn a hyperedge into a timed-transition by writing `[AFTER(sec)]` instead of an event's name in the *trigger* textbox. We can also turn a hyperedge into a conditional transition by specifying a boolean expression (a condition) in the *guard* textbox, such as:

- `( x == true )`. Where `x` is a boolean variable which must have been previously defined in the statechart.
- `( y > 10 )`. Where `y` is an integer variable which must have been previously defined in the statechart.
- `checkCondition()`. Where `checkCondition` is a method returning a boolean value. In order for the statechart to be able to call this method, it should be defined in a class, and an instance of this class should be passed to the statechart as a parameter.
- `[INSTATE(Orthogonal2.Basic3)]`. This is also one of SVM's pre-defined macros. When the transition in question is taken, the model must currently be in the specified state in order for the transition to be taken.

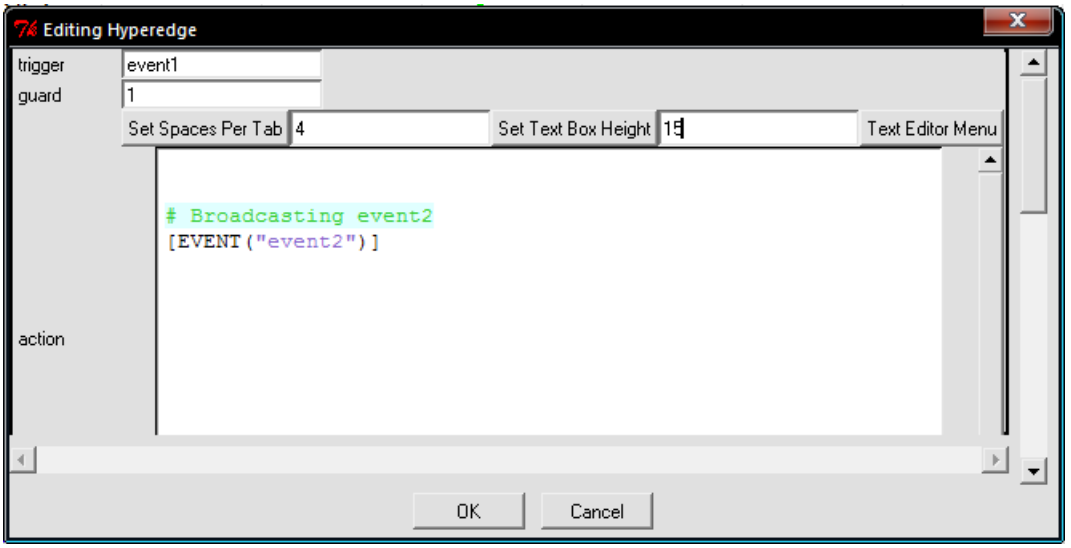


Figure 2.22: Editing a hyperedge.

Finally, and still referring to the hyperedge edition window captured in Figure 2.22, the *action* textbox is where we write the code to be executed when the transition in question is triggered and taken. Only single line statements are allowed as well.

2.3 A reactive system example: the digital watch

Now that we've seen the most important features of statecharts, we propose an example to compile them all and put them into practice, by completely modelling the detailed behavior of a reactive system: a digital watch. Let us first describe the basic operation of our digital watch:

- If we assume that the watch can be turned on and off, when it's turned on it will be displaying the current time by default.
- The time value should be updated every second, even when it is not displayed. However, time is not updated when it is being edited.
- Pressing the top left button alternates between the stopwatch and the time display modes.
- Pressing the bottom right button starts the stopwatch when in stopwatch display mode. The running stopwatch updates in 1/100 second increments. Subsequently pressing the bottom right button will pause/resume the stopwatch. Pressing the bottom left button resets the stopwatch. The chrono will keep running (when in running mode) or keep its value (when in paused mode), even when the watch is in a different display mode.
- Pressing the top right button turns on the Indiglo light. The light stays on for as long as the button remains pressed. From the moment the button is released, the light stays on for 2 more seconds, after which it is turned off.
- Pressing the bottom right button for at least 1.5 seconds when in time display mode will put the watch into time editing mode.
- Pressing the bottom left button when in time display mode sets the alarm on or off. If the bottom left button is held for 1.5 seconds or more, the watch goes into alarm editing mode. The alarm is activated when the alarm time is equal to the time in display mode.

When it is activated, the screen will blink for 4 seconds, then the alarm turns off. The alarm can be turned-off before the elapsed 4 seconds if any button is pressed. After the alarm is turned off, activity continues exactly where it was left-off.

- When in (either time or alarm) editing mode, briefly pressing the bottom left button will increase the current selection. If the bottom left button is held down, the current selection is incremented automatically every 0.3 seconds. Editing mode should be exited if no editing event occurs for 5 seconds. Holding the bottom right button down for 2 seconds will also exit the editing mode. Pressing the bottom right button for less than 2 seconds will move to the next selection (for example, from editing hours to editing minutes).

Simply by reading these requirements we can clearly identify the orthogonal components of the digital watch statechart. We'll describe and discuss each component separately in the following subsections, assuming that

- the digital watch statechart has two internal variables: *Tcurrent*, the value of which is the current time, and *Talarm*, the value of which is the alarm time;
- *tm(x)* defines a **timed transition**, which is not triggered by an event but by the passing of time. A timed transition *tm(x)* is automatically taken after x seconds from the moment the origin state was entered;
- every state may have some *enter action* and/or *exit action* code defined, and every event may have some *output action* code attached too.

2.3.1 Display mode

This is quite a complex statechart, as captured in Figure 2.23. The default state is *time_ display*, in which the current time is displayed on the digital watch screen. Assuming that the time display is refreshed/updated every time that state *time_ display* is entered, it must be exited and re-entered every one second in order to properly display the passage of time. That's the purpose of the event labeled *tm(1)*, which originates and terminates in state *time_ display*. We must remark that the refreshing of the time and the stopwatch displays is independent from the updating of the time and the stopwatch corresponding values. The other events originating in *time_ display* are *topLeft* [button pressed] and *bottomLeft* [button pressed], which transfer the display component to *chrono_ display* and *alarm_ display* respectively.

Once state *chrono_ display* is entered, it's exited and re-entered every 0.01 seconds in order to refresh the stopwatch display. If the top left button is pressed again the display component is transferred back to state *time_ display*.

2.3.2 Time update

State *time_ running*, as captured in Figure 2.24, is exited and re-entered every second in order to simulate the passage of time by updating the value of Tcurrent. If the digital watch goes into time editing mode, the event *editTime* is received from the display component's statechart, and the time component is transferred to state *time_ off*. While in editing mode, the watch is stopped and thus the value of Tcurrent is not updated. When the editing mode is exited, the display component broadcasts the event *resumeTime*, which transfers the time component back to state *time_ running*.

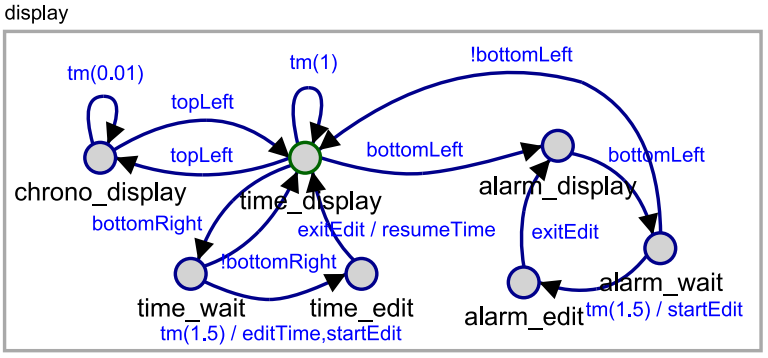


Figure 2.23: *Display* component of the digital watch statechart.

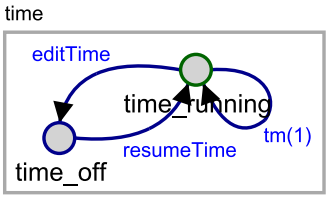


Figure 2.24: *Time* component of the digital watch statechart.

2.3.3 Stopwatch

The digital watch chronometer/stopwatch statechart is depicted in Figure 2.25. In order to enter the chrono mode, the bottom right button of the watch must be pressed while the chrono is on display: *bottomRight [in chrono_ display]*. This conditioned transition transfers the stopwatch component from its default state to *chrono_ running*. While in this state the chrono is updated every one millisecond, the same rate at which the chrono display is refreshed. Pressing the bottom right button again while the chrono is running pauses the chrono, and viceversa. As well, pressing the bottom left button while the chrono is paused transfers the stopwatch from state *chrono_ paused* back to its default state, thus reseting the counter. If the top left button is pressed while the stopwatch is running, the digital watch goes back to displaying the time, but the chrono keeps running in the background.

2.3.4 Alarm status

As captured in Figure 2.26, the alarm's default state is off. When the digital watch is in *time_ display* mode and the bottom left button is pressed, the current state is transferred to *alarm_ display* and the event *setAlarm* is broadcasted. When the *setAlarm* event is received in the alarm_status statechart, the current state is transferred to *alarm_ enabled* and the alarm is set on. If the *setAlarm* event is received again, the alarm_status component is transferred back to state *alarm_ disabled*3 and the alarm is set off. While in state *alarm_ enabled*, the alarm time and the current time must be compared every second (Tcurrent == Talarm ?); if the alarm time is equal to the current time the alarm_status component broadcasts the *alarmStart* event, which activates the alarm.

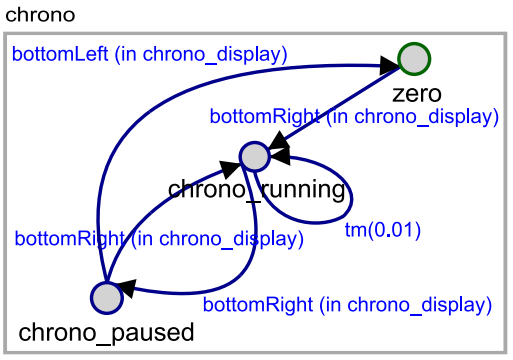


Figure 2.25: *Chrono* component of the digital watch statechart.

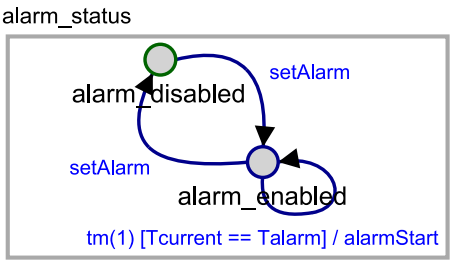


Figure 2.26: *Alarm_ status* component of the digital watch statechart.

2.3.5 Alarm

When the event *alarmStart* is received from the alarm_status component of the digital watch, the alarm is activated. The *alarmStart* event, as depicted in Figure 2.27, transfers the alarm component's current state from *alarm_ off* to *blink_ on*, which is contained within composite state *alarm_ blink*. The current state transfers from *blink_ on* to *blink_ off* every 0.5 second. After 4 seconds, unless any button is pressed before, the alarm component is transferred back to the *alarm_ off* state, thus turning the alarm off.

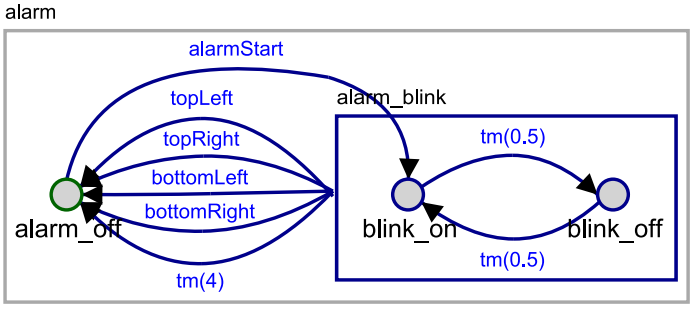


Figure 2.27: *Alarm* component of the digital watch statechart.

2.3.6 Editing mode

When in time display mode, pressing the bottom right button and holding it pressed for 1.5 seconds turns the time editing mode on. We can observe by looking at the statechart depicted in Figure 2.23 that, releasing the bottom right button before 1.5 seconds, transfers the component from state *time_ wait* back to state *time_ display*; otherwise, state *time_ edit* is entered and the events *editTime* and *startEdit* are broadcasted.

In a very similar way, pressing the bottom left button and holding it pressed for 1.5 seconds, when in alarm display mode, turns the alarm editing mode on. The current state transfers from *alarm_ wait* to *alarm_ edit* and the event *startEdit* is broadcasted as well.

The *editTime* event causes the time to stop running, by transferring the *time* component's current state from *time_ running* to *time_ off*. Event *startEdit*, in turn, triggers the transition from state *edit_ main* to state *edit_ sel*, in the *edit* orthogonal component of the digital watch statechart, which is captured in Figure 2.28. Once in *edit_ sel* state, we can really start editing.

When in (either time or alarm) editing mode, pressing the bottom right button for less than 2 seconds transfers the current state from *edit_ sel* to *edit_ held* and immediately back, causing the watch's edit display to move to the next selection (for example, from editing hours to editing minutes). Alternatively, briefly pressing and releasing the bottom left button transfers the current state from *edit_ sel* to *edit_ auto*, increasing the current edit selection. If the bottom left button is held down instead, the current selection is incremented automatically every 0.3 seconds.

Finally, pressing the bottom right button and holding it pressed for 2 seconds or more while in state *edit_ held*, transfers the *edit* component back to state *edit_ main*, thus exiting the editing mode and broadcasting the event *exitEdit*. The editing mode is also exited automatically if no editing occurs for 5 seconds. The event *exitEdit* is received in the *display* component of the digital watch statechart and triggers the transition leading either from state *alarm_ edit* to state *alarm_ display* or from state *time_ edit* to state *time_ display*. When leaving the time editing mode, the event *resumeTime* is broadcasted too, causing the time to start running again from where the manual edition left it.

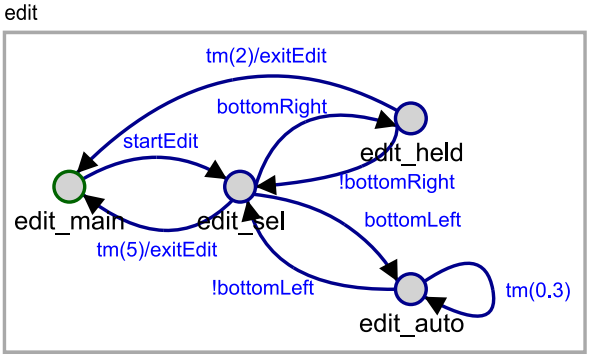


Figure 2.28: *Edit* component of the digital watch statechart.

2.3.7 Indiglo light

As we can observe in Figure 2.29, the default state of the indiglo light is off. When the top right button of the digital watch is pressed the indiglo light is turned on, and it remains on for

as long as the top right button is pressed. When the top right button is released, the *indiglo* component's current state is transferred to *indiglo_ leaving*, in which the component will remain for two seconds and then transfer back to *indiglo_ off*.

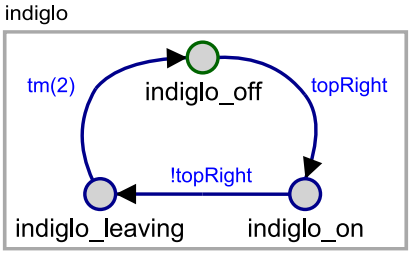


Figure 2.29: *Indiglo* component of the digital watch statechart.

2.3.8 Putting the components together

Now that all the orthogonal components of the digital watch statechart have been individually presented and described, it's time to put them together in order to clearly observe the way they interact with each other by means of event broadcasting. The final statecharts model is captured in Figure 2.30.

2.4 Communicating statecharts through a controller: narrow-cast

Event broadcasting, as we've already seen, is the sending of an event (or output action) to all the orthogonal components of a statechart. A single event can cause multiple transitions to be taken in a statechart such as the one depicted in Figure 2.31. This statechart represents a well known example of a reactive system, a car, as a small group of very simple orthogonal components: the radio, the windshield wipers, the side repeater lights, etc. All of these components function independently from each other, and can be switched/turned on and off indistinctly at any time when the car's started. Some of these components, though, can also be switched/turned on and off even when the car's stopped, such as the radio or the side repeater lights. While in states *engine_ running* or *engine_ on*, if the car is turned off the *engine* orthogonal component is transferred to state *engine_ off*, and the event *turn_ off* is broadcasted to the statechart. As we can see in Figure 2.31, this event triggers several transitions in the car statechart: if the lights were on, they are turned off; if the windshield wipers were switched on, they're switched off; if the air conditioning was switched on, it's switched off. But the radio and the side repeater lights remain on if they were, for they don't depend on the car being started to function but on the key being introduced in the car, therefore it is possible to listen to the radio when the car's stopped, or we can leave the side repeater lights on while the car's parked in an inappropriate place.

We can take advantage of event broadcasting when defining models such as the one captured in Figure 2.31, which is composed of several orthogonal components that require to interact with one another. AToM^3's SCC works like a charm for DCharts, providing us with orthogonality, state composition, history, etc., but what if we want to define our models using the Class Diagrams formalism? The models are then defined as classes -each with its attributes and methods- that are in some way related. The class's behavior is defined by a statechart, but our creativity is limited: AToM^3's compiler for Class Diagrams is very simple and it doesn't support

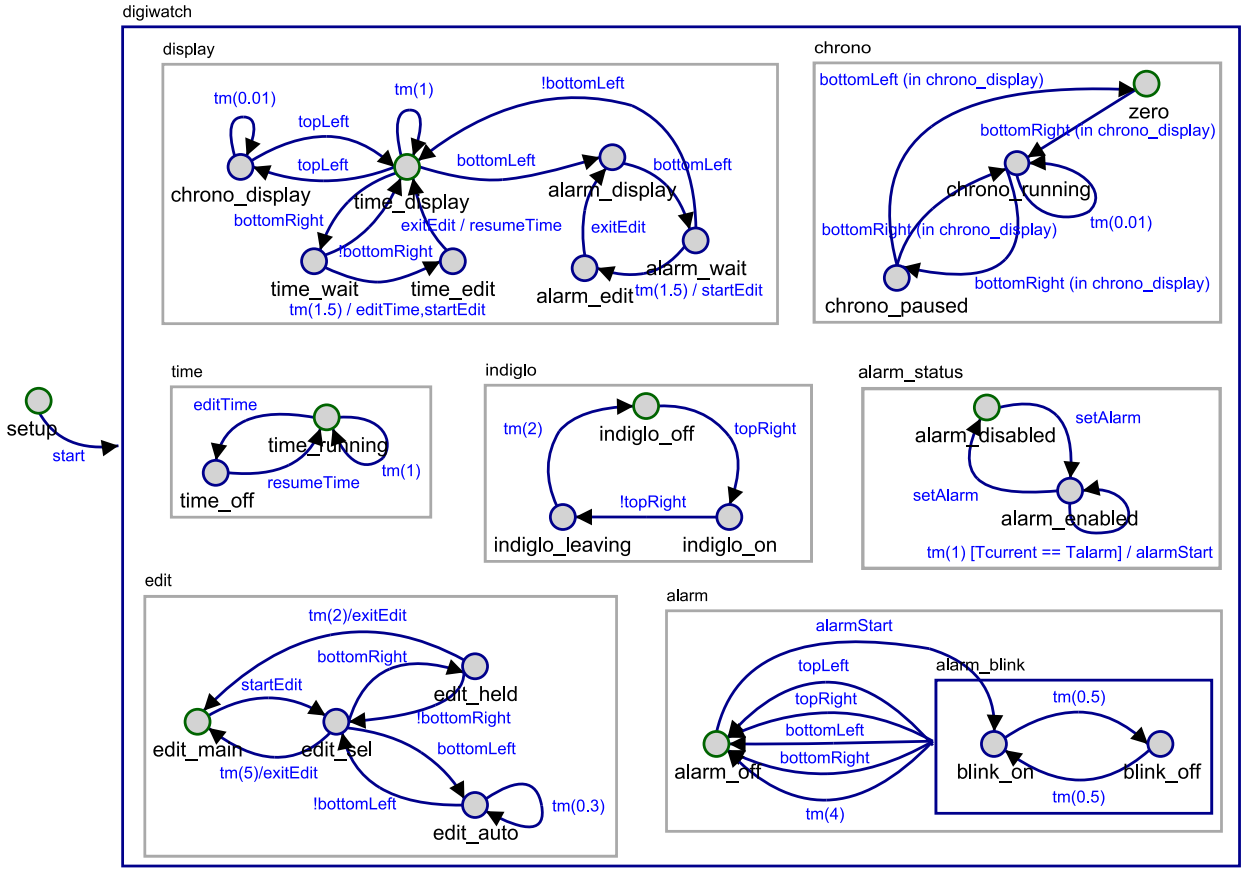


Figure 2.30: Complete digital watch statechart.

orthogonality, nor state composition, nor any of the statecharts features that the DCharts formalism provides us with. Hence, if we were to model a reactive system using Class Diagrams instead of DCharts, event broadcasting was not an available feature and therefore we had to come up with a solution to this problem.

The car, as a reactive system, can be modelled using the Class Diagrams formalism. The car components, which were orthogonal components in our DCharts model, are here defined as classes. Still, we reuse the components's statecharts in the definition of each class's behavior, and we define some attributes and methods for each class. Figure 2.32 captures the car components's classes. The classes can be related, but the orthogonality property of statecharts is lost, and events/actions generated in any of the classes's statecharts are not broadcasted, thus they can trigger no transitions other each class's own. How then can we get the car components to interact with one another?

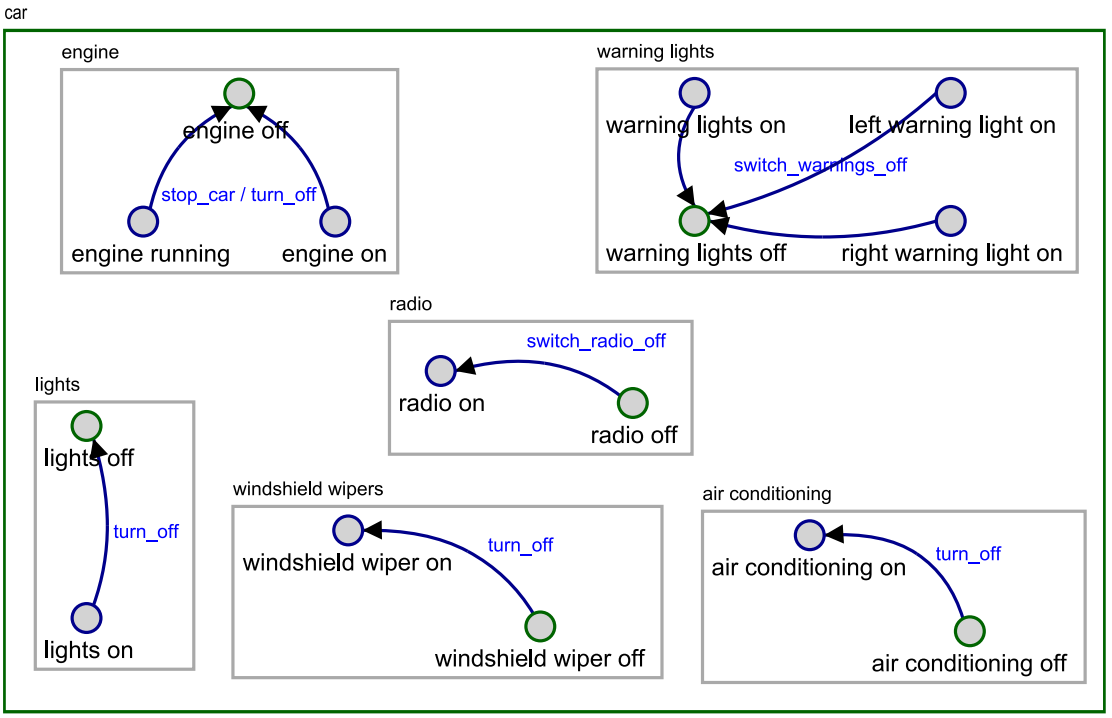


Figure 2.31: Simple statechart of a car, a reactive system.

When compiled, AToM^3 will generate Python code from our Class Diagrams model; the generated source file contains the definition of several independent classes, and this is where we introduce the concept of the *Controller*, which we could also call "composite" or "container" class. In this case, the composition of all the car components's classes is, we could say, the Car class itself. Therefore, an instance of each of the components's classes is declared as an attribute of the controller class, which acts as the link between them. The following is an example of how we'd define the car controller class, assuming that the Python source file generated by AToM^3 is called `CarComponents.py`.

```
import CarComponents
```

```
class CarController:
      def __init__(self):
          # definition of the controller's attributes
          self.engine = Engine()
          self.srepeaters = SRepeaters()
          self.lights = Lights()
          self.wwipers = WWipers()
          self.aircond = AirCond()
          self.radio = Radio()

          # initialization of the classes's statecharts
          self.engine.initModel()
          self.engine.event("start", self)

          self.srepeaters.initModel()
          self.srepeaters.event("start", self)

          self.lights.initModel()
          self.lights.event("start", self)
          ...
```

The methods *initModel()* and *event()* are DCharts's methods generated by the compiler. Whereas *initModel()* starts the statechart's execution, transferring the system to the statechart's default state/s, the method *event()* as the name suggests sends an event to the statechart -possibly triggering one (or more) of its transitions-, and allows the passing of a parameter. Thanks to this methods we're able to implement a solution to the lack of event broadcasting, what we've called *narrow-cast*. The idea is simple and clear: an event is originated in one of the components's statecharts, or through a system external action, and the controller broadcasts the event to all its components/attributes, one by one. The obvious question that arises now is, how does the controller know that an event has been generated in some of its components's statecharts? The trick to this is that the controller doesn't really detect the event generation in any of its components; it's actually the components which broadcast their events by means of the controller. Figure 2.33 captures the actual definition of a class behavior statechart: the default state is *init*, and the triggering of the event *start* will transfer the statechart to composite state *engine*. It has been previously explained that, when designing dcharts with AToM^3, we can specify an enter action and an exit action for every state, as well as we can assign output actions to events. The following code is attached to the event *start* in the statechart depicted in Figure 2.33, and hence it's executed when the event is triggered.

```
print 'start event'
# we collect the event parameter in a variable
ctl=[PARAMS]
```

`[PARAMS]` is a reserved word of the DCharts formalism syntax, and it's the object passed as a parameter to the method *event()*. The parameter is collected in this code piece by the variable `ctl` (short for "controller"). As we've seen in the previous code piece, the event *start* is sent to each of the controller class's components/attributes by means of the method *event()*, and

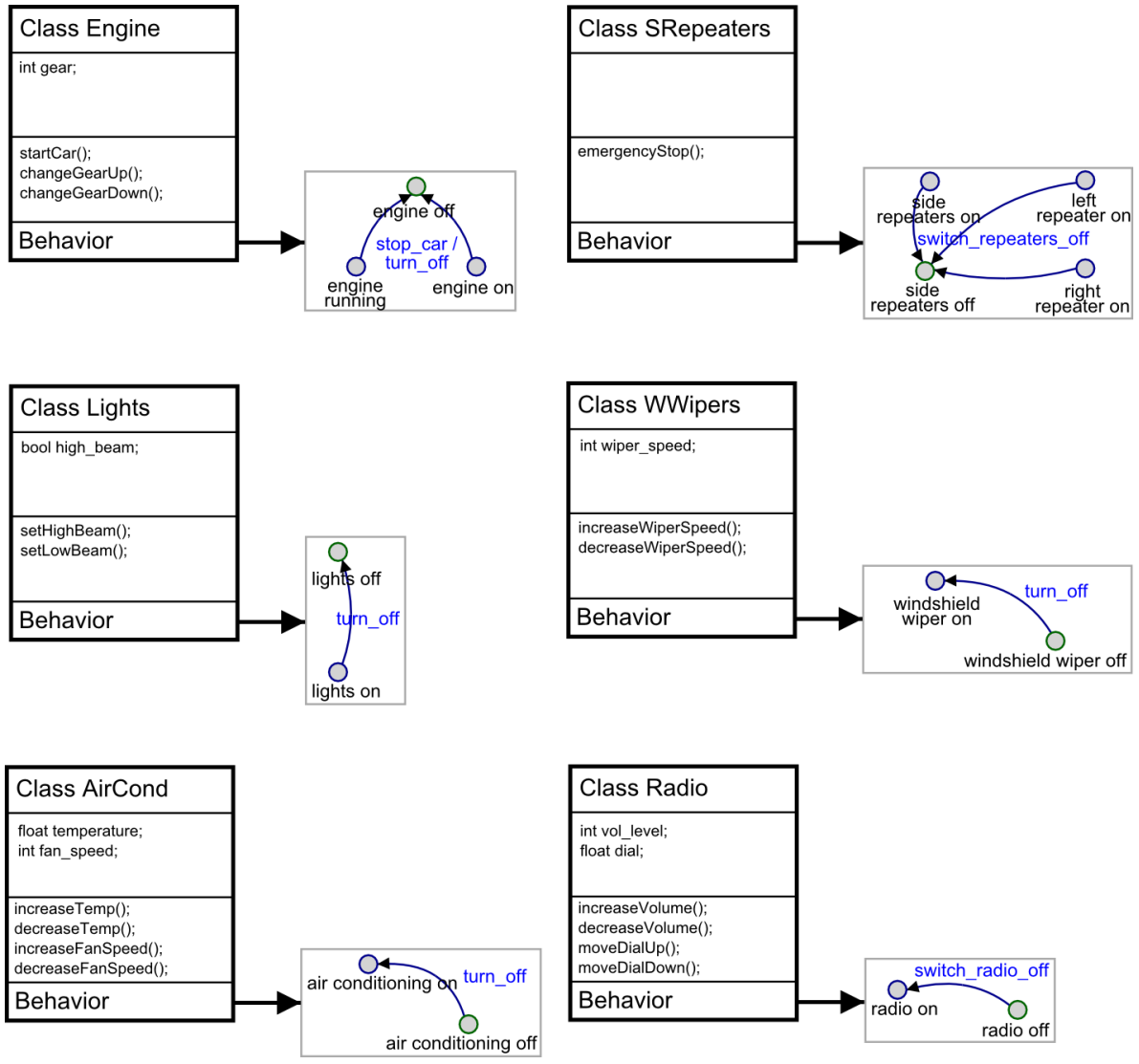


Figure 2.32: Car components defined using the Class Diagrams formalism.

the controller itself is passed along as a parameter. By doing this, we give each component statechart of the controller class a reference to it, and therefore they can access the controller's methods and attributes too. Even though this solution entails creating cyclical references, it's no problem since Python allows them, whereas with other programming languages it would not be possible.

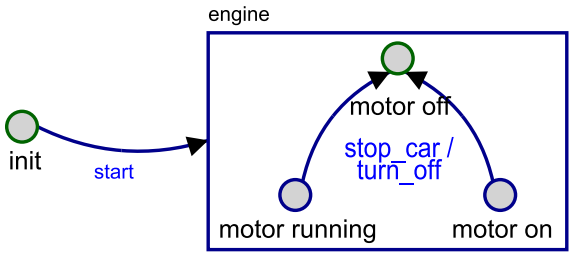


Figure 2.33: Actual definition of the Engine class's behavior statechart

We'll assume now that, among the *CarController* class's methods there is one called *engineTurnOff*, and that it's defined as follows.

```
class CarController:
      def __init__(self):
          ...

      def engineTurnOff(self):
          self.lights.event(“turn_off”, self)
          self.wwipers.event(“turn_off”, self)
          self.aircond.event(“turn_off”, self)
      ...
```

Back to the statechart captured in Figure 2.33, if the event *stop_car* is triggered while the system's current state is either *engine running* or *engine on*, it is transferred to state *engine off* and the event *turn_off* is generated as an output action. The following code, as well, is part of the output action and therefore it is executed too.

```
print 'stop_car event is generated'
# we call a method of the controller class
# to broadcast the output event turn_off
ctl.engineTurnOff()
```

As indicated in the code, the Engine class's behavior statechart generates the output event *turn_off*, and calls the method of the controller that will broadcast it to the rest of the controller's components. This whole process of event generation and broadcast at component's statechart level by means of the controller is what we've called *narrow-cast*. We'll delve deeper in the *narrow-cast* concept in the next chapter: *Statecharts modelling of a computer-controlled game character behavior*.

3

Statecharts modelling of a computer-controlled game character behavior

Introduction

In the late 70's, Atari Inc. released an arcade game called *Breakout* [23], the game concept of which resembled very much that of the earlier *Pong*, also by Atari Inc. *Breakout* would become such a great success that, some years later, it would be ported to video game consoles under the more popularly known name *Arkanoid*. The objective in *Breakout* is to break a brick wall on top of the screen by using a ball. The player controls the ball's direction by making it bounce against a small platform that moves horizontally in the bottom of the screen, while preventing the ball from falling off the screen.

The sensation in the early 80's was another arcade game called *Pac-Man* [24], released by the Japanese Namco. The *Pac-Man* is a character resembling a pizza that's missing a slice, and the player controls it through a maze. The objective is to make the Pac-Man eat all the white pellets in the maze while avoiding the ghosts that chase him.

In 1998, Konami set a precedent in modern video games with *Metal Gear Solid* [25]. In this game, the main character has to infiltrate a nuclear weapon facility to neutralize a terrorist threat. The armed guards patrolling the many game's areas will be alerted by such things as suspicious noises or footprints in the snow, then leave their routines and start looking for the intruder.

More than 30 years after *Breakout*, it can be quite hard to tell the difference between a movie and a video game. While in *Breakout* neither the ball or the bricks behave according to any artificial intelligence whatsoever, and there is some in the ghosts chasing *Pac-Man*, both games are very far from the AI in *Metal Gear Solid*. This a very clear example of how extremely fast the video game industry has evolved over such a short period of time.

Whereas, just a few years back, video games were considered to be for kids, its market target covers a wide age range nowadays, leading the video game industry sales during 2007 alone to experience a 43 percent increase over the previous year [26]. With this huge growth and the ever-improving technology in video game consoles comes a big demand for more realistic games, which doesn't only refer to better graphic engines but, and very especially, to better AIs too. The latest trend in multi-player video games are MMORPGs (Massive(ly) Multiplayer Online Role-Playing Games), in which there is a large amount of the so-called NPCs (Non-Player Characters), computer-controlled characters that interact with the player-controlled characters by providing information, items, or some kind of aid during battles. To make a MMORPG

the most realistic as possible, one would expect the NPCs to behave as a real person would in situations similar to those happening in the game, or as closely as possible.

As video games grow in complexity, it gets more and more difficult and takes more and more time to develop such complex AIs and, most of the time, AIs are completely coded from scratch for every new video game being developed. It seems clear then that, in order to develop video games faster, developers should be able to reuse parts of the AI codes written previously for other video games. This necessity for reusability brings along a necessity for modularity. An AI's architecture can be assembled from small components and, the more and smaller these components are, the less effort will be required to adapt and reuse them in assembling other AIs, even if those are designed for video games of very different genres.

This is the starting point for the proposal of the paper *Model-based Design of Computer-Controlled Game Character Behavior* [2], on which the research work done for this Master Thesis is originally based. This chapter presents an introduction to that paper, as well as an in-depth look at the project developed to better comprehend the paper's approach and to take it further.

3.1 Related work

The authors of the paper *Model-based Design of Computer-Controlled Game Character Behavior* [2], state that AI specification should be done at the appropriate level of abstraction, using the appropriate language. They also state that AIs should be modeled instead of coded, so that the modelers don't necessarily have to know any programming languages, and thus it's easier for them to abstract from the implementation issues and focus on the logic of the model. The modelling formalism should be chosen based on the architecture of the model, an AI in this case.

An AI is abstracted as a reactive system: it will receive some input and react to it by generating some output. Using the game *Metal Gear Solid*, previously mentioned in the introduction, as an example, the guard patrolling an area will discover a group of suspicious footprints in the snow (input) and react to this event by alerting other guards or start looking for an intruder (output). Thus, in order to model an AI we'd be looking for a formalism that

- is state/event-based,
- describes autonomous/reactive behavior,
- allows modularity,
- is well known and there is availability of tools to work with

To demonstrate the approach of the paper, the chosen formalism is a variant of Rhapsody Statecharts, and the AI to be modeled that of a tank to be inserted in the EA (Electronic Arts) Tank Wars main game loop.

EA Tank Wars is an AI programming competition the objective of which is to *"develop an AI for a tank that lives in a 2D world. This AI must manage the internal resources for the tank (health, fuel), find the enemy tank, map out the world, and destroy the enemy tank"* [3]. The Tank Wars game environment is provided by EA and programmed in C++. The contestants must link their own code to the EA Tank Wars game code through the function `ai(const TankAIInput in, TankAIInstructions & out)`.

The tank's AI architecture is modularly defined at different levels of abstraction, being the lowest ones the input and output components. Originally, in the paper's approach, each component's

structure is modeled with a class and its behavior with a statechart, using the Class Diagrams formalism. Initially, those statecharts were very detailed and described a high level of interaction between components since, obviously, these must be orthogonal (otherwise, for example, the tank could not move and turn its turret at the same time). The final statechart designs, those created to run in the EA Tank Wars environment, are a lot simpler though, due to limitations of the compiler used to generate the C++ code from the models, which didn't support statechart orthogonality by the time the paper was written, but also, and very importantly, because of the difficulties that would have to be faced, and the problems that could arise, in the process of bridging the event-based world of the statecharts formalism with the time-based world of the EA Tank Wars game. Once compiled the classes and written the necessary bridging code in the EA Tank Wars environment, some successful simulations were made, proving that the tank behaves according to its modeled AI and as expected for the EA Tank Wars game.

NOTE: By the time we were over and done with this part of the Master Thesis, and we were already working on *Statecharts modelling of a robot's behavior*, Reehan Shaikh, one of our colleagues at the MSDL [1] of McGill University, finished building the new, greatly improved version of the Class Diagrams compiler for AToM^3, which currently allows statecharts orthogonality, composition, history, etc.

3.2 The Tank Wars simulation project

As a way of better understanding the tank's layered AI architecture and the statechart formalism itself, further research work was performed on the paper's [2] approach. The main goal was to focus on building the complex models depicted in the paper, instead of building models that could be easily inserted in the EA Tank Wars main loop.

The main reason for this is that a lot of extra work is required after compiling the models, to make them appropriate to be run in the EA Tank Wars game. First of all, the models must be compiled into fit-for-Windows C++ code (since the EA Tank Wars game is a Microsoft Visual Studio project). Secondly, the compiled code must be somehow inserted into the EA Tank Wars project, which requires some additional code typing in order to bridge the gap between the event-based world of our statecharts and the time-based world of the EA Tank Wars game. Since making the tank's models run in the EA Tank Wars game wasn't in the scope of this project, we decided to build our own extremely simple simulation environment to test the models. We used Python code to describe the actions in our statecharts, and thus we compiled them into Python and built our simulation environment using Python's Tkinter. The development of this simulation environment is covered in section 3.7.

Originally, it was our intention to be as faithful to the paper's designs but as the project was being developed the models evolved. Some of them were modified, some others were ruled out. We decided to use only statecharts instead of class diagrams. In the following sections, we'll provide and in-depth look at the original models and at the definitive, actual models.

3.3 Tank's AI architecture and its components

What we have been calling "the tank's AI" so far is actually not such thing; what we are modelling is in reality the tank pilot's AI. We're not interested in the physics of the tank as a heavily armored vehicle, but in the behavior of the tank pilot. As well, the interaction with the environment is taken care of by the EA Tank Wars game code, by providing input to the AI every 50ms. Therefore, as the AI developers, we can model at a higher level of abstraction, focusing only on the behavior of our model.

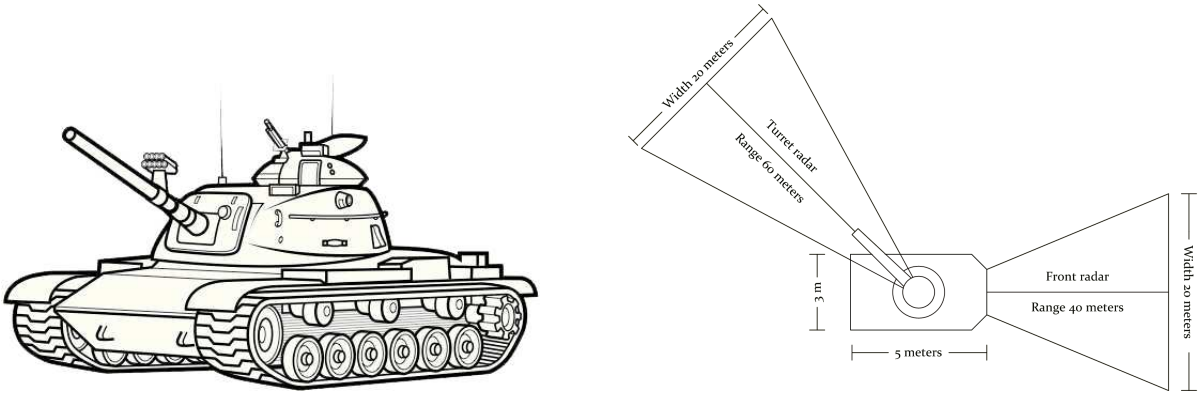


Figure 3.1: Abstraction of the tank and its components [2].

The tank pilot makes decisions based upon the information he/she gets from the environment and the state of the tank itself: have we been spotted by the enemy? Have WE spotted the enemy? What's the fuel level of the tank? How much more damage can the tank sustain? Thus, the AI's architecture is divided in different layers, at different levels of abstraction. At the lowest level the input information is received from the environment and is conveniently filtered to the higher levels, where the decision is made (by the tank pilot). Next, the decision is translated into an action to be carried out by the tank, again at the lowest level of abstraction, and some output is given to the environment.

The lowest levels of abstraction are also where the physical components of the tank are located. Our tank is abstracted as a rectangular box with a gun mounted on a rotating turret anchored in the middle of the box. The tank perceives the environment through its sensors, and reacts to it through actions, or actuators. The transformation of the input received through the sensors into the output given through the actuators is described by a set components, each of which is covered in detail in the following subsections. The event flow depicted in Figure 3.2 describes the order of this transformation process.

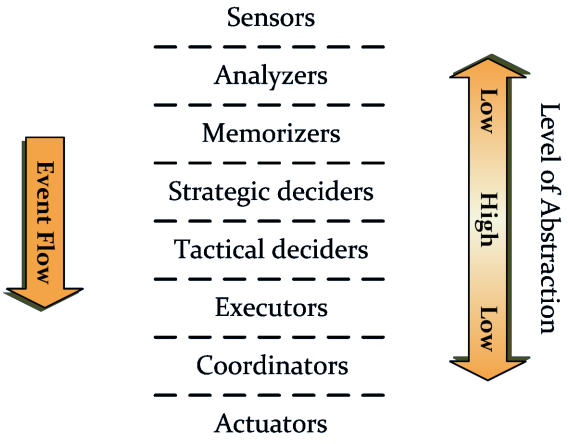


Figure 3.2: Layered AI architecture [2].

3.3.1 Sensors

We can distinguish two kinds of sensors in our AI: those detecting and keeping track of the events that are external to the tank, and those managing the events that are internal to the tank. The internal sensors are in charge of keeping the state of the tank updated.

There is a set of variables that describe the state of the tank at any point in time during the game or simulation. Some of these variables are the fuel level, the damage level and the current position in the world map (given by X and Y coordinates). As the game evolves, so does the state of the tank. The changes in the values of these variables are detected through the internal sensors, and the state of the tank is updated. One of these internal sensors is the *FuelTank* class (see Figure 3.3).

At the beginning of the simulation, the fuel tank will be at its full capacity, and the *FuelTank*'s statechart in the *FuelLevelOk* state. Fuel is consumed every time the tank moves or turns, and when the turret is turned too. Whenever the fuel level drops below a predefined critic value, the state in *MonitorTank* will change to *FuelLow* and a *fuelLow* event will be generated. This event might trigger some transitions in other components of the tank, with the objective of forcing it to abandon its current course of action and start looking for the refuel station instead. The external sensors are the tank's radars: the front radar and the turret radar. The radars can detect the presence of surrounding obstacles in the map terrain and the position of the enemy. Once the enemy is spotted, the turret is turned in order to aim at the enemy's position before shooting. The front radar of the tank has a range of 40 meters and covers 20 meters in width. The turret radar has the same width coverage but a range of 60 meters.

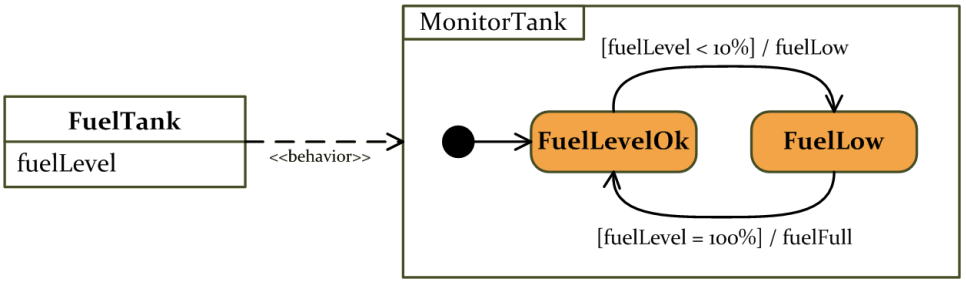


Figure 3.3: Checking and announcing the fuel level of the tank [2].

The input data received from the environment will give the pilot information on what's been detected by both radars, whether the tank has been hit by an enemy missile or not, etc. Ultimately, all this information will take part in the decision making process, but the different components at each level of abstraction need to dispose of it in a different way. That's why it must be filtered and spread from the sensor components, at the lowest level of abstraction, to the upper layers of the AI architecture.

Figure 3.4 depicts the statecharts modeled behavior of the radars. The upper orthogonal component of the statechart is in charge of generating events upon detecting enemy presence in the range of the corresponding radar. The event *enemySighted* might make the tank pilot change the current course of action to aiming at and shooting the enemy. The lower orthogonal component of the statechart is in charge of generating events upon detecting obstacles in the tank's path or its surroundings. The tank's own map of the world will then be updated with new information, making it more precise and accurate as the simulation evolves.

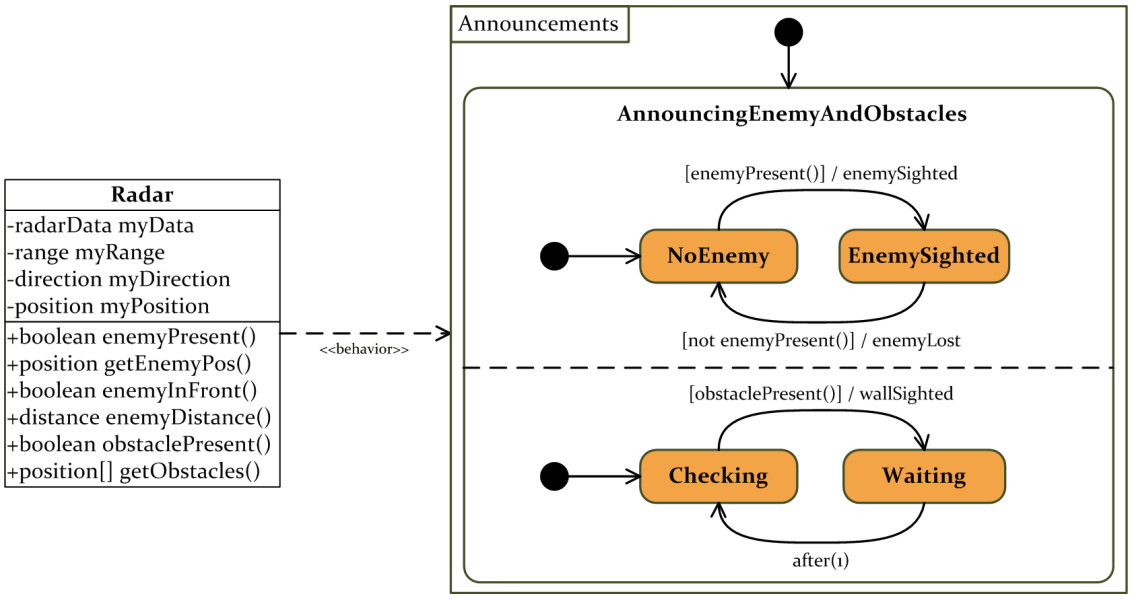


Figure 3.4: Announcing detection of enemy and/or obstacles through the radars [2].

3.3.2 Analyzers

The events previously generated by the sensors will cause the generation of more events at other levels of the tank's AI architecture and so on, consecutively. As well, some events will be generated upon correlating the information provided by events coming from components at very different levels of the AI architecture.

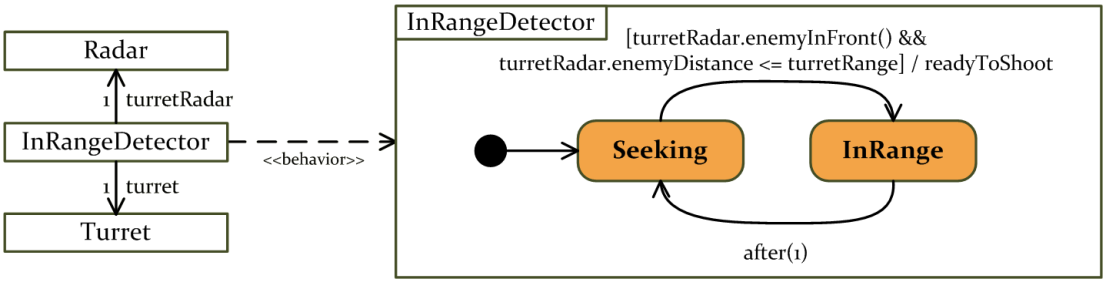


Figure 3.5: Correlating sensor events [2].

The function of the analyzers is to correlate the information gathered by the sensors with the state of several tank components in order to generate new events. When the event *enemySighted* is received, the *InRangeDetector* component of the tank will call the function `enemyInFront()` from the turret radar. If the turret is not facing in the direction of the enemy, it will have to be turned to aim at the enemy. But, even if the enemy is in front of the turret, it might be out of the turret's range; this condition must also be satisfied. If the enemy is both in front of the turret and in the turret's range, the event *readyToShoot* will be generated. This behavior is depicted in the *InRangeDetector* statechart, in Figure 3.5.

3.3.3 Memorizers

Going back to realistic video games nowadays, there are certain features that one shall expect from the enemies behavior. For example, the enemies should run for cover when the user-controlled character's bullets start raining on them. And, even more importantly, the enemies should remember the patterns or strategies of previous attacks. With this information, the enemy AIs shall be able to come up with a better attack/defense strategy of their own, thus making it more difficult for the player to advance.

The tank pilot is also expected to make decisions based on the knowledge of events occurred previously during the simulation. This is possible thanks to the memorizers. Even when the enemy is not in the range of the tank's radars anymore, the memorizers will remember at what position in the map the enemy was last spotted and, therefore, the tank pilot will start looking for it there. As shown in Figure 3.6, the enemy's position is stored as a coordinate in the variable `enemyPosition` of the *EnemyTracker* class. The operation `enemyMoved()` compares the current position of the enemy (if it's still in the range of at least one the tank's radars) with its previously known position, stored in the variable `enemyPosition`. If both positions differ, the value stored in `enemyPosition` is updated and the event *enemyPosChanged* is generated.

One of the objectives of the game, apart from destroying the enemy tank, is to collect as much information of the world layout as possible during the running time of the simulation. Therefore, each tank is supposed to map out the world as accurately as possible. The tank's perception of the world map is like a blank canvas at the beginning of the simulation. As the game evolves, the tank keeps running into obstacles (namely walls but, in a very detailed map, could be any kind of geographical features). These obstacles are detected through the radars and, upon doing so, the event *wallSighted* is generated in any of the radar's statecharts. These event will trigger the only transition in the *UpdateMap* statechart and, as a result, the function `updateMap()` of the *ObstacleMap* class will be called (see Figure 3.7). This function will update the tank's own map of the world with the new data.

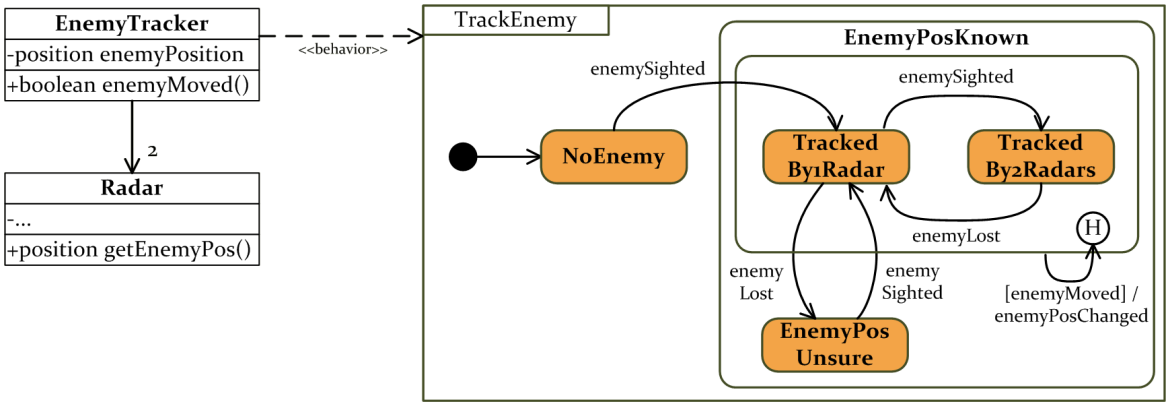


Figure 3.6: Tracking the enemy's last known position [2].

Having an accurate knowledge of the map's layout is very important. The tank doesn't move around the world randomly, but pinpointing a destination on the map, then calculating and following an indefinite number of waypoints along the shortest path to the destination. If there are undiscovered obstacles in the path of the tank, the pilot will have to make several detours,

thus wasting time and fuel in reaching the destination. The next time the tank pilot decides to go through the same region of the map, the obstacles will be avoided during the path calculation, thanks to the knowledge of their positions.

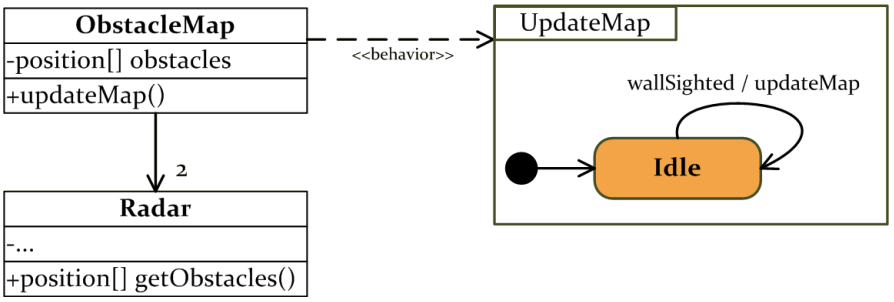


Figure 3.7: Mapping out the world [2].

3.3.4 Strategic deciders

These components are at the highest level of abstraction of our tank's AI architecture, and this is where the decision making process takes place, based on the information gathered in the previous, lower levels. The pilot's strategy will depend on many factors and, while the final goal is to destroy the enemy tank, this will not always be the pilot's highest priority.

As depicted in the *PilotStrategy* statechart, in Figure 3.8, when the simulation starts the pilot will be in *Exploring* mode, not only looking for the enemy but also reconnoitering the world's terrain, mapping out its obstacles and looking for the refuel station and the repair station. Whenever the enemy is spotted and tracked by at least one of the radars, and the tank still has enough fuel, the pilot will change operation mode to *Attacking*. While in this operation mode, the sole objective of the tank is to aim at and shoot the enemy.

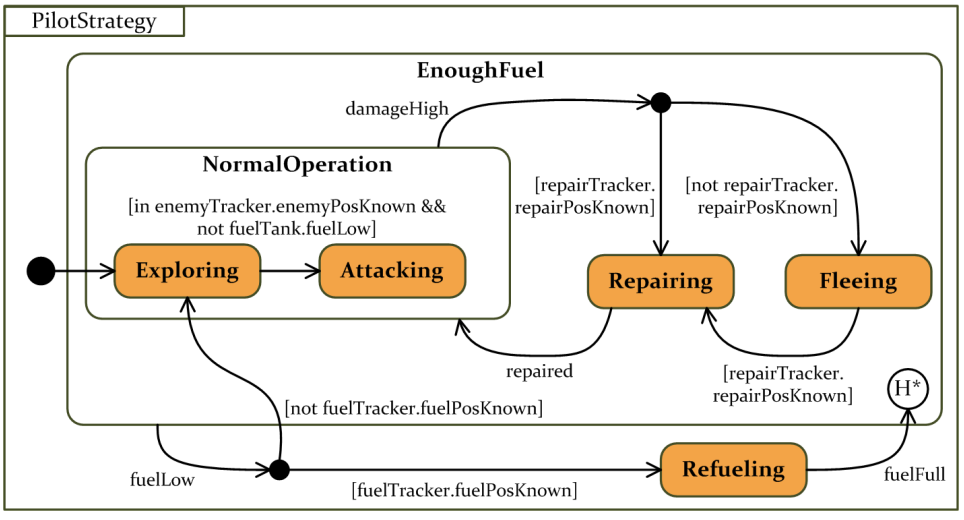


Figure 3.8: Deciding which strategy to follow [2].

If, at any point during the simulation, and being the tank either in*Exploring* or *Attacking*

operating modes, the damage sustained by the tank is too high, the pilot will choose to escape from the enemy's range by switching to *Fleeing* operating mode. Furthermore, if the location of the repair station is known, the pilot will switch the operating mode to *Repairing* instead, and head to the repair station. Therefore, logically, and according to these behavior patterns, it's clear that the self-survival of the tank is a priority over the destruction of the enemy.

But even more importantly than keeping the tank alive is to keep it fueled, since fuel is essential for the tank to function; therefore, running out of fuel is tantamount to being destroyed and losing the game. So it doesn't matter what is the current operating mode of the tank whenever the *fuelLow* event is received, the pilot should switch it to *Refueling*, if the location of the refuel station is known. Otherwise, the best option is to keep or switch back to *Exploring*, hoping to find the refuel station soon. After refueling, the event *fuelFull* will be generated and the tank pilot will switch back to whatever was the operating mode before the *fuelLow* event was received.

The strategic deciders are the components at the highest layer, and they represent the inflection point of this transformation process, where decisions are made. Whereas every component has its own objective, the strategic deciders, as we've seen, must decide on a high-level goal.

3.3.5 Tactical deciders

As we've seen, the process of transforming inputs into outputs in the Tank Wars environment begins at the lowest level of abstraction of our AI architecture. Data, in the form of events, flows from the sensors to the upper layers of the architecture. Once the tank pilot's goal is decided, it's the task of the tactical deciders to plan how to achieve that goal. The high-level goals sent by the pilot strategy component have to be translated into lower-level commands that can be understood by the different actuators of the tank, such as the motor and the turret. The planning should as well take into account the history of the game, i.e. consult the memorizers for important game state or events that happened in the past. For example, if the current pilot's strategy is to refuel, the *UpdateMap* component should be consulted in order to find out if the refuel station's position is known, so that the most direct path to it could be traced.

Each strategy of the pilot should have a corresponding planner component. Figure 3.9 illustrates how the *AttackPlanner* decides to carry out an attack: whenever the tank is ready to shoot, a shoot event is sent. The movement strategy is also simple: the planner chooses to move the tank to the position of the enemy. Whenever the enemy position changes, it sends out a *newDestination* event.

The *Pathfinding* component, shown in Figure 3.10, knows how to perform obstacle avoidance. It translates the newDestination event into a list of waypoints by analyzing the current world information obtained from the *Map*. The *Pathfinding* component then announces the first waypoint by sending an event. Whenever the tank reaches a waypoint, the next waypoint is announced.

3.3.6 Executors

The executors map the decisions of the tactical deciders to events that the actuators can understand. The mapping of events is constrained by the rules of the game or simulation. There is typically one executor for each actuator. In our case the *Steering* component shown in Figure 3.11 translates the waypoints into events that the *MotorControl* understands. Every second, depending on whether the waypoint is ahead of, left of, right of or behind the tank, the corresponding command event is sent to the *MotorControl* component.

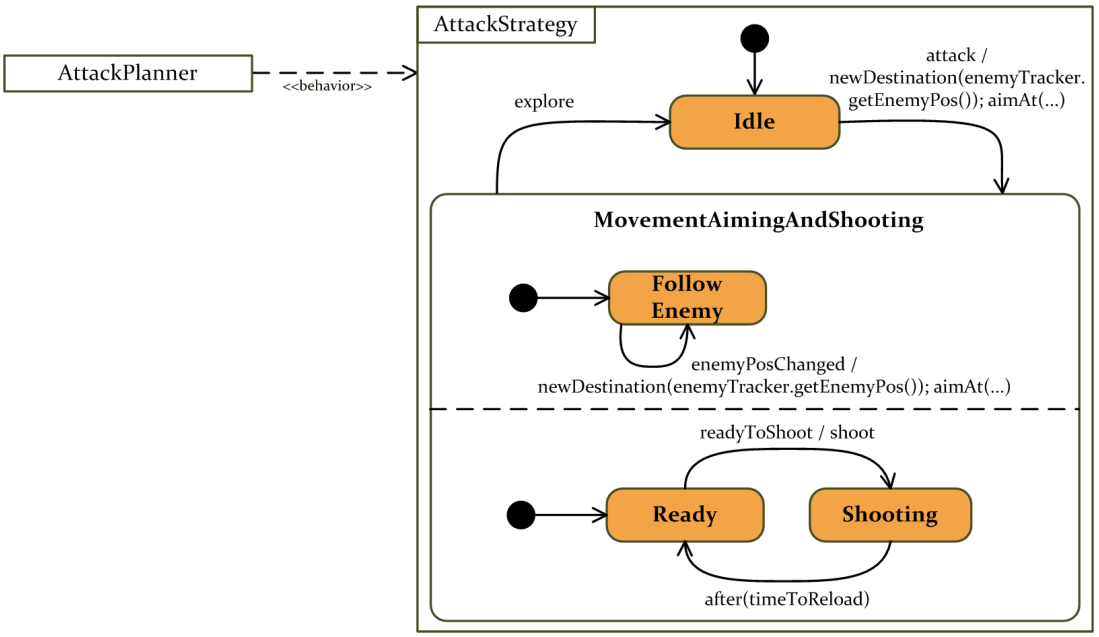


Figure 3.9: Deciding the tank's next attack move [2].

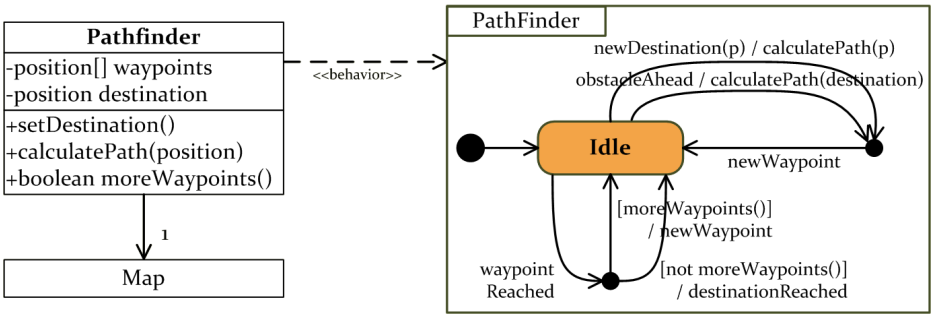


Figure 3.10: Tracing a path to a destination [2].

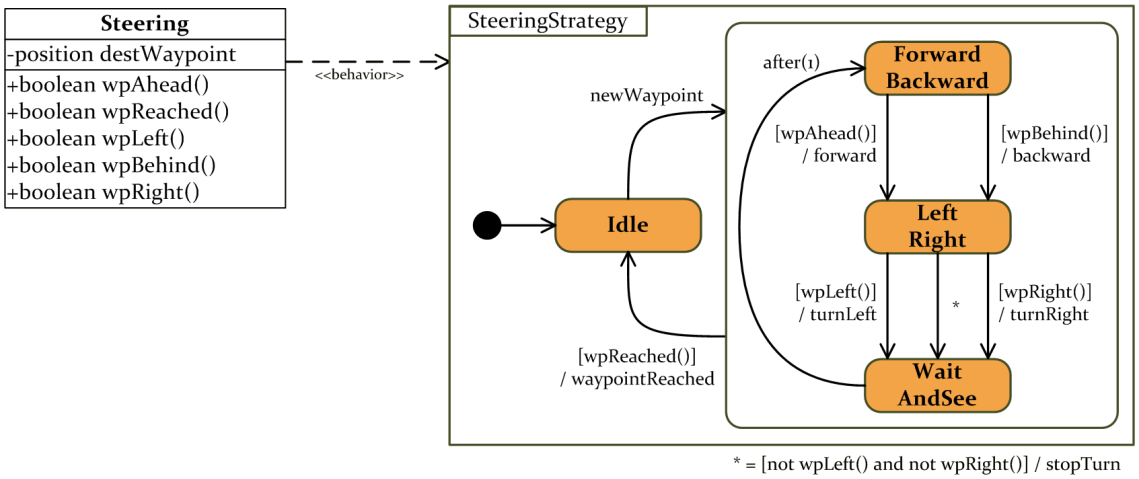


Figure 3.11: Moving the tank one waypoint at a time [2].

3.3.7 Coordinators

For modularity and composability reasons, executors individually map tactical decisions to actuator events. This mapping can result in inefficient and maybe even incorrect behavior when the effects of actuator actions are correlated. In such a case it is important to add an additional coordinator component that deals with this issue. For example, while attacking, the turret should turn until it is facing the enemy tank and then shoot. However, the optimal turning strategy depends on whether the tank itself is also turning or not. Figure 3.12 illustrates a *Turret-TankMovementCoordinator* class that performs this coordination step. The calculations required to determine if it is faster to turn right or turn left based on the current turning decisions of the motor are done in the operations `reachTurnLeft()` and `reachTurnRight()`.

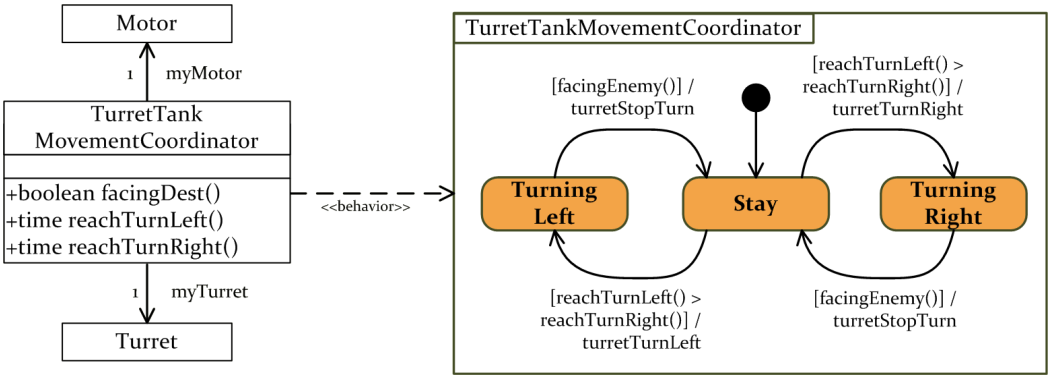


Figure 3.12: Coordinating and controlling the turret and the tank [2].

3.3.8 Actuators

At our level of abstraction, the tank actuators are very simple. A tank pilot can decide whether to advance or move the tank backwards at different speeds, and whether to turn left or right. Likewise, a turret can be turned left or right, and shots can be fired at different distances.

Finally, commands can be given to refuel or repair, if the tank is currently located at a fuel or repair station. We suggest to model each actuator as a separate Control class. Figure 3.13 shows the *MotorControl* class, an actuator that controls the movement of the tank. The state diagram shows how the motor reacts to *turnLeft, turnRight, stopTurn, forward, backward* and *stop* events. How this action is finally executed within the game or simulation is going to be discussed in the following sections.

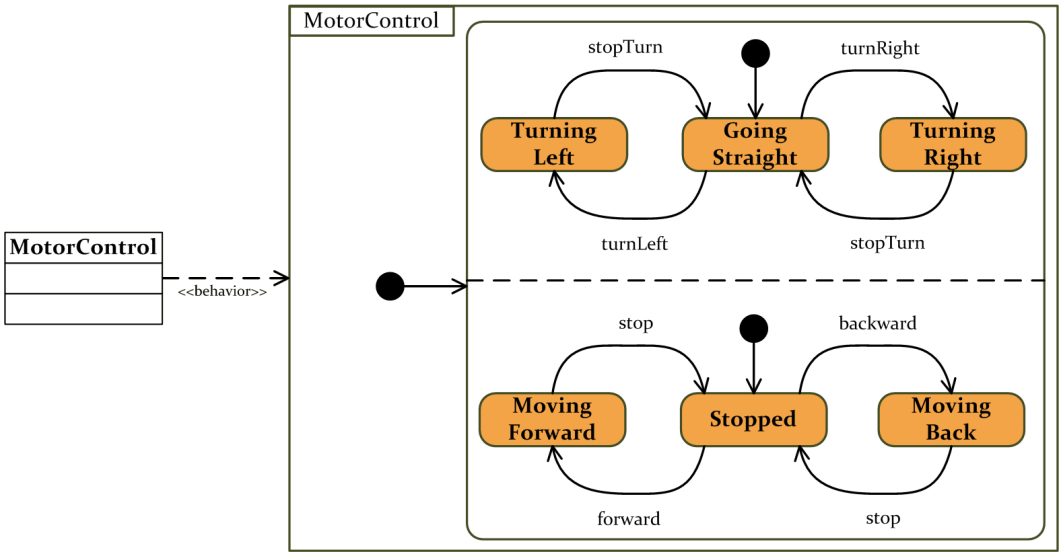


Figure 3.13: The motor actuator [2].

3.4 Solving the event-based Vs. time-based problem

In the original class diagrams/statecharts designs most of the transitions have guards, which means that, in addition to the trigger event, there is a condition that must evaluate to true in order to enable the transition to be taken. It's also possible to have guard-triggered only transitions, so that the guard is not an addition to the trigger event but completely replaces it. When a state S is entered, all of its outgoing transitions are checked and the guards, if there is any, are evaluated. The guards are only evaluated again whenever an event triggering any of this S state transitions is fired. If an event E is fired, thus triggering a transition T with a guard G, and the guard G evaluates to false, the transition is not taken. The transition T won't be checked again until the next time the event E is fired; if, in the meantime, the guard G changes its value, thus evaluating to true, when the event E is fired again the transition T will be taken. The problem of having guard-triggered only transitions, as we did in many of our original statecharts designs, is that there are no events triggering the guards evaluations and, therefore, the guards are only checked once, that is when the origin state is entered. So if we wanted to use unlabeled "eventless" transitions we had to find a way to force periodic guards evaluations in order for the transitions to be taken eventually. Of course, the only possible solution implied using time.

We have used *timed self-transitions* in the modelling of all the AI components's statecharts discussed in section 3.5. Imagine a state S which has one unlabeled outgoing transition and a timed self-transition which is labeled *after(X)*; the timed self-transition forces the system to

re-enter state S every X seconds, thus causing an evaluation of the guard of S's unlabeled transition. Once state S is entered, its unlabeled transition is automatically checked and its corresponding guard is evaluated. Since the transition in question is "eventless", it will be taken immediately if the guard evaluates to true. On the contrary, if the guard evaluates to false, the time counter starts running and, when it reaches X, the timed self-transition will be triggered. State S will be re-entered and the whole process will start all over again. In order to define this timed self-transitions we've used the `AFTER` macro of AToM^3's SVM.

Since our purpose is to ultimately build a simulation application in which to test our statecharts models, we are to develop a very simple simulation environment. The simulation application will consist of a static part, the GUI, and a dynamic part, our compiled statecharts models. We'll define a set of methods in the static part which will update the GUI as the simulation develops. These methods are called at the lowest level of abstraction of our AI; for example, from the *MotorControl* component of the tank -which is discussed further on in the following section-.

Another reason why we need timed self-transitions is the simulation of the tank's movement. Let's assume there is a method in the static part of the application that executes the tank's movement, and it's called *moveForward()*. This method is called when the state *MovingForward* of the *MotorControl* statechart is entered (see Figure 3.22). If the method *moveForward()* caused the tank to move forward indefinitely, it should as well consume a unit of fuel every 's' seconds or every 'm' traveled meters -depending on how we decide to manage the fuel-. Therefore, and assuming we'd choose the easy way of implementing this, we'd need some kind of loop with a time function such as *sleep()*; this is already getting quite complex. Also, we'd be stuck in the method *moveForward()* until another movement event was raised in the simulation. How would we exit the method then, how would we abort its execution? The easy solution is to use timed self-transitions.

Instead of having methods that cause the virtual tank to move indefinitely in any direction, we define our methods to perform "movement steps". In one movement step the tank consumes one single unit of fuel and travels a certain short distance in the selected direction. Thus, our movement methods, such as *moveForward()*, are executed in just a couple of milliseconds. Again referring to the *MotorControl* statechart captured in Figure 3.22, the method *moveForward()* is called every time the state *MovingForward* is entered, and so we use timed self-transitions to simulate a continuous movement by repeatedly entering the state. If the time constant used to trigger these timed self-transitions is small enough, we can simulate a pretty fluid movement, without abrupt steps.

3.5 The actual AI components

One of our main reasons to develop the Tank Wars simulation project was to prove that we could model statecharts in independent files and then put them together as orthogonal components of the same system. This meant that the statecharts would have to be able to communicate by sending events to each other. Therefore, it was not our intention to implement a very detailed simulation and we realized we could dispense with many of the tank's components. Whereas the original components were modeled using class diagrams, we decided to redesign them by using only statecharts. The fact that we chose to use Python instead of C++ to build our simulation environment also made our work a lot easier.

Even though we dispensed with the class diagrams and used simple statecharts instead we had to keep each class's methods and properties, which had to be "located" somewhere else in our

code. Since our statechart compiler automatically generates a class out of every statechart, each component would become an instance in our tank controller class, which was coded by hand. Therefore, each component's methods are rewritten as methods of our tank controller class, and each component's attributes are declared as public variables of the respective statecharts compiled class, after the instantiation. The following subsections discuss the actual AI components that we used.

The reader will notice that the *tank controller class* is mentioned in repeated occasions in the following subsections. Please concede us the use of it prior to the definition of the concept; it is explained in detail in section 3.6.

3.5.1 *FuelTank* and *HealthTank*

Figures 3.14 and 3.15 capture the statecharts models of the sensor components of the tank. *MINFUEL, MAXFUEL, MINHEALTH*, and *MAXHEALTH* are constants defined in the tank controller class. The tank moves according to the will of the pilot, consuming fuel at a constant rate; the movement of the turret consumes fuel as well. As well, the tank's "health" decreases every time it is hit by enemy fire. The current levels of fuel and health of the tank are controller class's variables, which values are updated by other modules of the tank as the execution develops. But also, the tank movement is not possible if the fuel tank is empty, so the fuel tank monitor must send a "warning" to the other tank modules when the current fuel level is low. As depicted by Figure 3.14, the state *FuelLevelOk* is re-entered every second (although it could be less than that), thus triggering the guard check in the unlabeled transition which transfers state *FuelLevelOk* to state *FuelLow*. If the guard check reveals that the fuel level has reached a level below *MINFUEL*, the event *fuelLow* is narrow-casted to the tank modules that have transitions which might be triggered by it. This is the output action code:

```
ctl.fuelTank.fuelLow = True
print "narrow-casting event: fuelLow"
ctl.pilotStrategy.event("fuelLow")
```

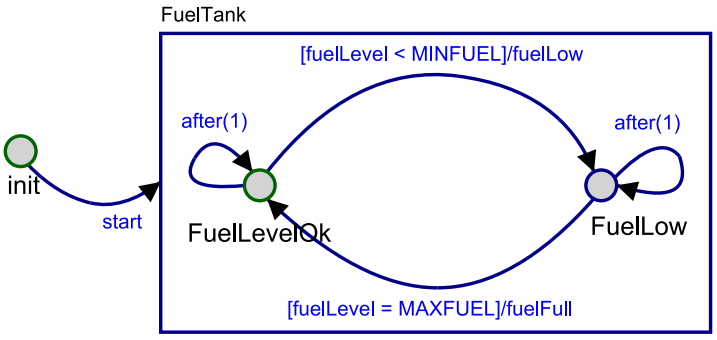


Figure 3.14: *FuelTank* statechart.

Once the fuel tank monitor statechart is transferred to state *FuelLow*, it won't go back to *FuelLevelOk* until the fuel tank is refilled, and that can only happen if the tank pilot finds a fuel station in time. The health monitor works in a very similar way to the fuel tank monitor, as captured in Figure 3.15. The current health level of the tank is checked every one second and, in case it reaches a level below *MINHEALTH*, the event *damageHigh* is narrow-casted.

The tank pilot must then urgently find a repair station, or else the tank could be completely destroyed by the next enemy attack, and the execution would be over as well. The following is the output action code:

```
ctl.healthTank.healthLow = True
print "narrow-casting event: damageHigh"
ctl.pilotStrategy.event("damageHigh")
```

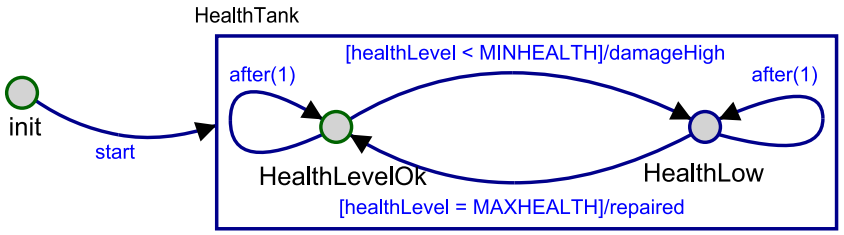


Figure 3.15: *HealthTank* statechart.

3.5.2 *Announcements*

The statecharts model to the tank's *Announcements* module consists of two orthogonal components, the enemy detection component and the obstacle detection component, as captured in Figure 3.16. Timed transitions cause guard checks of the unlabeled transitions transferring the system from states *NoEnemy* and *Checking* to states *EnemySighted* and *Waiting* respectively. *enemyPresent()* is a method of the tank controller class which reads the tank's radars data in order to find out if either an enemy or/and an obstacle have been detected in the perimeter of the tank. If an enemy has been sighted, the corresponding transition is taken generating the event *enemySighted* as an output action. The event is narrow-casted to some of the tank's modules.

```
print "narrow-casting event: enemySighted"
ctl.enemyTracker.event("enemySighted")
```

Whereas, when the enemy is no longer in the range of the tank's radars, the system is transferred back from state *EnemySighted* to state *NoEnemy*, and the following code is executed:

```
print "narrow-casting event: enemyLost"
ctl.enemyTracker.event("enemyLost")
ctl.pilotStrategy.event("enemyLost")
```

The obstacle detection component has a little different behavior. The tank controller class's method *obstaclePresent()* reads the tank's radars data every one second as well and, if an obstacle is detected in the range of the radars, the event *wallSighted* is narrow-casted, and the system is transferred from state *Checking* to state *Waiting*.

```
print "narrow-casting event: wallSighted"
ctl.obstacleMap.event("wallSighted")
```

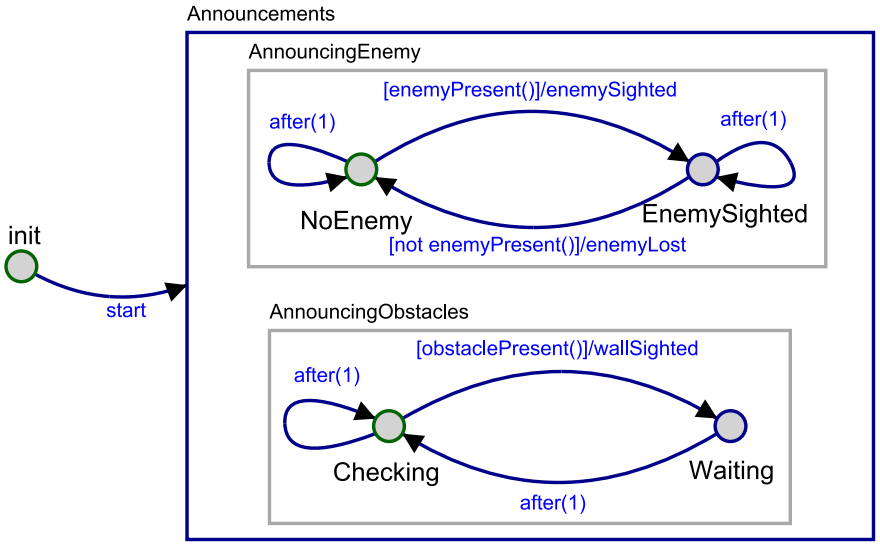


Figure 3.16: *Announcements* statechart.

One second after entering the *Waiting* state the system is transferred back to *Checking*, and there is no output action attached to this transition. Therefore, the *Waiting* state could be deleted and the *obstaclePresent()* transition could be directed to the *Checking* state itself -hence making it a self-transition-, since the only purpose of it is to generate and narrow-cast an output event. It's the task of the components at the higher levels of abstraction of the tank's AI to take decisions and generate output actions in response to the radar input provided by the components at the lower levels of abstraction.

3.5.3 *InRangeDetector*

Once an enemy is spotted -which means it has been sighted within the range of the tank's radars- it still can not be attacked unless the tank's turret is aiming at it; if it is not, then the turret must be turned until it is facing the enemy. In order for the tank pilot to shoot successfully, the enemy must also be in range of the turret, otherwise the tank should move closer to the target. As captured in Figure 3.17, state *Seeking* has an unlabeled outgoing transition to state *InRange*, that is taken when the guard condition evaluates to true; state *Seeking* is re-entered every one second thus forcing this guard's check. The *TurretRadar* class provides the methods *enemyInFront()*, which returns true if the tank's turret is facing the enemy, and *enemyDistance()*, which returns the current distance separating the tank from the enemy. The system is transferred from state *Seeking* to state *InRange* when the enemy is aimed by and in range of the tank's turret. The event *readyToShoot* is generated as an output action.

```
print "narrow-casting event: readyToShoot"
ctl.attackStrategy.event("readyToShoot")
```

The system remains in state *InRange* while the condition that triggered the outgoing transition from state *Seeking* remains true; otherwise, the system is transferred back to state *Seeking*.

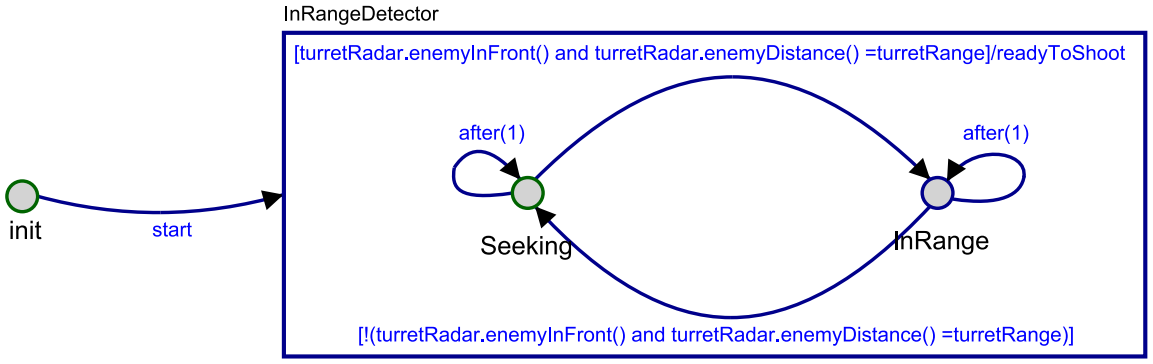


Figure 3.17: *InRangeDetector* statechart.

3.5.4 *EnemyTracker*

The tank pilot must take care of many things going on around him at the same time, but his main objective is only one and it remains simple: to find the enemy and destroy it. The *EnemyTracker* module is in charge of tracking the enemy by memorizing its last known position. Taking a look at the statecharts model captured in Figure 3.18 we can see that, right after the execution starts, the initial state will be *NoEnemy*, but the system will never transfer back to this state once the enemy has been sighted for the first time. The moment the enemy falls within the scope of any of the tank's radars, the event *enemySighted* is narrow-casted to the *EnemyTracker* module from the *Announcements* module. The first time this event is fired, it causes the system to transfer from state *NoEnemy* to state *TrackedBy1Radar*. If the event is received again while in *TrackedBy1Radar*, the system is transferred to state *TrackedBy2Radar*. Note that it doesn't matter if both radars spot the enemy at the same time; since the firing of the event *enemySighted* is sequential it would still be narrow-casted twice, within a very little time lapse.

Once is the enemy's position is known, it must be "memorized"; we store it in a 2D-coordinate variable of the tank controller class. This variable must be kept up-to-date by the *EnemyTracker* module and, therefore, all of its statechart's transitions that are triggered by the event *enemySighted* generate exactly the same output action, which is the call to a method:

```
print "new data received: enemy has been sighted"
print "updating enemy's coordinates on the map"
ctl.updateEnemyPos()
```

The method *updateEnemyPos()* is defined in the tank controller class. It calculates the enemy's current position from the tank's radars data.

When the enemy escapes a radar's sight, the event *enemyLost* is narrow-casted from the *Announcements* module, causing the *EnemyTracker*'s current state to be transferred either from *TrackedBy2Radar* to *TrackedBy1Radar* or from *TrackedBy1Radar* to *EnemyPosUnsure*, if both of the tank's radars no longer have the enemy in range. The enemy's last known coordinates on the map remains unaltered until the enemy is sighted again by any of the radars.

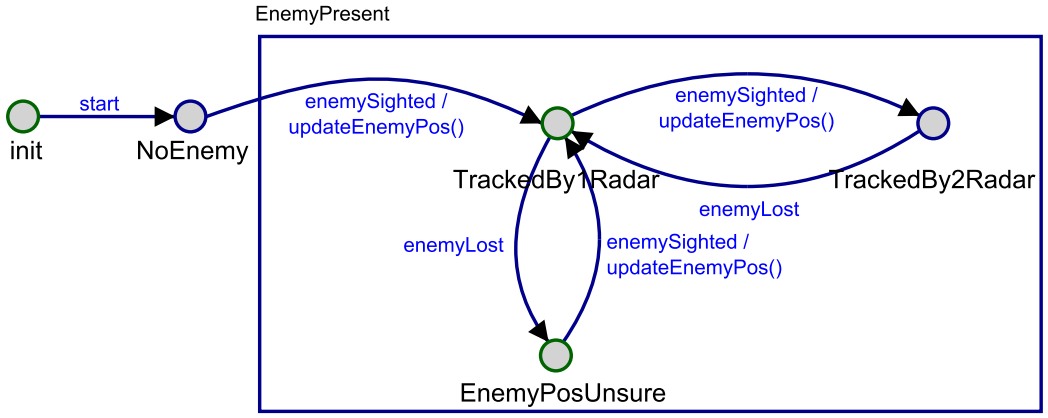


Figure 3.18: *EnemyTracker* statechart.

3.5.5 *UpdateMap*

The world surrounding himself and the tank is unknown to the pilot every time the execution begins. The tank slowly glides through the world, unveiling and dodging its obstacles thanks to the information provided by the tank's sensors. This data is stored in an internal data structure -most conveniently a matrix-, hence building a map of the world which keeps growing in detail as the execution develops. The *UpdateMap* module's statechart, as captured in Figure 3.19, is very simple: it consists of only one state with a self-transition. When the event *wallSighted* is received from the *Announcements* module, the transition is taken causing the tank controller to update its world map with the new data provided by the tank's radars.

```
print "new data received: updating world map"
ctl.updateMap()
```

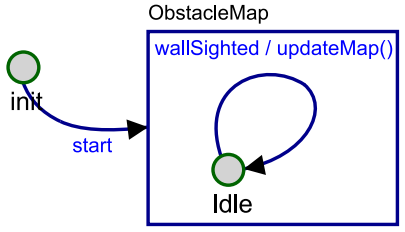


Figure 3.19: *UpdateMap* statechart.

3.5.6 *PilotStrategy*

This module, just like its name implies, reflects the decisions taken by the tank pilot, who is responsible for the tank's safety. The pilot's character will definitely influence his actions, according to which an offensive strategy could prevail over a defensive one, or otherwise. The size of the statecharts model to this module is directly proportional to the importance of the task it develops. Not in vain, the reader may recall, this is the component at the highest level of abstraction of the tank's AI, where decision making takes place. Many -if not all- of the transitions in this

statechart are triggered by events generated in the components at the lower levels of abstraction, which we have previously covered. The *PilotStrategy* statechart, depicted in Figure 3.20, consists of several composite states the default of which is *EnoughFuel.NormalOperation.Exploring*. The tank remains *exploring* while the enemy's position is unknown and the fuel level is not low; if the *EnemyTracker* module reports that the enemy has been spotted -so that *enemyTracker.enemyPosKnown* is true-, and the fuel level is still ok, the system is transferred to state *EnoughFuel.NormalOperation.Attacking*, generating the event *attack* as an output action.

```
print "narrow-casting event: attack"
ctl.attackStrategy.event("attack")
```

While in state *EnoughFuel.NormalOperation*, if the event *damageHigh* is received and the location of the repair station is known, the system transfers to state *EnoughFuel.Repairing*, otherwise it transfers to state *EnoughFuel.Fleeing*. The system cannot leave state *EnoughFuel.Repairing* until the event *repaired* is received, then it transfers back to *EnoughFuel.NormalOperation*. Also, if the location of the repair station becomes known while in state *EnoughFuel.Fleeing*, the system transfers to *EnoughFuel.Repairing*.

As well, while in states *EnoughFuel.NormalOperation* or *EnoughFuel.Fleeing*, if the event *fuelLow* is received and the location of the fuel station is known, the system transfers to state *Refueling*. Only when the event *fuelFull* is received, the system will transfer to state *OperationHistory* -which is a deep history state- and, therefore, back to the state it was in prior to transferring to state *Refueling*. Nevertheless, whenever the event *fuelLow* is received, the location of the fuel station is unknown, and the current state is *EnoughFuel*, the system transfers to state *EnoughFuel.NormalOperation.Exploring*.

3.5.7 *AttackStrategy*

Notwithstanding the fact that there can't be much strategy to a tank's attack resources, due to its very limited arsenal, reduced mobility, and little maneuverability, it's the *AttackStrategy* module's task to destroy the enemy. This module's default state, as captured in Figure 3.21, is *Idle*, and it remains so while the tank is in exploring mode. Once the enemy has been spotted and the pilot has decided, upon considering many factors, to enter the attacking mode, the event *attack* is received and the system is transferred to composite state *MovementAimingAndShooting*, which consists of two orthogonal components: *MovementShooting* and *MovementAiming*. The first of these two orthogonal components in the one in charge of shooting at the enemy. The event *readyToShoot*, which is narrow-casted by the *InRangeDetector* module, triggers the transition leading from state *Ready* to state *Shooting*, and a missile is shot as an output action to this transition -through a call to the respective tank controller class's method-. After a predefined "time to reload", the turret is ready to shoot again so the system transfers back to state *Ready*.

```
print "readyToShoot: entering MovementShooting.Shooting"
ctl.shoot())
```

As for the *MovementAiming* component, its mission is to try to keep the enemy in sight and in range of the tank turret at all times, as long as the tank is in attacking mode. The system remains static in state *FollowEnemy* until the event *enemyPosChanged* is narrow-casted by the *EnemyTracker* module. Then, state *FollowEnemy*'s self-transition is taken, and the following output action is executed:

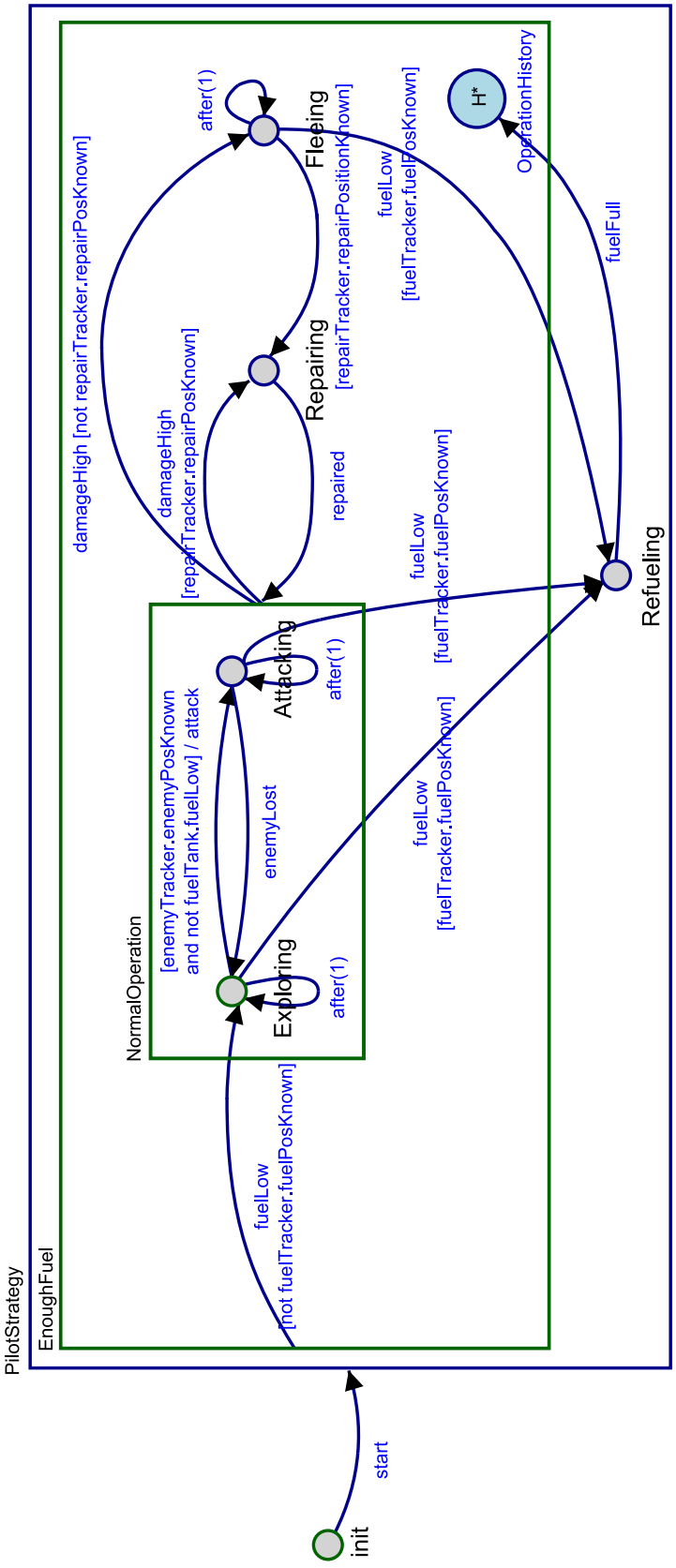


Figure 3.20: *PilotStrategy* statechart.

```
print "moving to enemy's last known position"
ctl.newDestination(ctl.enemyTracker.getEnemyPos())
print "aiming turret at enemy's last known position"
ctl.aimAt(ctl.enemyTracker.getEnemyPos())
```

The call to the first method, *newDestination()*, will move the tank to the enemy's last known position, so that it might remain **in range** of the tank turret. The call to the second method, *aimAt()*, will turn the turret until it is aimed at the enemy's last known position, so that it might remain **in sight** of the tank turret.

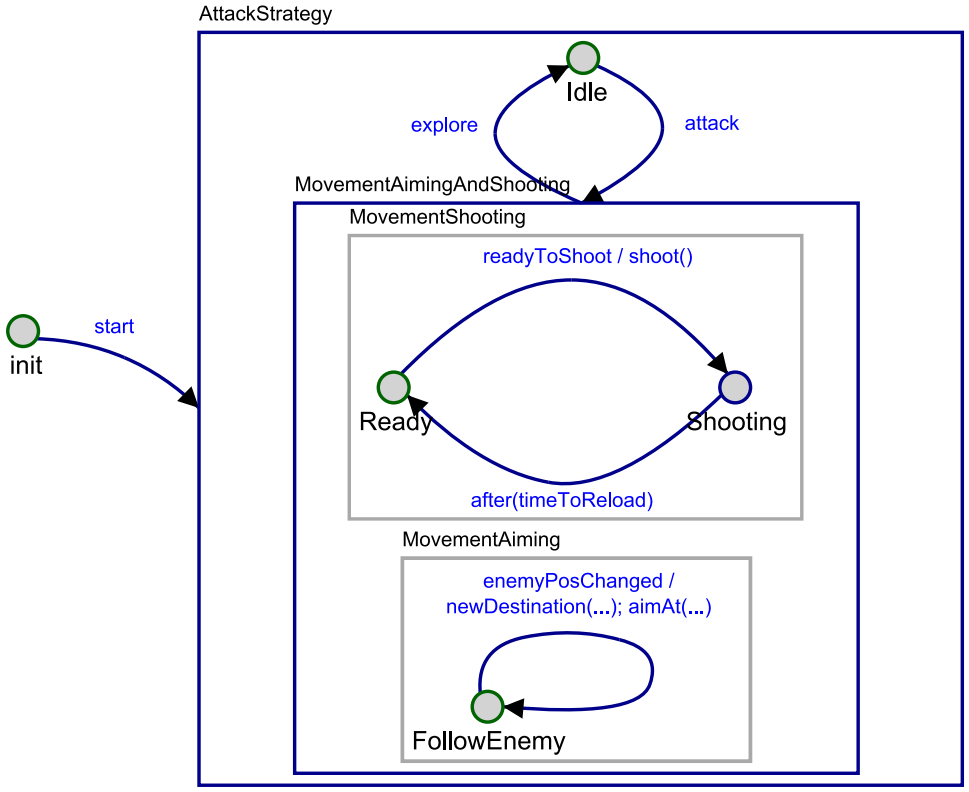


Figure 3.21: *AttackStrategy* statechart.

3.5.8 *MotorControl*

As depicted in Figure 3.22, the statecharts model of the *MotorControl* module consists of two orthogonal components, *Turning* and *Moving*. According to this, the tank can move forward or backward while turning left or right. Each of the states *MovingForward, MovingBack, TurningLeft*, and *TurningRight* have an enter action defined, which calls the corresponding tank controller class's method that performs one "movement/turn step" in the designated direction. Once the method in question is executed and the step is performed, the current orthogonal states remain the same but the tank does not keep moving. This is solved by adding the timed self-transitions, which cause the current states to be re-entered -and thus their respective enter actions to be executed- until an event is received and it transfers the system to a different pair of states. For example, if the system is currently in states *MovingForward* + *TurningRight*, the

tank will keep moving forward and turning right indefinitely. Then the event *stopTurn* is received and the system transfers to states *MovingForward* + *GoingStraight*. The distance covered in one movement step, as well as the angle covered in one turn step, and the fuel consumed by steps of both kinds, are constant values defined as attributes of the tank controller class.

The tank controller class's methods called by the motor control are:

```
print "entering MotorControl.MovingForward"
ctl.moveForward()
...
print "entering MotorControl.MovingBack"
ctl.moveBackward()
...
print "entering MotorControl.TurningLeft"
ctl.tankTurnLeft()
...
print "entering MotorControl.TurningRight"
ctl.tankTurnRight()
```

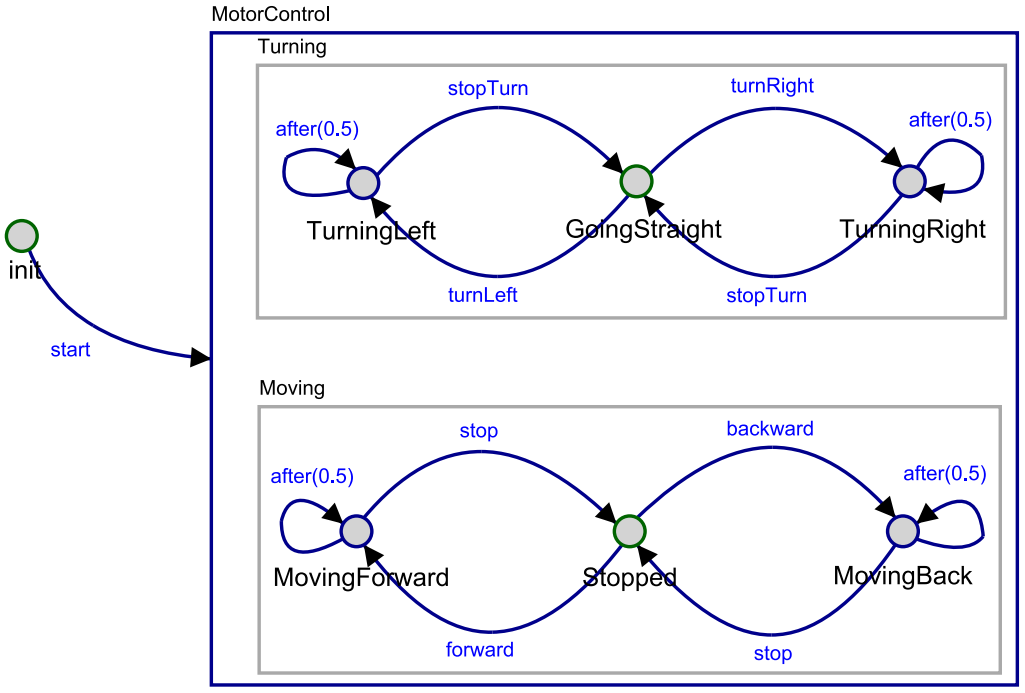


Figure 3.22: *MotorControl* statechart.

3.5.9 *TurretControl*

The turret is mounted on top of the tank and it has a 360° turning capability, having its turning axis in the geometric center of the tank. Although the *TurretControl* module statechart is very similar to that of the *MotorControl* module, it would not be possible to have them share the same statechart, since the turret moves independently from the rest of the tank. A quick look at the statechart depicted in Figure 3.23 is enough to understand how the turret works.

```
print "entering TurretControl.TurningLeft"
ctl.turretTurnLeft()
...
print "entering TurretControl.TurningRight"
ctl.turretTurnRight()
```

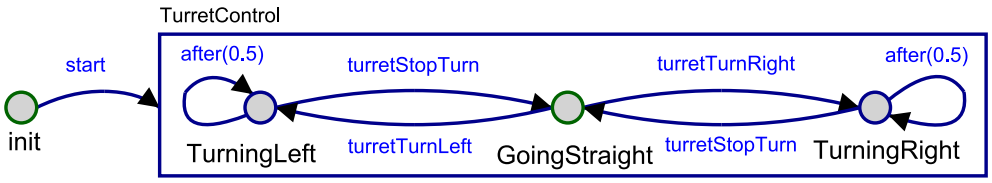


Figure 3.23: *MotorControl* statechart.

3.6 The controller: linking the AI components

In the previous section's subsections we've repeatedly mentioned the *tank controller class*, but what is it? Once we have individually modelled all of our AI components with statecharts and compiled them into Python we must put them together, combine them in order to create one single reactive system: the AI itself. Therefore, we needed to come up with a way of linking the components so that they could communicate and interact with one another; the solution is the *controller*.

The controller is a Python class in which we declare an instance of each of the AI components's classes as attributes. The following code snippet is the constructor method of the tank controller class.

```
class TankController:
      def __init__(self):
          # define an instance of each of our statecharts
          # fuel tank (sensors)
          self.fuelTank = FuelTank.FuelTank()
          self.fuelTank.fuelLevel = MAX_TANK_FUEL
          self.fuelTank.fuelLow = False
          self.fuelTank.initModel()
          self.fuelTank.event('start', self)

          # announcements (sensors)
          self.announcements = Announcements.Announcements()
          self.announcements.initModel()
          self.announcements.event('start', self)

          # in range detector (analyzers)
          self.inRangeDetector = InRangeDetector.InRangeDetector()
          self.inRangeDetector.initModel()
          self.inRangeDetector.event('start', self)
```

```
# enemy tracker (memorizers)
self.enemyTracker = TrackEnemy.TrackEnemy()
self.enemyTracker.enemyPosKnown = False
self.enemyTracker.initModel()
self.enemyTracker.event('start', self)

# obstacle map (memorizers)
self.obstacleMap = ObstacleMap.ObstacleMap()
self.obstacleMap.initModel()
self.obstacleMap.event('start', self)

# pilot strategy (strategic deciders)
self.pilotStrategy = PilotStrategy.PilotStrategy()
self.pilotStrategy.initModel()
self.pilotStrategy.event('start', self)

# turret tank movement coordinator (coordinators)
self.turretMovCoord = TurretTankMovCoord.TurretTankMovCoord()
self.turretMovCoord.initModel()
self.turretMovCoord.event('start', self)

# motor control (actuators)
self.motorControl = MotorControl.MotorControl()
self.motorControl.initModel()
self.motorControl.event('start', self)

# turret control (actuators)
self.turretControl = TurretControl.TurretControl()
self.turretControl.initModel()
self.turretControl.event('start', self)
```

Note that, for each of these tank controller class's attributes -which are compiled statecharts-, the method `initModel()` is called right after the instantiation; this method initializes the execution of the statecharts model, putting the system in its default state. After that, the event *start* is narrow-casted to each statechart/attribute, definitely starting the AI's main loop. The controller itself is passed as a parameter to each statechart along with the event *start*. By having a reference to the controller, each component's statechart can narrow-cast events to the other components's statecharts, like we've already seen in the code snippets of the previous section's subsections. The AI components can only interact with one another by means of the controller, so it is an essential part of our application.

3.7 Creating a simple simulation environment

Once the statecharts models were finished and compiled, we needed to put them together in order to test them. Hence we developed a very simple simulation environment using Python's Tkinter. The simulation environment consists of a static part, which is the world map, and a dynamic part, which is/are the tank/s. Both parts are covered in detail in the following subsections.

3.7.1 The world map

The map is defined in a plain text file using numbers, each number representing a tile in the map. These numbers are:

- 0 - EMPTY. The default color of the map is gray.
- 1 - WALL. The color of walls on the map is black.
- 2 - TANK. The color of the tank and its turret is green.
- 3 - FUEL STATION. There is only one per map; its color is blue.
- 4 - REPAIR STATION. There is only one per map; its color is red.

The function `loadFile()` in the class `FileReader` receives the name of the map file to be opened and reads it, loading the lines/strings into a Python list. This list is later treated by the drawing class, `Drawer`.

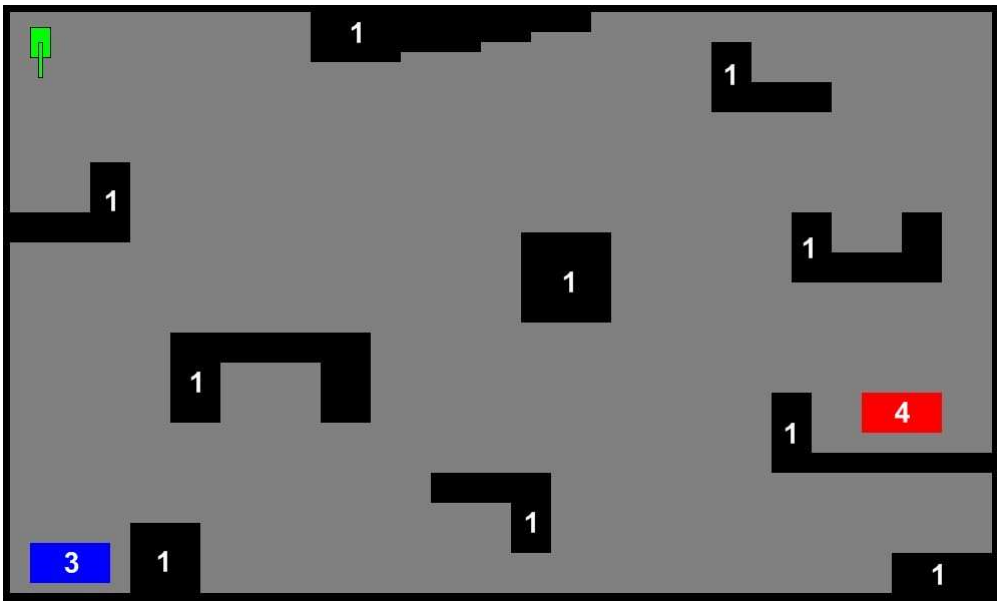


Figure 3.24: Simple recreation of a world map using Tkinter canvas rectangles.

A Tkinter canvas widget is created as the base upon which the map tiles will be drawn. The canvas height is given by the number of lines/strings read from the map file, and the width is given by the length of these strings (all strings must have equal length). The line list is accessed like a matrix, using two indexing variables X and Y as coordinates to parse the strings read from the map file. For every number in each string, a canvas rectangle item is created on the canvas, and the reference to it is kept in a reference matrix that is indexed by the same X and Y coordinate variables. The map and its elements are static, so it is only drawn once at the beginning of the simulation.

3.7.2 The tank

The tank is represented on the screen by two rectangular green boxes, a big one for the body and a very thin, larger one on top of that for the turret. The turret's inner edge is anchored in the middle of the body box, so that the turret and the body share the same rotation center.

The two boxes representing the tank are Tkinter canvas polygon items. Polygons are created by giving an indefinite number of consecutive vertex (specified by X and Y coordinates), the last of which is automatically connected to the first vertex. The canvas `create_polygon()` function returns a reference to the created polygon. The references to both tank's polygons are stored in two variables, so that both items can be deleted from the canvas when the tank has to be redrawn. The tank is deleted from and redrawn in the canvas every time it moves or turns.

3.7.2.1 Rotating the tank

It was chosen to use canvas polygon items instead of canvas rectangle items, even though both the tank's body and the tank's turret are rectangular, because it was the only way the tank could be rotated and redrawn. Whenever the tank turns the body and turret polygons must be rotated; the amount of degrees to turn is predefined by a constant in our drawing class.

In order to rotate the polygons, their vertex must be rotated first and then the polygons must be redrawn. Each of the polygons's vertex are rotated with respect to the same rotation center, the tank's center. The function `rotate()` receives a Python list of points, the origin point with respect to which the points in the list are going to be rotated, and the rotation angle. For every point in the list the function `rotatePoint()` is called, returning the new, rotated point. The tank's original polygons are deleted from the canvas and the new ones are drawn, according to their rotated lists of vertex.

3.7.2.2 Translating the tank

The first time the polygons representing the tank and its turret are drawn on the canvas, the same origin point is used as a reference to define the outer vertex. When translating the tank, all its polygons's vertex are translated, as well as the origin point. The current angle of the tank is necessary to calculate the translated vertex since, depending on the angle, not all the vertex are going to be translated, e.g. if the tank is facing right (0^o angle) or left (180^o angle) the tank must be translated only on the X axis, and if the tank is facing up (90^o angle) or down (270^o angle) the tank must be translated only on the Y axis.

3.8 Statecharts modelled simulation

At this point, we have discussed the tank's AI components, how we modelled them in statecharts and then compiled them into Python classes. We have also discussed the tank controller class and the GUI of our simple Python simulation application. There is only one thing left to discuss: how is the simulation run? According to the MSDL's motto -"Model everything"-, it seemed appropriate to *model the simulation* too. Therefore, we defined the simulation with a statecharts model that we later compiled into Python. The most interesting aspect about modelling the simulation instead of coding it is that we can easily define several different simulation statecharts, as many as we want, and simply combine them with the AI's compiled classes, the tank controller class, and the GUI; we don't have to worry about writing extra bridging code in order to put the components together, they fit right away. Thus, we tried a few different simulation statecharts in our application, one of which is captured in Figure 3.25.

In our main application file we define an instance of the tank controller class, and an instance of the simulation class. The latter receives the tank controller as a parameter. The simulation statechart -as depicted in Figure 3.25-, simply specifies a series of movement commands in order to keep the tank moving randomly for as long as the execution runs. The AI events must be introduced manually by the user, and then they are narrow-casted to the components's

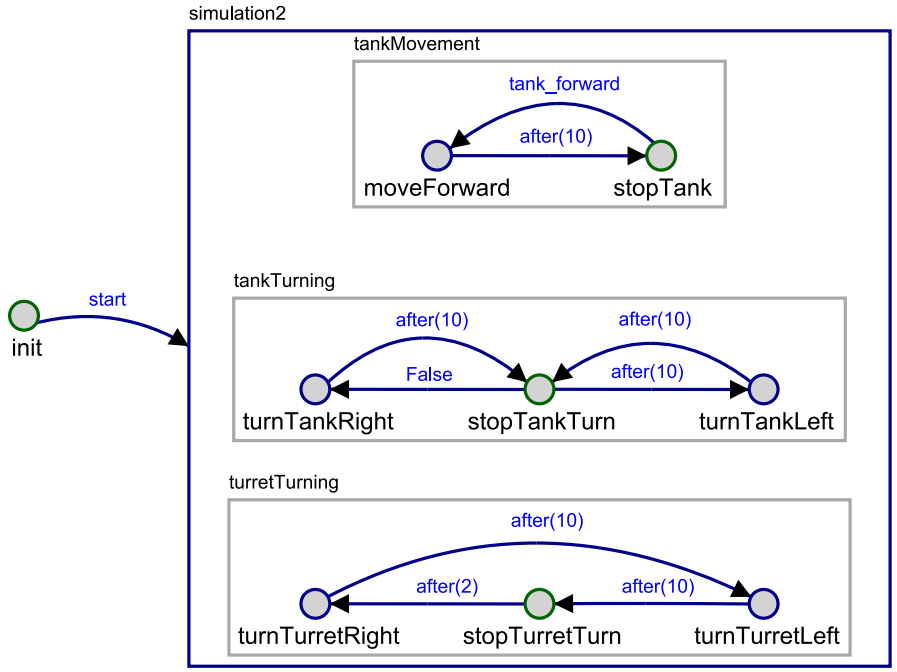


Figure 3.25: The simulation statechart defines random movements.

statecharts by means of a controller class's method called *narrow_ cast()*.

```
class TankController:
      def narrow_cast(event, self):
          self.fuelTank.event(event, self)
          self.announcements.event(event, self)
          self.inRangeDetector.event(event, self)
          self.enemyTracker.event(event, self)
          self.obstacleMap.event(event, self)
          self.pilotStrategy.event(event, self)
```

The method *narrow_ cast()* simply transmits the user's manually introduced events to the AI's statecharts. Some events might cause no effect in some statecharts, but they might trigger certain transitions in some others. The following code snippet captures the main application file; the last few lines define the input loop through which the user can keep introducing events until the event name 'exit' is introduced, which terminates the simulation execution.

```
# create main window
root = Tk()
root.withdraw()
window = newWindow(root)

# load input file that contains the definition of the world map
fr = FileReader.FileReader('input2.txt')
fr.loadFile()
```

```
# define an instance of our drawer class
dr = Drawer.Drawer(root, window, fr.lines, 40, 40)
# define an instance of our controller class
ctl = TankController.TankController(dr)

# SIMULATION
# instantiate the simulation class that we want to run
# the simulation class is a compiled statechart as well
sim = simulation_Compiled.simulation()
sim.initModel()
sim.event('start', ctl)
root.mainloop()

while True:
      event = raw_input('++ enter event: ')
      if event == 'exit'
            break
      print 'narrow casting ' + event
      ctl.narrow_cast(event)
```

4

Statecharts modelling of a robot's behavior

Introduction

Ten years ago, a robot that autonomously cleaned the house for you would only have been possible in science-fiction books and movies. iRobot Corporation turned fiction into reality in 2002, when the first model of Roomba robot was released, making the wishes of many housewives come true: the Roomba is a round, flat, and quite small robot and an autonomous home vacuum cleaner. Equipped with wheels and sensors, the Roomba [27] moves around a room searching for dirt and cleans it, can sneak under sofas to vacuum the floor dust, and finds its way back to the charging base when the capacity of its battery is low. What's best, being completely autonomous, the Roomba doesn't require any kind of supervision during the fulfillment of its cleaning tasks (only, maybe, in the case of having pets wandering around the house). Since the first model of Roomba robot was released, iRobot Corporation hasn't stopped adding new members to the Roomba family; nowadays it's possible to choose amongst vacuum cleaning, floor washing, shop sweeping, pool cleaning, gutter cleaning and virtual visiting Roombas.

Figure 4.1: First generation Roomba (left) Vs. the latest Roomba 500 Series (right).

The iRobot Create [6] is very much like a Roomba robot with the main difference being that the first doesn't vacuum. Create is a programmable robot designed mainly for research purposes, so it is especially suited for educators and roboticists. This is the full list of features as described in iRobot Create's website:

- Mobile out-of-the-box. No need to assemble the drive system or worry about low-level code.

- Use the included serial cable to send individual commands from a PC or write basic scripts of up to 100 Open Interface Commands which can be stored on the robot.
- 10 built-in demos perform various, preprogrammed behaviors and can be controlled via Open Interface Commands.
- Over 30 built-in sensors allow the robot to react to both internal and external events.
- 25-pin expansion port to add the iRobot Command Module (not included) or your own electronics including sensors, actuators, wireless connections, computers, cameras, or other hardware.
- Spacious cargo bay with threaded holes for mounting additional hardware.
- Includes fourth wheel to improve stability of larger payloads
- Full compatibility with iRobot Roomba accessories including the rechargeable batteries, power supply, home base, remote control, and virtual walls.

Figure 4.2: iRobot Create (left) and the Command Module (right).

Access to all of the robot's sensors and actuators over a serial port is provided via the iRobot Create Open Interface. As well, the robot's behavior can be tested by sending simple commands through a serial port by using any basic terminal program. Only for highly advanced users there's also the possibility of programming the Create robot by using its Command Module [28], which we vulgarly refer to as "the green bar". The Command Module attaches to the Create robot for completely autonomous behavior, it *"contains an 8-bit, 20 MHz (Atmel ATMega 168) microcontroller enabling full programmability of iRobot Create's motors, lights, sounds, and sensor readings. The Command Module plugs into the Create robot's cargo bay connector and provides on-board control with programs you write in C. It has four DB-9 expansion ports for adding your own hardware."*.

4.1 iRobot Create Open Interface

iRobot Create has three operating modes:

- **Passive**. When in passive mode, it's possible to request and receive data from Create's sensors, but the current command parameters for the actuators (motors, speaker, lights,

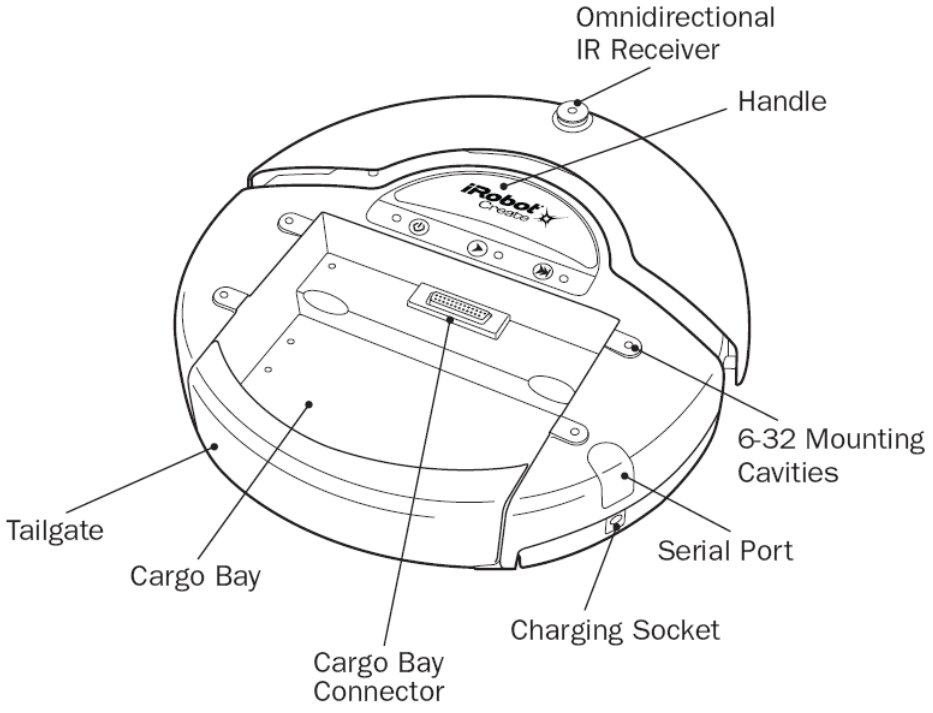


Figure 4.3: Detailed top view of the iRobot Create and its components.

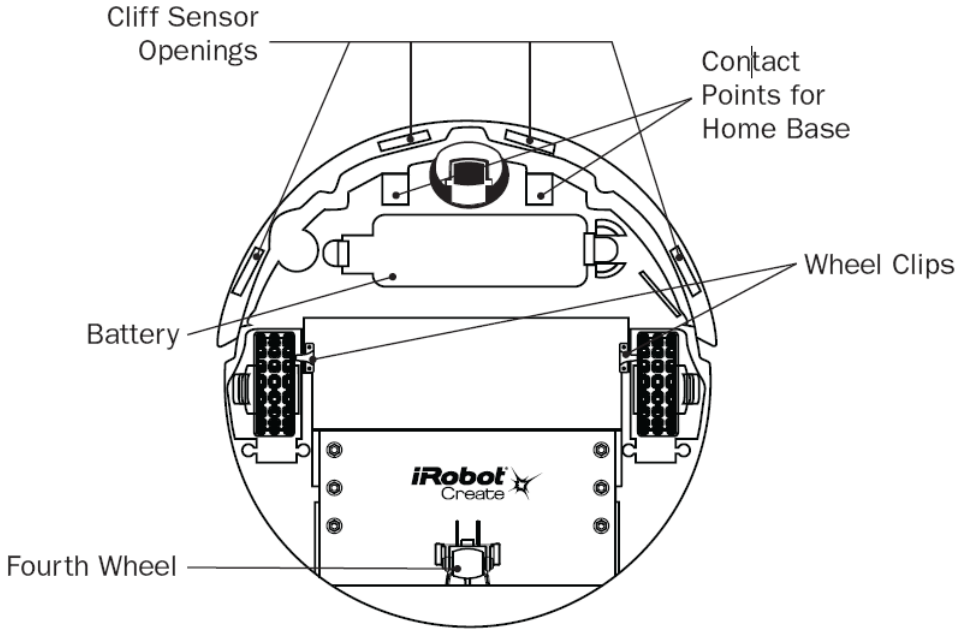


Figure 4.4: Detailed bottom view of the iRobot Create and its sensors.

low side drivers, digital outputs) can not be changed. While in passive mode, Create's built-in demos can be executed and the battery can be charged as well.

- **Safe**. Safe mode gives the user full control of Create, with the exception of some safety-related conditions, which are:
  - Detection of a cliff while moving forward, which would allow Create to avoid falling down stairs.
  - Detection of a wheel drop, i.e. unevenness in the floor level.
  - Charger plugged in and powered.
- **Full**. Full mode gives the user complete control over Create, all of its actuators, and all of the safety-related conditions that are restricted when the OI is in Safe mode. If no commands are sent to the OI when in Full mode, Create waits with all motors and LEDs off and does not respond to Play or Advance button presses or other sensor input. Besides, Create's battery can not be charged when in Full Mode.

There are 30 operations codes to iRobot Create OI, some of which are the following:

- **Opcode 128:** ***Start***.
  This command starts the OI. The Start command must always be sent before sending any other commands to the OI.
  - Serial sequence: `[128]`
  - Changes mode to: Passive. Create beeps once to acknowledge it is starting from "off" mode
- **Opcode 136:** ***Demo***.
  This command starts the requested built-in demo. There are 10 built-in demos; sending demo number -1 stops the demo that Create is currently performing.
  - Serial sequence: `[136][Which-demo]`
  - Demo data byte 1: Demo number (-1 - 9)
- **Opcode 137:** ***Drive***.
  This command controls Create's drive wheels. It takes four data bytes, interpreted as two 16-bit signed values using two's complement. The first two bytes specify the average velocity of the drive wheels in millimeters per second (mm/s), with the high byte being sent first. The next two bytes specify the radius in millimeters at which Create will turn. The longer radii make Create drive straighter, while the shorter radii make Create turn more. The radius is measured from the center of the turning circle to the center of Create. A Drive command with a positive velocity and a positive radius makes Create drive forward while turning toward the left. A negative radius makes Create turn toward the right. Special cases for the radius make Create turn in place or drive straight, as specified below. A negative velocity makes Create drive backward.
  - Serial sequence: `[137] [Velocity high byte] [Velocity low byte] [Radius high byte] [Radius low byte]`
  - Drive data byte 1: Velocity (-500 - 500 mm/s)
  - Drive data byte 2: Radius (-2000 - 2000 mm)

Special cases:

- Straight = 32768 or 32767 = `hex 8000` or `hex 7FFF`
- Turn in place clockwise = `hex FFFF`
- Turn in place counter-clockwise = `hex 0001`

- **Opcode 140: *Song*.**
  This command lets you specify up to sixteen songs to the OI that can be played at a later time. Each song is associated with a song number. The Play command uses the song number to identify the song selection. Each song can contain up to sixteen notes; each note is associated with a note number that uses MIDI note definitions and a duration that is specified in fractions of a second.
  - Serial sequence: `[140] [Song Number] [Song Length] [Note Number 1] [Note Duration 1] [Note Number 2] [Note Duration 2], etc.`
  - Song data byte 1: Song Number (0 - 15). The song number associated with the specific song. If a song with the same song number had been previously loaded on Create, the old song is overwritten
  - Song data byte 2: Song Length (1 - 16). The length of the song, according to the number of musical notes within the song
  - Song data bytes 3, 5, 7, etc.: Note Number (31 - 127). The pitch of the musical note Create will play, according to the MIDI note numbering scheme.
  - Song data bytes 4, 6, 8, etc.: Note Duration (0 - 255). The duration of a musical note, in increments of 1/64th of a second

- **Opcode 141: *Play song*.**
  This command lets you select a song to play from the songs added to iRobot Create using the Song command.
  - Serial sequence: `[141] [Song Number]`
  - Play Song data byte 1: Song Number (0 - 15). The number of the song Create is to play.

- **Opcode 142: *Sensors*.**
  This command requests the OI to send a packet of sensor data bytes. There are 43 different sensor data packets; each provides a value of a specific sensor or group of sensors (read further below).
  - Serial sequence: `[142] [Packet ID]`
  - Sensor data byte 1: Packet ID (0 - 42)

- **Opcode 152: *Script*.**
  This command specifies a script to be played later. A script consists of OI commands and can be up to 100 bytes long. "Wait" commands cause Create to hold its current state until the specified event is detected.
  - Serial sequence: `[152] [Script Length] [Opcode 1] [Opcode 2] [Opcode 3] etc.`
  - Script data byte 1: Script Length (0 - 100). Specifies the length of the script in terms of the number of commands. Specify a length of 0 to clear the current script

  - Script data bytes 2 and above: Open Interface commands and data bytes

- **Opcode 153:** ***Play script***.
  This command loads a previously defined OI script into the serial input queue for playback.
  - Serial sequence: `[153]`

- **Opcode 156:** ***Wait distance***.
  This command causes iRobot Create to wait until it has traveled the specified distance in mm. Until Create travels the specified distance, its state does not change, nor does it react to any inputs, serial or otherwise.
  - Serial sequence: `[156] [Distance high byte] [Distance low byte]`
  - Wait Distance data bytes 1-2: 16-bit signed distance in mm, high byte first (-32767 -32768)

- **Opcode 157:** ***Wait angle***.
  This command causes Create to wait until it has rotated through specified angle in degrees.Until Create turns through the specified angle, its state does not change, nor does it react to any inputs, serial or otherwise.
  - Serial sequence: `[157] [Angle high byte] [Angle low byte]`
  - Wait Angle data bytes 1-2: 16-bit signed angle in degrees, high byte first (-32767 -32768)

Create can request information from all of its sensors through OI commands too. As we've already seen, there are 43 different sensor data packets; most of the packets (numbers 7 - 42) contain the value of a single sensor or variable, which can be either 1 byte or 2 bytes. Two-byte packets correspond to 16-bit values, sent high byte first. Some of the packets (0-6) contain groups of the single value packets. Having a look at the sensor command serial sequence again:
Serial sequence: `[142] [Packet ID]`
`[Packet ID]` identifies which of the 43 sensor data packets should be sent back by the OI. The following table describes which packets does each packet group contain, and their sizes in bytes.

Group packet ID	Packet size	Contains packets
0	26 bytes	7 - 26
1	10 bytes	7 - 16
2	6 bytes	17 - 20
3	10 bytes	21 - 26
4	14 bytes	27 - 34
5	12 bytes	35 - 42
6	52 bytes	7 - 42

These are some of the most interesting sensor packets:

- **Packet Id: 7.** ***Bumps and wheel drops***.
  The state of the bumper (0 = no bump, 1 = bump) and wheel drop sensors (0 = wheel raised, 1 = wheel dropped) are sent as individual bits.
  - Data bytes: 1 unsigned
  - Range: 0 - 31

Bit	7	6	5	4	3	2	1	0
Sensor	n/a	n/a	n/a	Wheeldrop caster	Wheeldrop left	Wheeldrop right	Bump left	Bump right

- **Packet id: 8. *Wall*.**
  The state of the wall sensor is sent as a 1 bit value (0 = no wall, 1 = wall seen).
  - Data bytes: 1 unsigned
  - Range: 0 - 1
- **Packet id: 9. *Cliff left*.**
  The state of the cliff sensor on the left side of Create is sent as a 1 bit value (0 = no cliff, 1 = cliff). Same for packet ids 10: *Cliff front left*, 11: *Cliff font right*, and 12: *Cliff right*.
  - Data bytes: 1 unsigned
  - Range: 0 - 1
- **Packet id: 19. *Distance*.**
  The distance that Create has traveled in millimeters since the distance it was last requested is sent as a signed 16-bit value, high byte first. This is the same as the sum of the distance traveled by both wheels divided by two. Positive values indicate travel in the forward direction; negative values indicate travel in the reverse direction.
  - Data bytes: 2 signed (1 short)
  - Range: -32768 - 32767
- **Packet id: 20. *Angle*.**
  The angle in degrees that iRobot Create has turned since the angle was last requested is sent as a signed 16-bit value, high byte first. Counter-clockwise angles are positive and clockwise angles are negative.
  - Data bytes: 2 signed (1 short)
  - Range: -32768 - 32767
- **Packet id: 25. *Battery charge*.**
  The current charge of Create's battery in milliamp-hours (mAh). The charge value decreases as the battery is depleted during running and increases when the battery is charged.
  - Data bytes: 2 unsigned
  - Range: 0 - 65535
- **Packet id: 37. *Song playing*.**
  The state of the OI song player is returned. 1 = OI song currently playing; 0 = OI song not playing
  - Data bytes: 1 unsigned
  - Range: 0 - 1

In the following section we'll cover several OI command examples.

4.2 iRobot Create Open Interface command examples

4.2.1 Requesting a sensor packet

Command	Opcode	Data bytes 1	Data bytes 2	Data bytes 3	Data bytes 4	Etc.
Sensors	142	Packet ID: (0-42)				

Example:

```
[142] [20]
```

Requests sensor packet 20, Angle (the angle in degrees that iRobot Create has turned since the angle was last requested)

Example:

```
[142] [1]
```

Requests group packet 1, which contains sensor packets 7-16 (Bumps and wheel drops, Wall, Cliff left, Cliff front left, Cliff front right, Cliff right, Virtual wall, and Low side driver and wheel overcurrents.

4.2.2 Driving forward

Command	Opcode	Data bytes 1	Data bytes 2	Data bytes 3	Data bytes 4	Etc.
Drive	137	Velocity (-500 - 500 mm/s)		Radius (-2000 - 2000 mm)		

Example:

```
[137] [0] [100] [1] [144]
```

Velocity and radius are represented by 2 signed bytes each, and sent to the robot high-byte first. Therefore:

Velocity = `[0] [100]` = `[hex 00] [hex 64]` = `hex [0064]` = 100 mm/s
Radius = `[1] [144]` = `[hex 01] [hex 90]` = `hex [0190]` = 400 mm

4.2.3 Loading and playing a song.

Command	Opcode	Data bytes 1	Data bytes 2	Data bytes 3	Data bytes 4	Etc.
Load Song	140	Song number (0-15)	Song length (1-16)	Note #1 (31-127)	Note duration 1 (0-255)	Note #2, etc.

Example:

```
[140] [0] [4] [62] [12] [66] [12] [69] [12] [74] [36]
```

Song number is 0, the number of notes is 4, the first note is 62 and its duration is 12.

Command	Opcode	Data bytes 1	Data bytes 2	Data bytes 3	Data bytes 4	Etc.
Play Song	141	Song number (0-15)				

Example:

```
[141] [0]
```

Play song number 0.

4.2.4 Driving in a square.

For this example we'll use a script.

Command	Opcode	Data bytes 1	Data byes 2	Data bytes 3	Data bytes 4	Etc.
Script	152	Script length (1-100)	Command opcode 1	Command data byte 1, etc.	Command opcode 2	Etc.
Play script	153					

We want our Create robot to drive in a square, so we'll have to write a script that describes the following series of commands:

- Drive a specified distance forward; this distance represents the square's side length.
- Turn 90 degrees left or right; it must always turn in the same direction.
- Repeat the previous two commands indefinitely.

This is the script we would write:

```
[152] [17] [137] [1] [44] [128] [0] [156] [1] [144]
[137] [1] [44] [0] [1] [157] [0] [90] [153]
```

Commands and arguments	Values	
Script	152	
Number of bytes	17	
Drive	137	
Velocity (300 mm/s)	1	44
Radius (Straight)	128	0
Wait for distance	156	
Distance (400 mm)	1	144
Drive	137	
Velocity (300 mm/s)	1	44
Radius (Counterclockwise)	0	1
Wait for angle	157	
Angle (90 degrees)	0	90
Restart script	153	

4.3 Getting to know the robot with the use of a serial cable

Before we could start really dealing with the robot we had to know how it worked and what it was capable of. For our first approach the easiest thing to do seemed to be plugging the serial cable into the robot and then trying and sending commands one at a time from the computer.

iRobot Corporation recommends using a freeware terminal program called *RealTerm* [29], *"especially designed for capturing, controlling and debugging binary and other difficult data streams"*, but we could not get our Create robot to work with this software so we had to search the internet for some other tool. Surprisingly, we found exactly what we were looking for in a robot's homonym software tool coded in Delphi: *Create* by Joe Lucia.

Joe Lucia [30] is an iRobot Create enthusiast who has developed many interesting projects with his own Create robots in his spare time, just for fun. He is also one of the most important

contributors in the official iRobot Create online forums [31]. His *Create* tool provides a very easy to use graphic interface that allows direct communication with the Create robot through any serial port. On this *Create* tool, most of the Create robot commands are linked to a button, such as the movement commands and the ones to read the robot's sensors. For those commands that cannot be executed by pressing a button it's also possible to send command byte streams to the robot like we would do with a terminal program such as *RealTerm*.

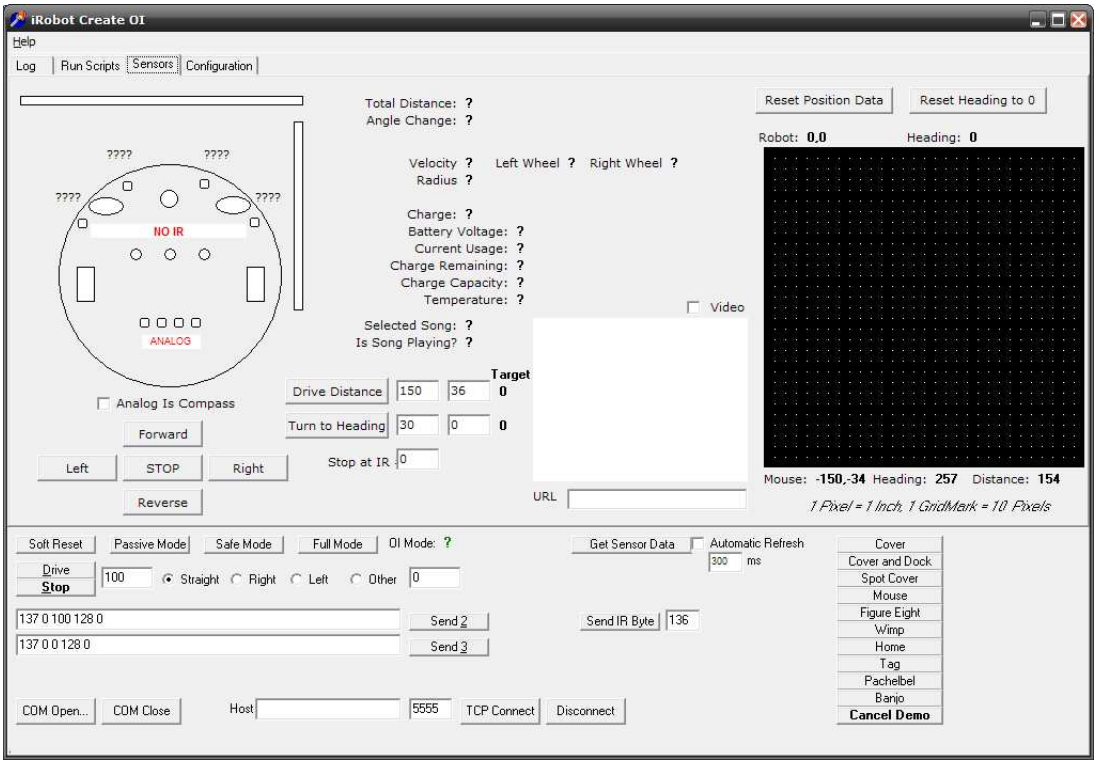


Figure 4.5: A screen capture of the *Create* software tool by Joe Lucia.

For these first behavior tests we used the robot's serial cable and therefore we could not allow the robot to get too far from the computer since the maximum distance was limited by the cable's length. We quickly decided we were ready to take another step and try the Bluetooth serial connection.

4.4 *PyRobot* interface and Bluetooth serial connection

Another of the very useful accessories of the Create robot is its Bluetooth Adapter Module (BAM) [32], which allows the user to wirelessly control the robot from up to 300 feet away through a Bluetooth serial connection. Obviously, it's also needed to have a Bluetooth dongle in the transmitter device (in our case a laptop), and we used a BlueSoleil Class 1 Bluetooth dongle. Once both Bluetooth devices are paired, a virtual serial port connection is created between them, and it works exactly like a tethered serial port connection. After trying the *Create* tool for wirelessly sending commands to the robot -thus testing that the Bluetooth serial connection worked- it was time to code our own solution.

When we started working on this part of our project it seemed clear that, very probably, our

biggest challenge would be to develop our own interface to communicate with the robot. We wanted to code it in Python, because our Statecharts are compiled into Python with AToM^3's SCC, and the linking of the different code components would thus be very straightforward. After looking into the iRobot Create Open Interface for a few days and concluding that it truly was going to be a difficult task, we came across *PyRobot*'s project web page on Google Code, pretty much by chance.

Figure 4.6: Bluetooth Adapter Module (BAM) for iRobot Create (left) and Bluetooth Class 1 dongle by BlueSoleil (right).

PyRobot [18] is an open source *"high-level Python interface, built on pySerial, to iRobot's Roomba and Create"*, developed by Damon Kohler and freely distributed under MIT license. *PyRobot* was exactly what we wanted and exactly in the way we wanted it, however, we adapted it to our specific needs by implementing some methods of our own. For example, we implemented the methods *LoadSong()* and *PlaySong()*, which correspond to two of iRobot Create's Open Interface commands; method *LoadSong()* sends opcode 140 (*Song*) to the robot, whereas method *PlaySong()* sends opcode 141 (*Play Song*).

Create's OI commands can be easily translated into methods thanks to *PyRobot* interface's *SerialCommandInterface* class. It provides a method called *__getattr__()*, which creates methods for opcodes on the fly. Each opcode method sends the opcode optionally followed by a string of bytes.

```
class SerialCommandInterface(object):
      ...
      def __getattr__(self, name):
          if  name in self.opcodes:
              def SendOpcode(*bytes):
                  logging.debug('Sending opcode %s.' % name)
                  self.Send([self.opcodes[name]] + list(bytes))
              return SendOpcode
          raise AttributeError
```

We also extended *PyRobot* interface's code with the implementation of a series of methods for the management of iRobot Create's sensors data. The main method we will use is *RequestAndGetSensorData(packet_id)*, being *packet_id* the identifier of the sensor data packet that we want to request from the robot. The method requests and obtains the sensor data packet in

question, decodes it into a Python library structure (a hashtable), and returns it. In order to get the data of a sensor in particular, the library must be indexed using the sensor's packet name as the key, i.e. *library['Bump right']* or *library['Battery temperature']*. All the code that we contributed to *PyRobot* interface can be found in *Appendix D*.

4.5 Combining *PyRobot* and Statecharts

In the following subsections we'll discuss our Create projects: a diverse set of Python applications in which we combined statecharts models -compiled into Python classes- with the *PyRobot* interface in order to develop statecharts modelled behaviors for our Create robot. Obviously, we started experimenting with small and simple statecharts, and kept adding more complexity to our models as we obtained satisfactory results and felt more confident. Our goal with these Create projects was to end up defining some statecharts modelled autonomous behaviors; we managed to do it somehow, but at a very basic level.

4.5.1 Simple console applications

Before we dived into designing complex behavior statecharts for our Create robot, we dedicated many days to the development of several small experiments the purpose of which was to become familiar with the *PyRobot* interface, and find out if we could easily combine it with our compiled statecharts classes. Each of these small experiments, which are coded in Python, consist of a main application file, a compiled statechart class, and *PyRobot* interface's Create class. Next we are going to show and explain two of these experiments.

Our very first statechart is the one captured in Figure 4.7, which describes a very simple movement routine: default state *stop* is automatically transferred to state *moveForward* by means of an unlabeled transition. The robot starts moving forward at a constant speed and, after exactly 4 seconds, state *moveForward* is transferred to state *turnLeft*, obviously causing the robot to start turning left. State *turnLeft* is automatically left after 2 seconds -which is the time it takes the robot to turn 90°- and the statechart's current state is transferred back to default state *stop*. The loop immediately starts all over again, causing the robot to drive in a square pattern.

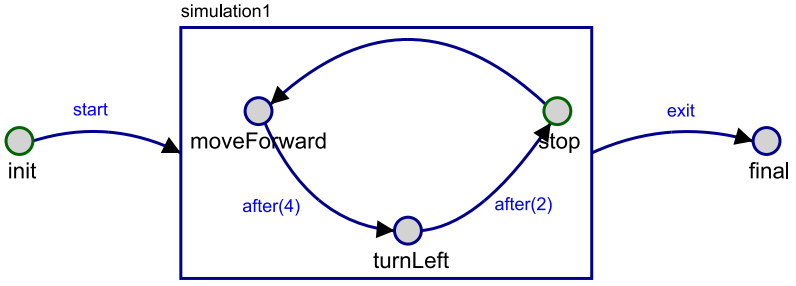


Figure 4.7: PyRobot + statecharts experiment #1.

Movement orders are given to the robot through calls to the *PyRobot* interface methods. The methods are called at state level, meaning that the code snippets are executed as state enter actions instead of event output actions.

- Entering state *moveForward*.

```
print "moving forward"
ctl.Stop()
ctl.Drive(ctl.vel, ctl.RADIUS_STRAIGHT)
```

- Entering state *turnLeft*.

```
print "turning left"
ctl.Stop()
ctl.Drive(ctl.vel, ctl.RADIUS_TURN_IN_PLACE_CCW)
```

- Entering state *final*.

```
print "finalizing execution..."
ctl.Stop()
```

After compiling this statechart with AToM^3's SCC, we import the generated Python class and import it into our main application file, the code of which is as follows.

```
import simulation1

# get bluetooth serial port id from user
port = raw_input("++ enter port:")
# declare instance of PyRobot's Create class
crt = Create(tty=port)
print "putting iRobot Create in full mode"
crt.Control()
self.crt.vel = 100
crt.RADIUS_STRAIGHT = RADIUS_STRAIGHT
crt.RADIUS_TURN_IN_PLACE_CCW = RADIUS_TURN_IN_PLACE_CCW
# declare instance of the compiled statechart class
sim = simulation1.simulation1()
sim.initModel()
sim.event("start", crt)
raw_input("******** PRESS ENTER TO EXIT ********")
sim.event("exit", crt)
```

When the event *exit* is narrow-casted to the simulation statechart, its current state transfers from composite state *simulation1* to state *final*, which terminates the statechart's execution.

For our second experiment we decided to introduce a bit more interaction with the Create robot, while keeping our statecharts model very simple. The behavior described by the statechart depicted in Figure 4.8 is almost totally autonomous, since the robot's movements depend on the data obtained from the robot's bumper sensors. Once the event *forward* is received and the state *MovingForward* is entered, the robot will start driving forward. The state has a self-transition that is automatically taken every one second, and the output action of which is to call the method *checkBumpers()*, which is defined in the simulation controller class. This method, in turn, calls another method of the *PyRobot* interface's Create class, *RequestAndGetSensorData()*, which returns the robot's sensor data. These data are checked in order to find out if the robot's bumpers have hit a wall or obstacle; if they have, the simulation controller class narrow-casts

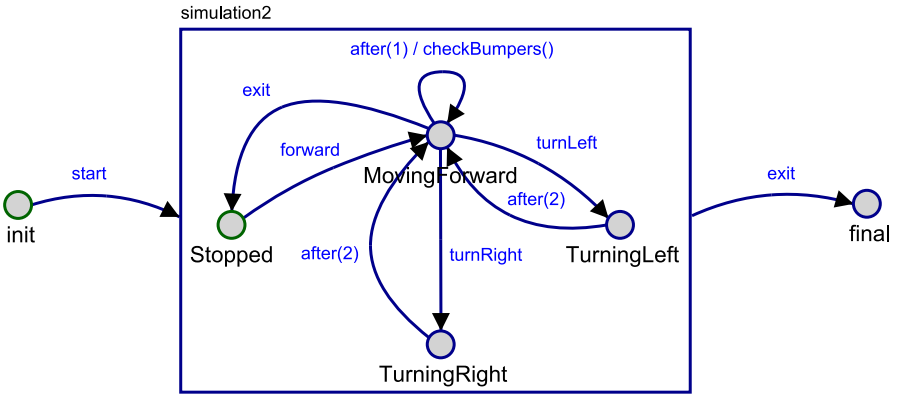


Figure 4.8: PyRobot + statecharts experiment #2.

an event to the statechart. Depending on which side bumper hit the obstacle, the narrow-casted event will be *turnLeft* -for the right bumper- or *turnRight* -for the left bumper-.

While in state *MovingForward*, if events *turnLeft* or *turnRight* are received, the system is transferred to states *TurningLef* or *TurningRight* respectively, causing the robot to turn in place (that is, without a horizontal displacement) in the corresponding direction for 2 seconds -as has been previously mentioned, it's the time it takes the robot to make a 90° turn-. After 2 seconds, the system is transferred back from either *TurningLeft* or *TurningRight* back to *MovingForward*. As a result of this strategy, the robot dodges the wall or obstacle it had hit, and starts driving forward once again.

Whereas in the previous experiment we defined the *PyRobot* interface method calls as state enter actions, in this experiment we define them as event output actions.

- Event *forward* taken.

```
print "moving forward"
ctl.crt.DriveStraight(ctl.crt.vel)
```

- Event *turnLeft* taken.

```
print "turning left"
ctl.crt.TurnInPlace(ctl.crt.vel, ctl.crt.TURN_LEFT)
```

- Event *turnRight* taken.

```
print "turning right"
ctl.crt.TurnInPlace(ctl.crt.vel, ctl.crt.TURN_RIGHT)
```

- Timed transitions leading from either *TurningLeft* or *TurningRight* to *MovingForward*

```
print "moving forward"
ctl.crt.Stop()
ctl.crt.DriveStraight(ctl.crt.vel)
```

The following is the code to our main application file:

```
import simulation2

class Controller2:
      def __init__(self, port):
          # declare instance of PyRobot's Create class
          self.crt = Create(tty=port)
          self.crt.vel = 100
          self.crt.STRAIGHT = RADIUS_STRAIGHT
          self.crt.TURN_RIGHT = 'cw'
          self.crt.TURN_LEFT = 'ccw'
          # declare instance of the compiled statechart class
          self.sim = simulation2.simulation2()
          self.sim.initModel()

      def checkBumpers(self):
          packet_id = 7,
          bumpers = self.crt.RequestAndGetSensorData(packet_id)
          if bumpers['Bump right']:
                  self.sim.event("turnLeft")
          elif bumpers['Bump left']:
                  self.sim.event("turnRight")

port = raw_input("++ enter port:")
# declare instance of controller class
ctl = Controller2(port)
print "+ putting iRobot Create in full mode"
ctl.crt.Control()
ctl.sim.event("start", ctl)
ctl.sim.event("forward")
raw_input("******** PRESS ENTER TO EXIT ********")
ctl.sim.event("exit")
```

In this second experiment as well, the statechart's execution is terminated right after the event *exit* is received.

4.5.2 *Create Remote*

One of the original Roomba robot accessories is its Standard Remote [33], an infrared remote control lets you control standard robot functions from across the room; the remote can also be used to control the Create robot. As a way to explore the many possibilities of the robot's behavior modelling while fully putting statecharts features into practice, we came up with the idea of developing our own virtual remote control: the Create remote.

CreateRemote is a simple program that uses the *PyRobot* interface to send the low-level instructions to the Create robot. This program features a graphic interface that resembles Roomba's remote control, and allows the user to interact with the Create Robot through pressing buttons that execute the actions previously defined in the robot's behavior statechart. *CreateRemote* consists of the following Python classes:

Figure 4.9: Original iRobot Roomba Standard Remote.

- *PyRobot.*

- *CreateRemoteStatechart.*
  This is the Python class we obtain from compiling our statechart in AToM^3. We can model as many behavior statechart as we want, all of them in different files, but the compiled file must always have this name. After being initialized by *CreateRemote_ Dynamic*, the statechart class receives a reference to the *CreateRemote_ Controller* class, sent as a parameter with the *start* event.

  ```
  ctl=[PARAMS]
  ```

  This way, the modeller can directly and easily access Create robot's functions by typing:

  ```
  ctl.create.function_name()
  ```

- *CreateRemote_ Controller.*
  The controller class declares an instance of PyRobot interface's Create class, therefore the controller is the class in charge of calling the interface's methods, directly interacting with the Create robot. It also has a reference to the application's statechart class and the GUI, because the button click events are defined in the controller class as well.

  ```
  def __init__(self, port):
      self.create = Create(tty=port)
      self.statechart = None
      self.GUI = None
      self.create.Control()
  ```

- *CreateRemote_ Dynamic.*
  The dynamic part of the application is the statechart and, thus, *CreateRemote_ Dynamic* simply declares an instance of the *CreateRemoteStatechart* class. Its constructor function receives a reference to the controller class, so that it can pass it on to the statechart class with the *start* event, after the model is initialized.

```
def __init__(self, _controller):
    self.controller = _controller
    self.statechart = CreateRemoteStatechart()
    self.statechart.initModel()
    self.controller.bindDynamic(self.statechart)
    self.statechart.event(“start”, self.controller)
```

- *CreateRemote_ Static.*
  The static part of the application binds the GUI with the controller.

```
def __init__(self, _parent, _controller):
    Frame.__init__(self, _parent)
    self.parent = _parent
    self.controller = _controller
    self.controller.bindStatic(self)
    self.image = PhotoImage(file="create_standard_remote.gif")
    self.lastPressed = “”
    self.parent.protocol(“WM_DELETE_WINDOW”, self.controller.window_close)
    self.createWidgets()
```

- *CreateRemoteGUI.* We have implemented the graphical user interface in this class, including the definition of the coordinates which delimitate the clickable areas or “virtual buttons” of the remote control picture.

```
def __init__(self, _parent, _port):
    self.controller = CreateRemote_Controller(_port)
    self.staticGUI = CreateRemote_Static(_parent, self.controller)
    self.dynamicGUI = CreateRemote_Dynamic(self.controller)
```

The *CreateRemote* statechart reacts to a predefined set of events. Since the only way to communicate with the robot is by means of the virtual remote controller buttons -as captured in Figure 4.10-, we can only use as many event names as buttons has the remote. But, if we are creative enough, we can do many things with this small set of events:

- *moveForward.*
- *turnLeft.*
- *turnRight.*
- *moveBackwards.*
- *stop.*
- *changeSpeed.*
- *playSong.*

These external events are fired by the user -by pressing a remote button-, and are broadcasted to the *CreateRemote* statechart through the controller class, but we can define as many internal events in the statechart as we wish.

Figure 4.10: Our own virtual version of a Create Standard Remote.

The remote control picture was manually edited from the original Roomba standard remote picture, and the button functions were redefined as follows (see Figure 4.10):

1. **Music button**. When pushed, the controller sends the event *playSong* to the *CreateRemote* statechart.
2. **Acceleration button**. When pushed, the controller sends the event *changeSpeed* to the *CreateRemote* statechart.
3. **Sensor button**. When pushed, the controller sends the event *requestSensors* to the *CreateRemote* statechart.
4. **Stop button**. When pushed, the controller sends the event *stop* to the *CreateRemote* statechart.
5. **Direction buttons**. When pushed, the controller sends either the events *moveForward*, *turnLeft*, *turnRight*, or *moveBackwards* to the *CreateRemote* statechart, depending on which of the direction buttons has been pressed.

Each remote buttons is associated to an event in the *CreateRemote_ Controller* class, as follows:

```
## Interface for the GUI
def turnLeftPressed(self):
    self.previous = 'turnLeft'
    self.statechart.event('turnLeft')

def turnRightPressed(self):
    self.previous = 'turnRight'
    self.statechart.event('turnRight')
```

```
def moveForwardPressed(self):
    self.previous = 'moveForward'
    self.statechart.event('moveForward')

def moveBackwardsPressed(self):
    self.previous = 'moveBackwards'
    self.statechart.event('moveBackwards')

def stopPressed(self):
    self.previous = None
    self.velocity = 100 # reset velocity
    self.statechart.event('stop')

def playSongPressed(self):
    self.statechart.event('playSong')

def changeSpeedPressed(self):
    # Increase Create's speed by 100mm/s or reset to 100mm/s if it's 500mm/s
    self.statechart.event('changeSpeed')

def sensorsPressed(self):
    packet_id = 0,
    # request all sensor packets and print the data in the console window
    self.create.RequestAndPrintSensorData(packet_id)
```

Therefore, this button-event association can be easily changed and different event names can be used; for example, we may not want to use the event *changeSpeed* and use an event called *changeSense* instead. According to our *CreateRemote* statecharts model definition, the event *changeSense* will cause the robot to turn 180° and keep driving forward -if it was-. We might associate the *Acceleration button* to this new event, but we could as well choose any other button. Of course, we could also edit the Create remote picture to add more buttons to it, or design and draw a completely different remote, but we wanted to remain faithful to the real thing.

4.5.2.1 Statechart #1: Basic robot movement

The first *CreateRemote* statecharts modelled behavior is really simple. As captured in Figure 4.11, there are only 5 main states to it, apart from the *init* state, which is only entered at the very beginning of the execution and never again. We can see that states *stopped, movingForward, turningLeft*, and *turningRight* are mutually exclusive (OR-states), which means that the system will only be in one of the four states at a time. As well, each of the four states can only be entered through a single transition and a single event; all of these transitions are hyperedges connecting the states with their parent composite state. Therefore, it doesn't matter in which state the system is when an event is triggered, the system will automatically be transferred to the corresponding state; e.g.: if the system is *stopped* and the event *movingForward* is triggered, it will transfer the system to *movingForward*. When the *changeSpeed* event is triggered, the system will leave composite state *CreateRemoteStatechart* and transfer to the *changingSpeed* state

but, immediately after, it will go back to the *stopped* state, since it is default amongst *stopped*, *movingForward*, *turningLeft*, and *turningRight*, and the hyperedge connecting the *changingSpeed* state to the *CreateRemoteStatechart* composite state is unlabeled and unconditional.

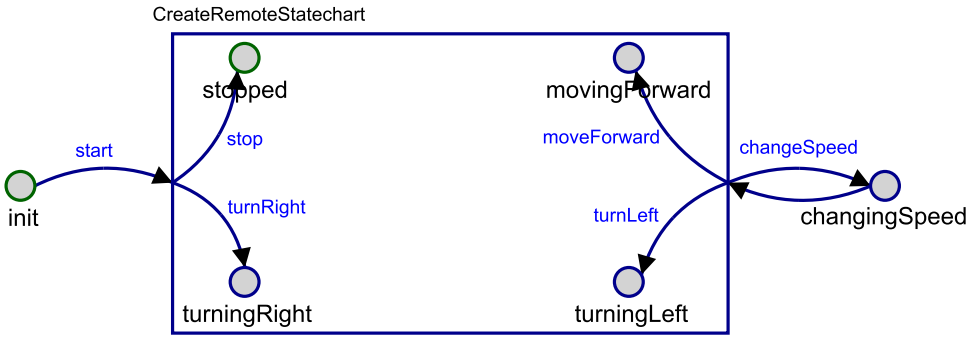


Figure 4.11: Simple *CreateRemote* movement statechart.

The following is the Python code behind the statechart's states and transitions:

- *stopped*.
```
print 'stopping'
ctl.create.Stop()
```

- *movingForward*.
```
print 'moving forward'
ctl.create.Stop()
ctl.create.Drive(ctl.velocity, ctl.STRAIGHT)
```

- *turningRight*.
```
print 'turning right'
ctl.create.Stop()
ctl.create.TurnInPlace(ctl.velocity, ctl.TURN_RIGHT)
```

- *turningLeft*.
```
print 'turning left'
ctl.create.Stop()
ctl.create.TurnInPlace(ctl.velocity, ctl.TURN_LEFT)
```

- *changingSpeed*.
```
ctl.velocity = ctl.velocity % 500 + 100
print 'current speed is', ctl.velocity
```

4.5.2.2 Statechart #2: Introducing orthogonal components

In this second *CreateRemote* statechart we introduce orthogonality, although keeping it simple. The orthogonal component *square* autonomously controls the robot's movement and makes it drive in a square. The user interaction by means of the virtual remote is all in the orthogonal

component *music*. When the execution starts, default state *silent* is entered and three songs are loaded into the robot. The first time the event *playSong* is received, song number one is played and the system is transferred to state *song1*. While in this state, song number one is played whenever the event *playSong* is received again. However, if the event *changeSpeed* is received instead, not only the robot increases its current velocity, but the system transfers to state *song2* and song number two is played as well. The same functionality applies to the transition connecting states *song2* and *song3*. Finally, while in state *song3*, if event *changeSpeed* is received the system transfers back to state *song1*, and the cycle starts all over again.

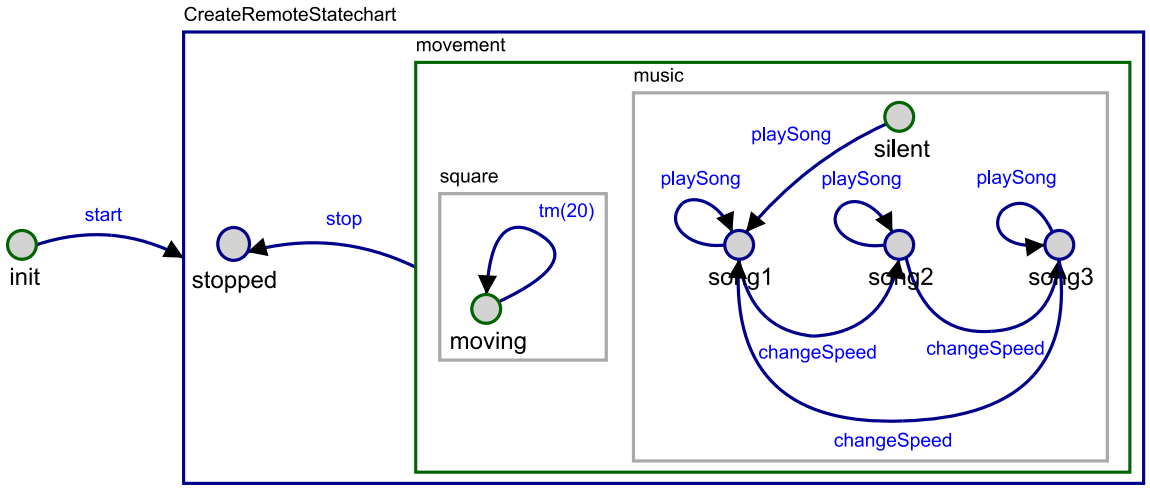


Figure 4.12: Introducing orthogonal components but keeping it simple.

This is the Python code behind the statechart's states and transitions:

- *stopped*.
```
print 'stopping'
ctl.create.Stop()
```

- *moving*.
```
print 'moving forward, current speed = 50mm/s'
ctl.create.Drive(50, ctl.STRAIGHT)
print 'wait distance is 400mm'
ctl.create.WaitDistance(400)
print 'turning left'
ctl.create.TurnInPlace(50, ctl.CCW)
print 'waiting for angle 90 degrees'
ctl.create.WaitAngle(90)
ctl.create.Stop()
```

- *silent*.
```
print 'silent mode: loading all songs'
song = 0,18,55,32,55,32,55,32,51,16,58,16,55,32,51,16,58,16,55,32,62,32,62
32,62,32,63,16,58,16,54,32,51,16,58,16,55,32
```

```
ctl.create.LoadSong(song)
song = 1,12,67,30,70,30,72,40,67,30,70,30,73,20,72,40,67,30,70,30,72,40,70,
40,67,60
ctl.create.LoadSong(song)
song = 2,15,45,90,48,60,48,30,50,30,50,60,53,15,52,15,53,15,52,15,53,15,52,
20,48,30,48,30,50,30,50,30
ctl.create.LoadSong(song)
```

- *song1.*
  ```
  print 'playing "The Imperial March"'
  ctl.create.PlaySong([0])
  ```

- *song2.*
  ```
  print 'playing "Smoke on the water"'
  ctl.create.PlaySong([1])
  ```

- *song3.*
  ```
  print 'playing "Ironman"'
  ctl.create.PlaySong([2])
  ```

4.5.2.3 Statechart #3: Extensive use of orthogonal components

This is already getting interesting, let's make another step forward. In our third *CreateRemote* statechart -captured in Figure 4.13-, we have used even more orthogonal components, thus defining a quite complex model. The biggest orthogonal components are *moving* and *turning*, which are obviously in charge of the robot's movements. The behavior described by both components is the following: when the execution starts, default states *notMoving* and *notTurning* are entered. Events *moveForward* and *moveBackwards* transfer the system to states *movingForward* and *movingBackwards* respectively. If event *moveForward* is received while in state *movingBackwards* the system transfers to state *movingForward*, and viceversa.

The orthogonal component *turning* has a very similar behavior; while in state *notTurning*, events *turnLeft* and *turnRight* transfer the system to states *turningLeft* and *turningRight* respectively. If event *turnLeft* is received while in state *turningRight* the system transfers to state *turningLeft*, and viceversa. However, if event *turnLeft* is received while the current state is *turningLeft*, or event *turnRight* is received while the current state is *turningRight*, the orthogonal state *radius* takes part in the game. This component has one single state, *changingRadius*, with two self-transitions labeled *turnLeft* and *turnRight* respectively. When either of this self-transitions is taken, event *changeRadius* is generated and a method of the controller class, which increases the turning radius of the robot, is called. The broadcasted event, *changeRadius*, is received in orthogonal state *turning* and the new radius is applied, causing the robot to turn describing a wider angle.

If a turning event is received in the *CreateRemote* statechart prior to a movement event, the robot starts turning in place -that is, spinning-. Going back to orthogonal state *moving*, note that states *movingForward* and *movingBackwards* both have self-transitions, labeled *moveForward* and *moveBackwards* respectively. Both self-transitions, as well, produce the same output action when taken: the generation of event *stopTurning*. Whenever event *stopTurning* is received in the *CreateRemote* statechart, it causes the system to transfer to state *notTurning* and, consequently,

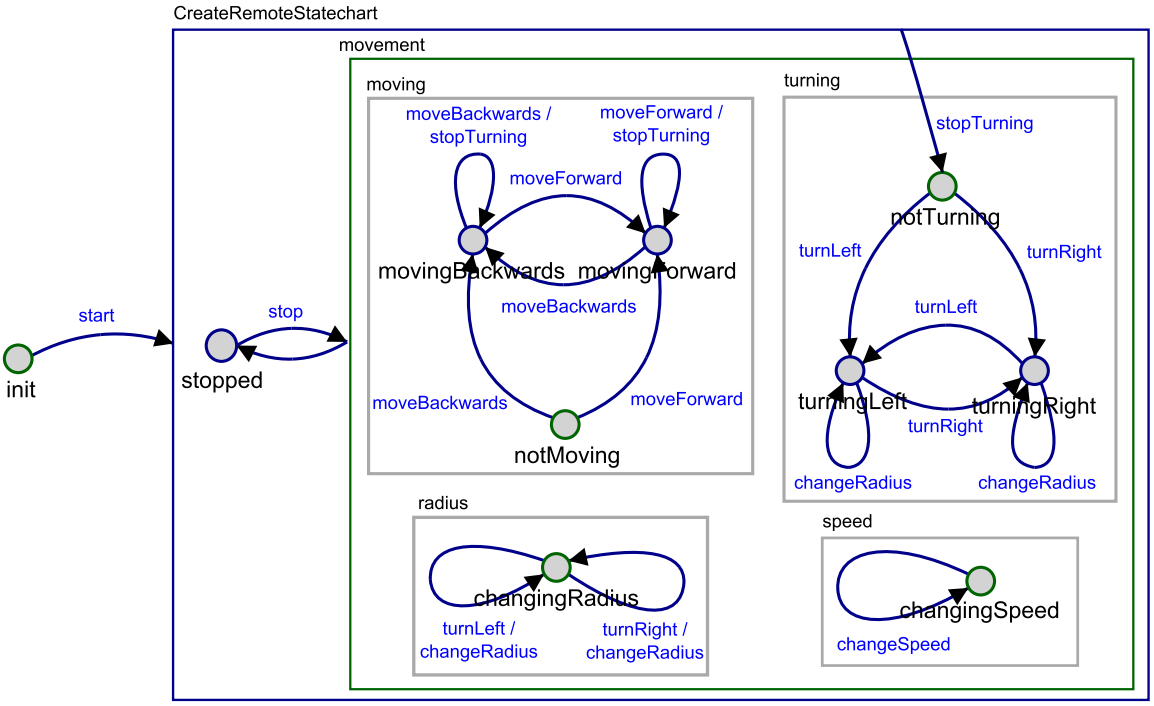


Figure 4.13: Using even more orthogonal components.

the robot to stop turning if it was. In short, the behavior of *moving*'s self-transitions defines that if the robot is moving and turning at the same time, and either event *moveForward* or event *moveBackwards* are received, the robot will stop turning and start driving straight. This is the Python code behind the statechart's states and transitions:

- *stopped*.
```
print 'stopping'
ctl.create.Stop()
```

- *movingForward*.
```
ctl.radius=ctl.STRAIGHT
ctl.setRadius(ctl.radius)
ctl.create.Drive(ctl.velocity, ctl.radius)
print 'moving forward, speed=',ctl.velocity,', radius=',ctl.radius
[EVENT("stopTurning")]
```

- *movingBackwards*.
```
ctl.radius=ctl.STRAIGHT
ctl.setRadius(ctl.radius)
ctl.create.Drive(ctl.velocity*(-1), ctl.radius)
print 'moving backwards, speed=',ctl.velocity,', radius=',ctl.radius
[EVENT("stopTurning")]
```

- *turningLeft*.
  - Entered through *turnLeft*, transferred from state *notTurning*.
    ```
    print 'turning.turning left'
    ctl.LEFT=0
    ```
  - Entered through *turnLeft*, transferred from state *turningRight*.
    ```
    ctl.LEFT=ctl.RIGHT
    ctl.setRadius(radius=ctl.LEFT*(-1))
    ```
  - Entered through *changeRadius*.
    ```
    print 'turning.turning left'
    ctl.LEFT=0
    ```

- *turningRight*.
  - Entered through *turnRight*, transferred from state *notTurning*.
    ```
    print 'turning.turning right'
    ctl.RIGHT=0
    ```
  - Entered through *turnRight*, transferred from state *turningLeft*.
    ```
    ctl.RIGHT=ctl.LEFT
    ctl.setRadius(radius=ctl.RIGHT*(-1))
    ```
  - Entered through *changeRadius*.
    ```
    print 'turning.turning right'
    ctl.RIGHT=0
    ```

- *changingRadius*.
  - Entered through *turnLeft*.
    ```
    ctl.turnLeft()
    [EVENT("changeRadius")]
    ```
  - Entered through *turnRight*.
    ```
    ctl.turnRight()
    [EVENT("changeRadius")]
    ```

- *changingSpeed*.
  ```
  ctl.velocity = ctl.velocity % 500 + 100
  ctl.echoPreviousEvent()
  ```

4.5.2.4 Statechart #4: Easily combining statecharts to create new statecharts

We have mentioned in previous chapters that one of the most interesting and useful features of statecharts is modularity. An example of it is the statecharts model captured in Figure 4.14: we have made it up by combining several orthogonal components from the previously discussed *CreateRemote* statecharts models.

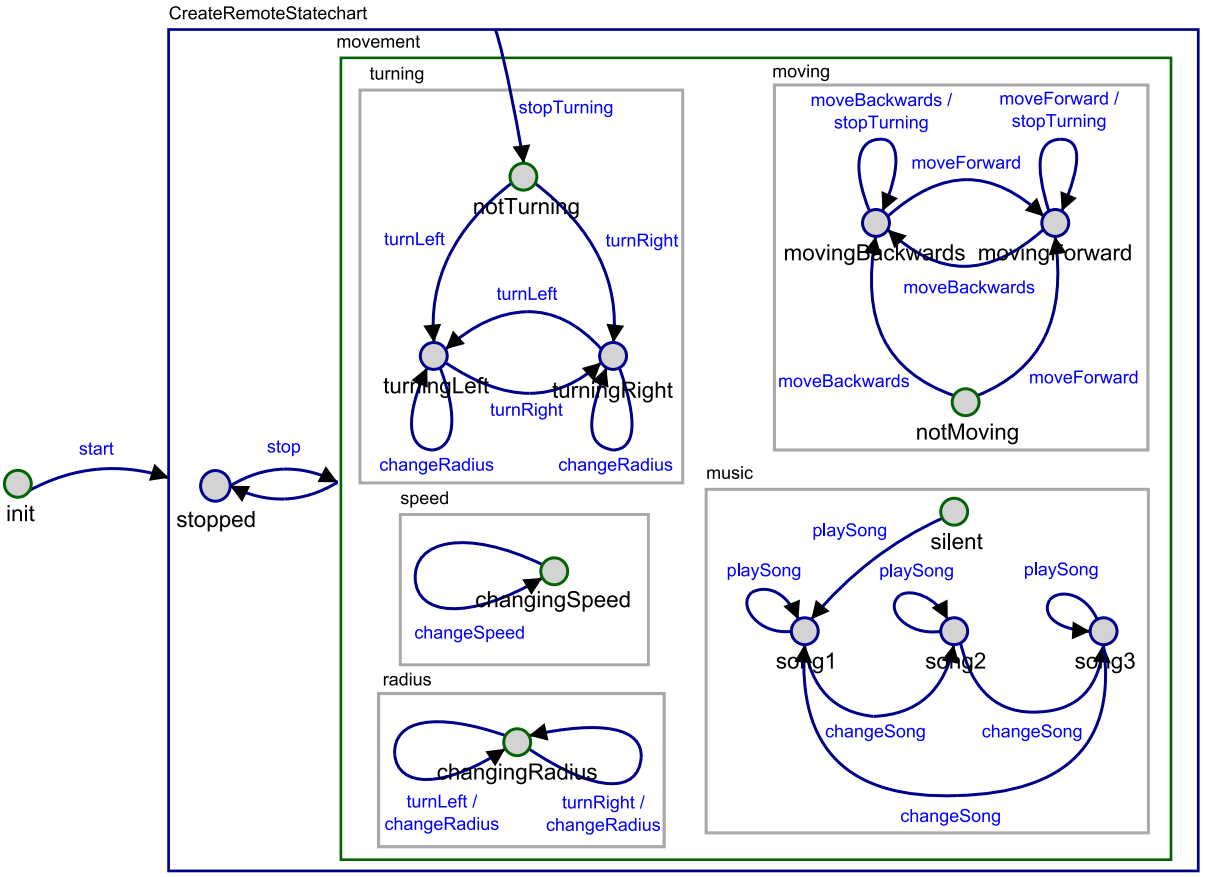


Figure 4.14: The result of combining orthogonal components from other statecharts.

4.6 Statecharts modelled autonomous behavior

Originally, it was amongst the objectives of this Master Thesis to develop a Robot Wars project. The intention was to turn our Create into a battle robot that could confront other Creates; we could even get multiple Create robots to fight each other in a robot brawl. Soon after we started working on this Master Thesis we discarded the Robot Wars project because it entailed too much time that we did not have, and too much hardware assembly that we could not guarantee, for we have very little knowledge of electronics. But still, Robot Wars kept being the aim of this Master Thesis: our work might be taken up and continued by some other student at the Modelling, Simulation and Design Lab at McGill University.

So, if we were to turn the iRobot Create into an autonomous battle robot, we should be able to define autonomous behavior statecharts models. The robot should get input from the environment surrounding it and generate some output in response to it. The only way for the robot to get input from the environment is by means of its built-in sensors. The robot must be constantly checking the data provided by its sensors and react accordingly: if the front bumpers say the robot has hit a wall, the robot should turn around; if the drop sensors say there is a hole in the way, the robot should stop immediately and make a detour. The robot must not only react to external events, but to internal events as well. For example, if the battery sensor indicates that the current battery charge level is low, the robot should go back to its docking station in order to recharge.

We didn't have much time left to delve deeper into this subject, but during our last days at MSDL we developed several statecharts modelled autonomous behavior experiments for our Create robot. We have combined some of these experiments in order to create the statechart captured in Figure 4.15. In this statecharts model we have taken advantage of the technique of using timed self-transitions in order to force the checking of the guards of each state's outgoing transitions. This way, the robot is constantly requesting the data from its sensors and consequently getting information about the environment. We developed a new controller class for the autonomous behavior experiments; in this controller we implemented a series of methods, each of which requests a specific data from the robot's sensors and returns a certain value according to it. The controller's code is as follows:

```
class Controller:
      def __init__(self, port):
          self.create = Create(tty=port)
          self.create.STRAIGHT = RADIUS_STRAIGHT
          self.create.TURN_RIGHT = 'cw'
          self.create.TURN_LEFT = 'ccw'
          self.create.vel = 100
          self.distance = 0

      def checkBumpers(self):
          packet_id = 7,
          data = self.create.RequestAndGetSensorData(packet_id)
          if data['Bump right'] or data['Bump left']:
               return True
          return False
```

```
def checkDrops(self):
    packet_id = 7,
    data = self.create.RequestAndGetSensorData(packet_id)
    if data['Wheel drop right'] or data['Wheel drop left']:
        return True
    if data['Wheel drop caster']:
        return True
    return False

def checkBattery(self):
    packet_id = 25,
    data = self.create.RequestAndGetSensorData(packet_id)
    if int(data['Distance']) < 10000:
        return True
    return False

def traveledDistance(self):
    packet_id = 19,
    data = self.create.RequestAndGetSensorData(packet_id)
    self.distance += int(data['Distance'])
    return self.distance
```

The statecharts model -captured in Figure 4.15-, consists of five orthogonal components, the behavior of each of which is the following:

- *movement*. Default state *moveForward* has an outgoing unlabeled transition with a guard. The guard is the call to the controller's method *traveledDistance()*, which returns the robot's total traveled distance. Thus, the guard declares that the transition will be taken when the value returned by *traveledDistance()* equals or exceeds 50000 millimeters. When the transition is taken the system is transferred to state *turnAround*, the enter action of which causes the robot to stop driving and turn 180 degrees. Two seconds after, state *turnAround* is exited and the system is transferred back to state *moveForward*. The robot starts driving straight again and the total traveled distance is reseted.
- *bumpers*. Default state *checkBumpers* has an outgoing unlabeled transition with a guard. The guard is the call to the controller's method *checkBumpers()*, which requests the data from the robot's front bumpers and returns a boolean value: 'False', if none of the front bumpers is bumping against anything, or 'True', if any of the bumpers is bumping against something. Therefore, the guard declares that the transition will be taken when *checkBumpers()* return 'True' and, consequently, at least one of the robot's front bumpers is bumping against something. When the transition is taken the system is transferred to state *alarm* and the event *playSong1* is broadcasted. Two seconds after, state *alarm* is exited and the system is transferred back to state *checkBumpers*.
- *music*. This orthogonal component has only one state, *silent*; while in this state the robot is, as its name implies, "silent". When either event *playSong1* or event *playSong2* are received, the corresponding transition is taken and an output action is executed: the robot plays a song (song number one is "The Imperial March" and song number two is "Smoke on the water").

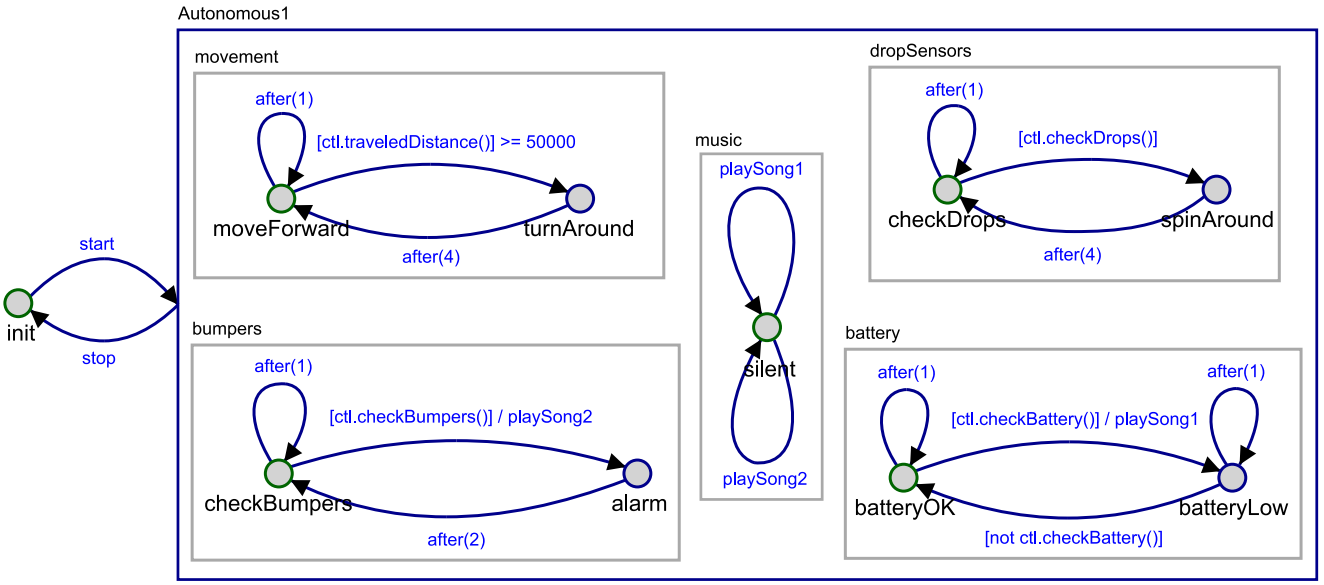


Figure 4.15: Statecharts modelled autonomous behavior.

- *dropSensors*. Default state *checkDrops* has an outgoing unlabeled transition with a guard. The guard is the call to the controller's method *checkDrops()*, which requests the data from the robot's drop sensors and returns a boolean value: 'False', if all of the robot's wheels are raised, or 'True', if any of the robot's wheels is dropped. Thus, the guard declares that the transition will be taken when *checkDrops()* returns 'True' and, consequently, at least one of the robot's wheels is dropped. When the transition is taken the system is transferred to state *spinAround*, the enter action of which causes the robot to stop driving and start turning in place. Four seconds after, state *spinAround* is exited and the system is transferred back to state *checkDrops*. As an output action to this timed transition the robot starts driving straight again.
- *battery*. Default state *batteryOK* has an outgoing unlabeled transition with a guard. The guard is the call to the controller's method *checkBattery()*, which returns a boolean value: 'False', if the battery charge level is still ok, or 'True', if the battery charge level is low. Therefore, the guard declares that the transition will be taken when *checkBattery()* returns 'True' and, consequently, the battery charge level is low. When the transition is taken the system is transferred to state *batteryLow* and the event *playSong2* is broadcasted. State *batteryLow* also has an outgoing unlabeled transition, but its guard declares the opposite to that of state *batteryOK*: the transition will be taken when method *checkBattery()* returns 'False' and, consequently, the battery charge level is ok. The system is then transferred back to state *batteryOK*.
  Note that when state *batteryLow* is entered, the robot should be commanded to go back to its docking station in order to recharge, otherwise the battery charge level would never increase and state *batteryLow* would never be exited.

Finally, this is the Python code behind the statechart's states and transitions:

- *moveForward*.

  - Entered through *after(4)*.
    ```
    ctl.create.Stop()
    ctl.create.Drive(ctl.create.vel, ctl.create.STRAIGHT)
    ```

- *turnAround*.
  ```
  print 'limit distance exceeded, turn around'
  ctl.create.WaitAngle(180)
  ctl.distance = 0 # reset distance
  ```

- *silent*.
  - Entered through *playSong1*.
    ```
    print 'playing "The Imperial March"
    ctl.create.PlaySong([0])
    ```
  - Entered through *playSong2*.
    ```
    print 'playing "Smoke on the water"
    ctl.create.PlaySong([1])
    ```

- *checkDrops*.
  - Entered through *after(4)*.
    ```
    ctl.create.Stop()
    ctl.create.Drive(ctl.create.vel, ctl.create.STRAIGHT)
    ```

- *spinAround*.
  ```
  print 'drop found, start spinning'
  ctl.create.Stop()
  ctl.create.TurnInPlace(ctl.create.vel, ctl.create.TURN_LEFT)
  ```

4.7 Other interesting iRobot Create projects

When facing a new project it's very common, even recommended, to start with some information gathering and some investigation, in order to find the answer to questions such as: "Has anyone developed anything similar before? Is there any piece of code, design, or idea that we can take advantage of?" When searching the internet for projects developed with iRobot Create robots, we we're surprised to discover that it's a very popular research tool in universities and companies worldwide, even amongst robotics enthusiasts. As a way to demonstrate the many possibilities that iRobot Create offers and broaden the reader's horizons out of the scope of this project, we'll comment some of the most interesting projects developed with Create robots, all of them published and reviewed on the official iRobot Create website [34].

4.7.1 Create Robot laser tag

We were most impressed to discover this Create project, and at the same time we were a bit disappointed. We were impressed because this was exactly our own project's main objective (iRobot Create Wars), and we were disappointed to realize that our idea was not as original

as we had thought and that someone had already developed it. Anyway, our approach was still different, since we pretended to model our robot's behavior using statecharts, instead of coding its AI. In the end, though, robot battles were ruled out of this Master Thesis's objectives due mainly to the lack of time and of a second Create robot.

The Create Robot laser tag project is definitely a very fun and entertaining proposal: the developers attached laser beams, video cameras, and wifi connection to the top of their Create robots; as well, there are sensors attached all over the robots's plastic bodies. The robots are programmed to be remote controlled by means of XboX gamepads, and the trigger buttons of the gamepads actually trigger the robots's laser beams, thus "shooting" blindly in the directions the robots are facing. Scoring a successful hit (or multiple hits) in those areas designated by the sensors can kill a robot or slow it down. The example video provided by the project developers shows how fast and smoothly the robots response to the remote control.

4.7.2 Rangoli

"Rangavalli", popularly known as "Rangoli", is a traditional decorative art which is drawn on the floor in front of houses and worship places in India. The print head of an old printer is mounted on the back of the Create along with a small dispensing nozzle in a box. The program that controls the robot and other printing parameters is written in the Create Command Module. As well, there is a program that converts bitmaps to an array of numbers that the robot can understand.

4.7.3 Fridgemate

A robot arm is mounted on the iRobot Create. The arm movement (extending and bending, lowering and lifting, grabbing and releasing) is controlled with a set of servos and motors. A Create docking station is placed 6" far from the fridge door in order for the iRobot Create to locate the fridge. The robot grabs the fridge door and then moves backwards to open it. Then it moves forward again -into the fridge- and the arm starts trying to grab a can; the arm keeps lowering and trying can grabs until a can is sensed in the hand. When the robot finally has a can, it goes to its predetermined destination and drops the can.

4.7.4 Robomaid

Very similar to the Fridgemate, the Robomaid is, as its name suggests, a robotic maid that "cleans your room". There is a robot arm mounted on the iRobot Create that is controlled with 6 servo motors. There is also an infrared sensor on the robot arm that points to the ground immediately in front of the robot. If the infrared sensor detects an increase in the floor elevation, and the robot doesn't bump into it when moving forward, the robot assumes that a small, pickable object has been detected. The infrared sensor is again used to do a detailed scan of the area in order to find the highest point in front of the robot. The robot then attempts to pick up the object; if it successes, the robot goes back to its docking station, which is next to a bin, and deposits the object in the bin.

4.7.5 Bionic hamster

There isn't much information about this project on the iRobot Create website, but a video that proves that it's developed and it works: a hamster is located on the back of the iRobot Create, inside a clear plastic sphere. By running in the sphere, the hamster controls where the robot moves and its speed.

5
Economic study

The pie charts captured in the following figures describe how the total execution time of this Master Thesis was distributed amongst the several stages that it consisted of. The time is measured in hours per stage. Note that the longest stage, as captured in Figure 5.2, was by far the writing of the present Master Thesis report; this, though, does not necessarily mean that it was most difficult stage.

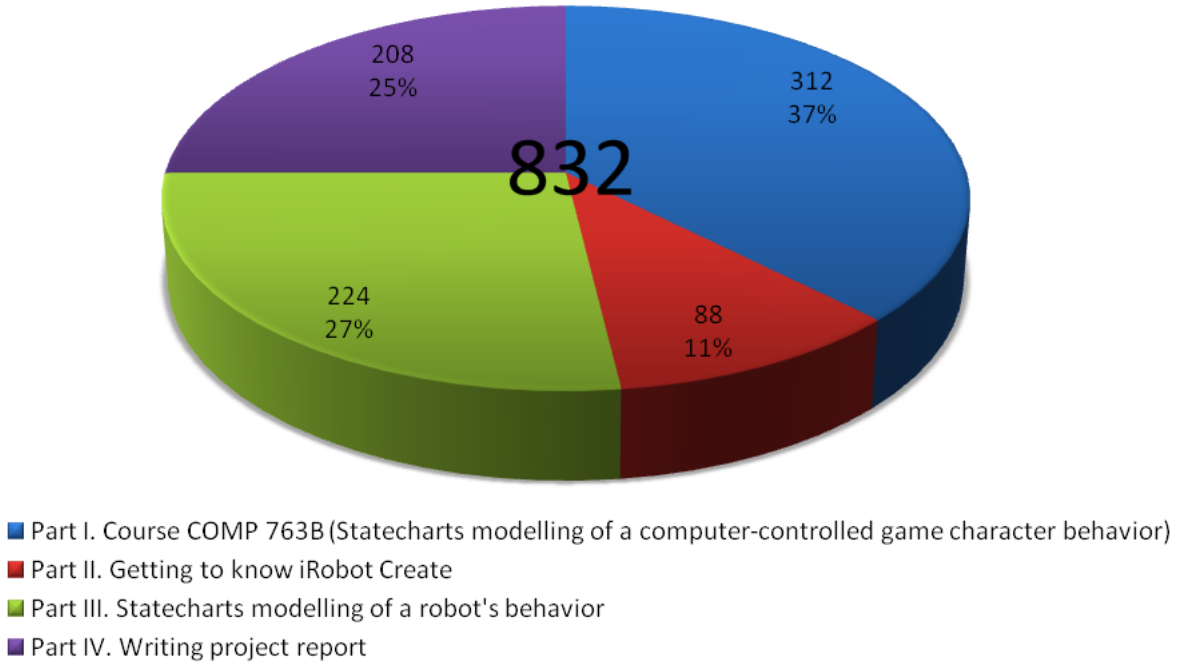


Figure 5.1: Time per stage distribution pie chart.

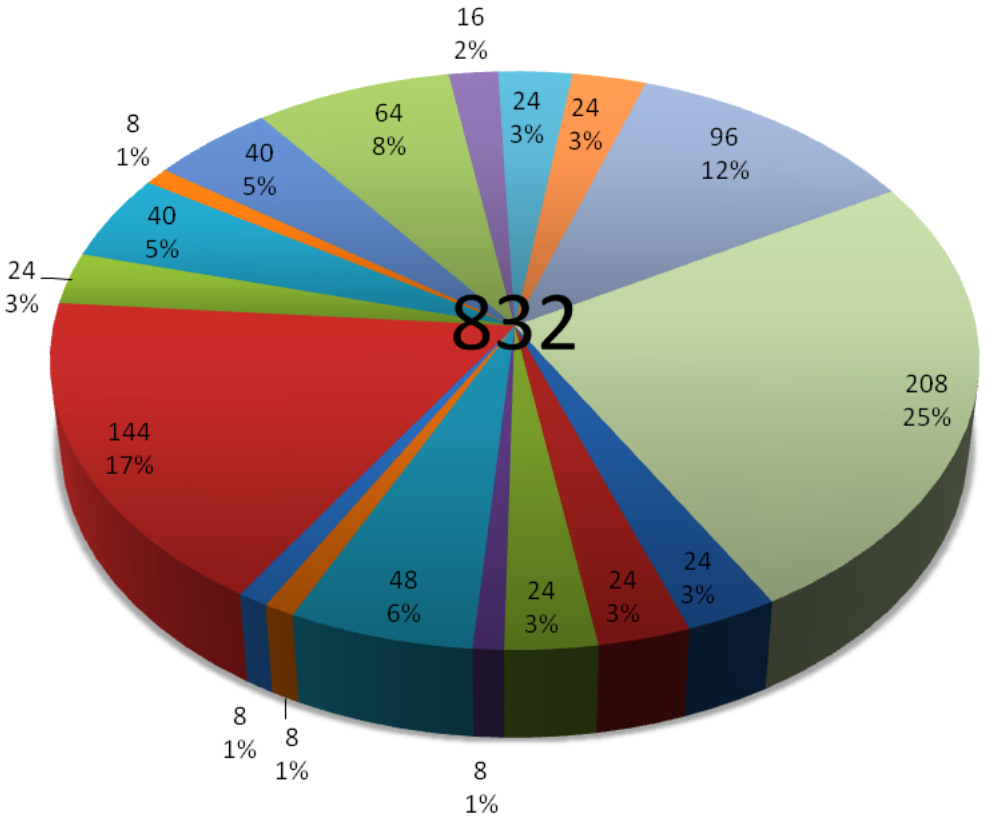


Figure 5.2: Detailed time per stage distribution pie chart.

6 Conclusions and Future lines

6.1 Conclusions

6.1.1 Project conclusions

Artificial intelligence and robot programming are not novel concepts at all, but this Master Thesis provides a different approach to both: behavior modelling. Most people with limited or non-existent knowledge on AI programming would be able to describe a reactive system's behavior only using the language; many of these people would even be capable of describing it with a diagram. AI programming is always bound to a determined programming language and, therefore, to operating systems's structural limitations. Our proposal is that AI specification should be done at a higher level of abstraction, that AIs should be modelled instead of coded: the AI modeller needn't know any programming language and he/she can focus on the logic of the model he/she is building. Since the main focus of the models is to define reactions to events, an event-based modelling formalism seems to be the most natural choice.

In short, this Master Thesis goal has been to demonstrate the proposal that modelling formalisms can greatly simplify AI specification, and we've undertaken several objectives in order to achieve that goal and illustrate our theoretical basis with practical examples. Our first objective -and *conditio sine qua non* it would not have been possible to undertake the following ones- was to learn the fundamentals of modelling formalisms and, obviously, to learn about some well known formalisms which are most commonly used in the modelling of reactive systems: the chosen ones were Petri Nets, DEVS, and, first and foremost, Statecharts.

We explored statecharts modelling of a video game AI, performing further research and practical development on the original work of Prof. Dr. Hans Vangheluwe, Prof. Jörg Kienzle, and Alexandre Denault from the School of Computer Science of McGill University, who are co-authors of a paper presented at the *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems* (MoDELS) entitled *Model-Based Design of Computer-Controlled Game Character Behavior*. In this paper, the authors introduced the video game AI modelling concept, stating that it could substitute AI programming with great results, and they proposed a practical experiment to demonstrate their approach: they would synthesize C++ code out of their statecharts models, and insert this code into the Tank Wars main game loop. EA Tank Wars is both a simulation video game environment and a student AI competition, so it was the perfect choice to prove, all in one, that behavior modelling is programming language independent, operating system independent, and genre independent -when it comes to video games in particular-.

Our AI is layered, with each layer being at a different level of abstraction. The AI consists of several components that we modelled separately using statecharts, which were later compiled

into Python classes; this is where our work started differing from that of Prof. Vangheluwe et. al.. These classes were put together and linked to each other by means of the use of a controller class -which was also programmed in Python-. The controller class has an instance of each of the AI's component classes, and thus can be seen as the tank itself; the components's statecharts interact with one another through event narrow-casting. We developed a simulation project -along with its really simple Python simulation environment-, which slightly tried to resemble EA Tank Wars video game, the only purpose of which was to put our modelled computer-controlled game character behavior to the test.

We did not stop to rejoice in our success, since our next objective was set a bit further both in time and complexity: we had to model the behavior of a robot. Not just any robot, but the Modelling, Simulation and Design Lab's own robot, which is an iRobot Create robot, a smaller brother and prototype of the world famous vacuuming robot, iRobot Roomba. More specifically, our next objective was to develop a series of applications in which we would control our Create wirelessly -by means of a Bluetooth connection-, using statecharts modelled behaviors and the least possible user interaction. Ultimately, the robot should be able to move completely autonomously. In order to achieve this objective, we needed an interface with which to communicate with the robot through a serial port connection. Searching the internet for a little help on this matter, we stumbled upon *PyRobot*, an open source, high-level Python interface, built on pySerial, to iRobot's Roomba and Create. It fitted our needs so we decided to use it. In order to be able to take advantage of as much of iRobot Create features as possible, we freely adapted *PyRobot* a bit, and added some new methods of our own -for example, a method to retrieve Create's sensor data-.

After some testing and experimenting with the *PyRobot* interface, we started developing applications for the Create robot. These were simple console applications at first, but we kept increasing their complexity as we obtained satisfactory results and felt more confident. For every application we developed we followed the same process: first, we decided what we wanted the robot to be able to do, that is, we described its desired behavior and wrote it down on a paper; then we modelled a statechart according to the behavior we had just described, and compiled it into Python; finally, we coded the main application and the support classes or methods that were needed. The Bluetooth connection worked like a charm and the robot's response was fast, so that gave us the idea to develop another application: the iRobot Create remote control application or *CreateRemote*.

We took a picture of the original Roomba robot remote control, edited it a bit in order to define our own buttons, and used it as the graphical user interface for the *CreateRemote* application. The main application file is implemented so that the name of the compiled statecharts model which is running in the background needs to be specified only once, and this makes it very easy to load different robot behaviors with the same GUI. We defined several statecharts modelled behaviors, from the simplest to the most complex, each of them combining as many Create's features as possible. The next and final step was to achieve complete autonomy.

It had originally been amongst this Master Thesis's objectives to turn our iRobot Create into a battle robot. In order to achieve this, our statecharts modelled behaviors needed to be designed in such a way that they could run autonomously, meaning that we would no longer be able to interact with the robot by sending the events required to trigger the transitions which caused it to change from one state to another. We dismissed the whole battle robot idea because of the great difficulty it entailed, but were determined to achieve autonomy nevertheless: we had to prove ourselves it was possible.

Our only means to generate events that the iRobot Create could react to was to use its sensors. The sensor data provided us with many tools to play with: we could make the robot react to both internal and external events. Internal events could be, for example, "when the traveled distance exceeds 50 meters, turn around 180 degrees and keep driving straight", or "when the battery is low, play an advice song". In turn, external events would depend on the environment as well, such as "whenever a wall is hit, turn 90 degrees to the left and keep driving straight", or "when a drop is detected -a staircase going down or a hole in the ground-, play an alert song".

Just a few days before our stay at McGill University was over, we finished working on our last autonomous behavior experiments, in which we had slightly exploited the possibilities that iRobot Create's sensors provide us with. So, even though we managed to define a few, really simple statecharts modelled autonomous behaviors that worked fine with the robot, we didn't have the time to delve deeper into that field. Still, we can say we achieved our purpose, and the path that opened up before us is full of possibilities. We've got many ideas for new projects that could derive from this, some of which we've put forward in the Future lines section of the present chapter. Finally, although we did not make a battle robot out of our Create, thus playing down the spectacularity of this Master Thesis, as well as making it a lot less interesting, we succeeded in setting a precedent at the Modelling, Simulation and Design Lab of McGill University, and we are pretty confident that iRobot Create Wars is not too far on the horizon.

6.1.2 Personal conclusions

The way I see it, becoming an engineer is but a matter of facing challenges and overcoming them. It's a very long and exhausting race, a constant, day-to-day struggle. It's a goal that can only be achieved through studying and hard work. Being already at the end of this race, the Master Thesis is just the final dash. In my opinion, the Master Thesis is an exam in which to prove that we have the qualities required in an engineer: capacity for work -and, especially, for team work-; capacity to learn and assimilate new technologies, programming languages, etc.; capacity to analyze a problem and find the best solution to it; in short, capacity to face challenges and overcome them. So, to me, the Master Thesis was not only a great opportunity to test my engineer skills in order to finally get my degree, but, first and foremost, it was a great opportunity to grow as a person in other aspects. So the following subsections are a compilation of all the challenges I've faced -and overcome- during the execution of my Master Thesis: a real life experience.

Life below zero

A project -any kind of project- is the result of many hours of dedication and hard work; it's the convergence of the achievement of many smaller goals, each of which is a necessary step to the final success. The work performed on a project is very much influenced by its environment and that of the people carrying it out. Thus, living under such conditions as extremely low temperatures and constant snowfall definitely affected the execution of this Master Thesis, and the proof of it is that the greatest part of the project was carried out during the Spring and Summer seasons. For some reason, there was a tendency of working less hours a day during the Wintertime, mainly because of the early nightfall. In short, the experience of surviving such cold weather should be considered amongst the many challenges that were faced and overcome during the carrying out of this Master Thesis.

Taking course COMP 763B at McGill University

Being a Visiting Student at the School of Computer Science of McGill University, my supervisor, Prof. Dr. Hans Vangheluwe, gave me the chance to attend one of the courses he teaches there during the Winter season, COMP 763B: Modelling and Simulation Based Design [8], which started just a week before I arrived in Montréal. The course covered modelling formalisms (from higraphs to Petri Nets, Statecharts, and DEVS), meta-modelling, and model transformation, all of them concepts which were unknown to me, and which I needed to learn for my Master Thesis. So I took the classes and did the course projects just like any other student, which was a great opportunity not only because of what I learnt, but also because it allowed me to meet many brilliant Computer Science colleagues, and I got to experience a different education method, that is, University lessons "the American way". Each Master course class has limited places; the availability ranges from 15 to 20 students approximately, and lessons are given in small classrooms. Therefore, the students are encouraged to take active part in the lessons, and interact with the teacher in a more personal and friendly way, providing the lessons with a relaxed atmosphere as well.

The lessons took place twice a week, two hours per day. The course schedule was tight, during the first month we learnt about modelling formalisms; we started with Petri Nets and were introduced to the traffic system example which, from that moment on, was the reference we kept coming back to through the whole course. Our first assignment was a series of small, related problems that asked to provide the building blocks for modelling traffic systems by means of Petri Nets. Next we studied statecharts, which introduced concepts such as orthogonality, composition, and history. Our second assignment was a lot longer and more difficult, but it also was so much fun to do: we were required to design a statecharts model specifying the reactive behaviour of a digital watch. We then moved on to DEVS (Discrete Event System Specification), though it was covered in less detail and in a more theoretical way. The corresponding assignment again revolved around traffic systems, more specifically we were asked to model the behaviour of car traffic on a straight stretch of road.

During the second part of the course we covered meta-modelling and model transformation. Meta-modelling is the modelling of modelling formalisms, that is, the building of meta-models. We learnt how to define the different elements of a modelling formalism and, more importantly, how to specify the rules and exceptions according to which these elements are interconnected. We were given a meta-modelling tutorial at a computer lab prior to facing the fourth assignment of the course which, once again, had to do with traffic systems; our job was to model a visual modelling environment for the traffic formalism. Our modelling formalism basically consisted of the following elements: roads, t-junctions, traffic lights, car generators, and car collectors, but the traffic system modelling possibilities are almost infinite.

Model transformation takes as input a model conforming to a given metamodel and produces as output another model conforming to a given metamodel, so it's very much like compilation, since the input is a piece of code written in a given language (i.e. C or C++) and the output is another piece of code with exactly the same meaning but in a different language (i.e. assembler). The model transformation is defined through a set of transformation rules that are applied to the original model in a specific order and for an undefined number of times. The transformation process loops through the rules and applies them if possible (each rule has a condition which must be fulfilled in order for the rule to be applied); when it's not possible to apply any of the rules any more, the model transformation process is over. The model transformation assignment consisted of defining the rules to transform our own traffic meta-model into Petri Nets, that is,

the process would take traffic system models -defined with our traffic meta-model- as inputs, and produce a Petri Nets model of the same traffic system as output.

The theoretical lessons were over by the end of February but the course was not over yet, the most important part of it was the development of a personal project, the subject of which was to be chosen by ourselves and had to be related to any of the concepts we had learnt during the course: modelling formalisms, meta-modelling, model transformation, etc. The personal project consisted of a reading presentation and a practical presentation; the first was a theoretical introduction to our project's subject, in which we talked about the reasons of our choice, we explained what was the goal of our project and how we planned to achieve it, and finally we gave a little sneak peek at our practical presentation. The reading presentations were held at class hours along two weeks, after which we were given a two weeks break to finish our projects and prepare the practical presentations. These were all held during the 30th of April, from 8 in the morning 'til around 8 in the evening, with one hour lunch break.

Obviously, since I was already working on that since I had arrived in Montréal, and it also was the goal of my stay at McGill, my personal project's subject was *Statecharts Modelling and Simulation of TankWars*, which represents the first part of this Master Thesis. Both my reading and practical presentations are included in this Master Thesis as *Appendix A* and *Appendix B* respectively.

Being a member of the *Modelling, Simulation and Design Lab*

Prior to joining the *Modelling, Simulation and Design Lab* (MSDL) at the School of Computer Science of McGill University, I barely knew what sort of projects were developed in it, or even what the words Modelling and Design referred to in a Computer Science context. I first met Prof. Dr. Hans Vangheluwe in Barcelona, in September 2007, and he introduced me to the lab and the projects they had been or were carrying out, the most important of which was -and still is- their software tool AToM^3. He also introduced me, quickly and briefly, to the statecharts formalism and the lab motto: *Model everything!*

During the Winter season (second semester at McGill University) I had the opportunity to attend the SOCS Colloquium [35], which is a weekly series organized by the School of Computer Science at McGill University. Twice a month, an invited speaker presents interesting research in some area of computer science. The colloquiums were always very interesting, providing valuable lessons on subjects that were totally unknown to me. There was another interesting thing about the colloquiums and it was the free drinks and snacks after the talk, which was a great opportunity to meet other international and/or local students and dive into a very enriching cultural exchange.

During the Spring and Summer seasons many new students joined the lab, some of them for an undetermined amount of time and others for just a few weeks or months, and then there were many projects going on at the same time. We would hold long lab meetings were each member would explain, with the help of a small presentation, what they were working on and what they had achieved so far. We would also discuss articles or papers about modelling formalisms. The information flow was very important in the lab, and thanks to it I had the privilege of meeting many brilliant people and learn a lot from them, especially about model transformation and modelling formalisms.

As a member of the MSDL I was also offered to present a poster at a very important event that was held during May 2008: the McGill Computer Science Alumni Open House. Thanks to being a poster presenter I was granted the permission to attend all the activities held during the

event. Many alumni of the School of Computer Science were invited to give speeches on their very successful professional lives after graduating from McGill University, and thus I had the privilege of meeting Alan Emtage [36], who conceived of and implemented the original version of the Archie search engine [37], the world's first Internet search engine and the start of a line which leads directly to today's Altavista, Yahoo!, and Google. Apart from this, the experience of presenting a poster for the first time ever was very positive and somewhat exciting. The subject chosen for the poster was, obviously, *Statecharts modelling of a robot's behavior*. The slides of the poster presentation are included in *Appendix C*.

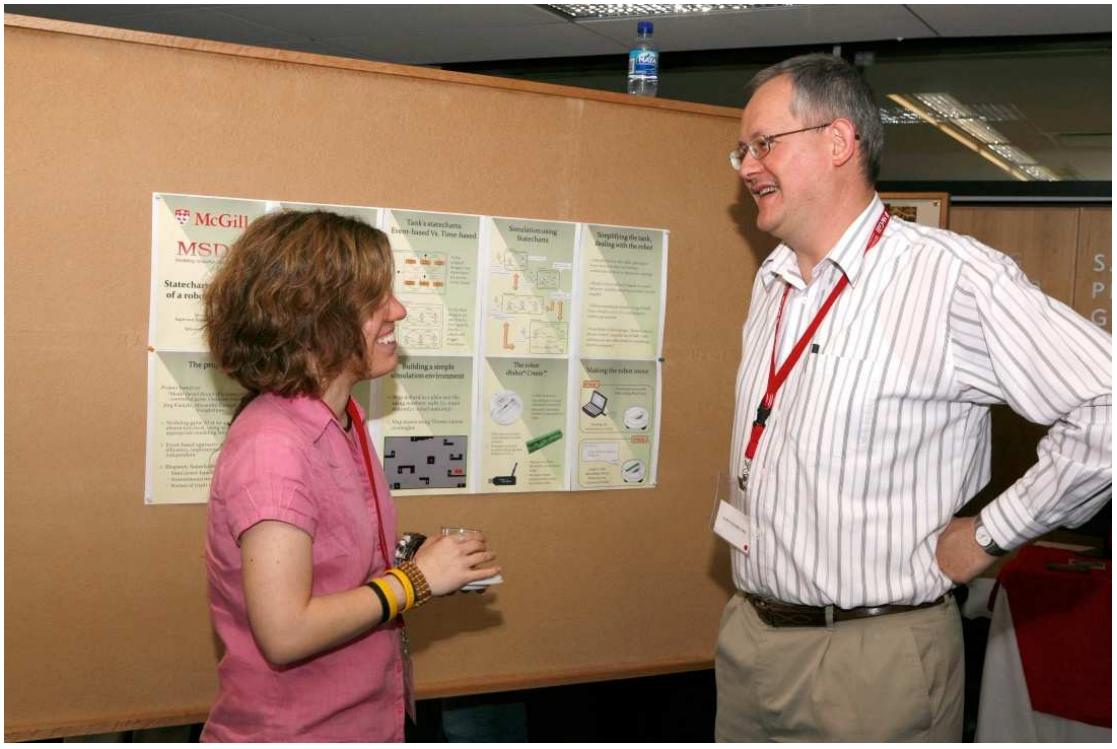


Figure 6.1: Prof. Dr. Hans Vangheluwe (right) and me, discussing statecharts in front of my poster during the first McGill Computer Science Alumni Open House.

Learning new things

The Master Thesis is the culmination of a very long and difficult learning process. However, the Computer Science Engineering is a never ending learning process. The engineer must keep studying for the rest of his or her working life, evolving as rapidly and constantly as the information technologies. The engineer must adapt him- or herself to the demands and necessities of the IT sector, and he/she must be prone to learning "new things". The Master Thesis is the final test in which the engineer must prove his/her learning and assimilation capacity, and it is also a framework in which to put into practice all the knowledge acquired in the course of the Engineering career.

In the carrying out of his/her Master Thesis, the engineer is expected to choose a subject that's slightly or completely unknown to him- or herself, investigate and learn about it on his/her own

and, finally, perform some work on it that could, eventually, lead to a discovery, an improvement, an evolution, etc. The job of an engineer is all about defeating challenges and, therefore, the opportunity of learning yet another programming language or how to use yet another software tool should be pleasantly welcomed by the engineer as intangible additions to the quality of his/her work, as well as other forms of personal enrichment.

This Master Thesis too is all about defeating challenges and learning new things:

- **Modelling and meta-modelling formalisms**. We read several articles on visual formalisms by David Harel; we have learnt about *higraphs*, *Petri Nets*, *Statecharts*, meta-modelling, model transformation, etc.
- **Programming languages**. We have learnt Python, in which we have discovered a fascinating programming language: easy to use, easy to learn, and multi paradigm. We have used some high-level Python interfaces such as *PySerial*, for defining and establishing serial port connections, and *pyRobot*, for communicating with the Create robot. We have also worked a little with Python's Tkinter, a graphical library. LaTeXcan also be considered a programming language itself, though the appropriate name for it is "mark-up language". Although the final report results are certainly impressive and it saves us a lot time in styling and formating the text, LaTeXhas proven to be a constant nightmare from the very first day we started using it for the writing of this Master Thesis. We even developed a simple Python application which generates ordered bibliography files, just to make our task easier. This application, called *LaTeX-BibitemStyler* [38] is published online in a Google Code open source project.
- **Software tools**. We have used *TeXnicCenter* [39] as our LaTeXeditor -which works pretty fine-, and AToM^3 for the modelling of all the statecharts appearing in this Master Thesis. Prof. Dr. Hans Vangheluwe himself, creator of AToM^3, taught us the basics of the application, and we learnt the rest on our own, through an extensive use of it. For writing Python code we simply used the traditional notepad application.
- **Hardware tools.** Working with iRobot Create has been my first approach to robotics, as a research field. Right out of the box the robot is useless -apart from having 10 built-in demos-, so the absolute control of Create's behavior is in the programmer's hands. Understanding the robot's open interface is not trivial, though we were fortunate to find a Python interface that dealt with the serial communication, making it a lot easier for us. The robot itself is very simple, but it's full of possibilities. Though it had originally been amongst this Master Thesis objectives to work with "the green bar" (Create's Command Module), thus making the robot completely autonomous, the time was too short and we had to discard it of the scope of our project. However, after studying "the green bar" a bit and going through the code of some of its program examples, we're confident it would be possible to adapt an AToM^3's StateCharts Compiler (SCC) generated code to the specific syntax requirements of "the green bar".

6.2 Future lines

6.2.1 iRobot Create wars

First and foremost, and since it had originally been amongst this Master Thesis main objectives, an obvious future line would be to turn our iRobot Create robot into a battle robot. This project, however, entails little software development and a lot of electronics and hardware assembly instead, which was also one of the reasons why the project was ruled out in the first place. Still, when we began working with the Create robot we experimented with its Command module. Once the hardware components would have been assembled on the robot's "green bar", the programming of it would have been the least of our headaches. Prof. Dr. Hans Vangheluwe and me discussed many times about how we would approach the implementation of the battle robot. Although my stay didn't provide us with time enough to dare stepping into that project, it was actually intended that another student of the Modelling, Simulation and Design Lab would take over from me and continue the work where I left it. Hence, the iRobot Create wars is still one of the projects currently under development at the MSDL of McGill University. The most important issues to resolve when considering the hardware implementation of the battle robot are:

(a) how could our robot actually "hit" the enemy, without physically damaging it -although Create robots are very sturdy, we don't want to break them!-, and

(b) how would our robot detect that it has been hit by enemy "fire".

We discarded the "sumo-fighting robot" approach because, by reading the data from the Create's bumper sensors, it's impossible to distinguish a collision against an enemy from a crash against an obstacle. As well, it would be very difficult to push a Create robot out of the ring with another Create robot, due to their perfectly round shape.

Then we thought about using laser beams. They could be mounted on the iRobot Create Command Module, and the robot could be covered with photo-sensitive devices, such as light sensors, all over the perimeter of its casing. But then, the laser is a continuous light beam and, even if we could turn it on and off by means of some kind of trigger, its range would remain unlimited, which in turn would make the robot battles too easy.

Although it didn't seem like a good idea at the beginning, we then considered using some kind of projectiles, small and soft enough in order not to damage the robots nor harm the human beings supervising the battles either. What about mounting tiny plastic ball guns on the Create's "green bar"? The Command Module is full of possibilities! The only condition we'd impose is to keep the Create robot's behavior completely autonomous and statecharts modelled.

6.2.2 World discovery

The concept of this project is inspired by the Tank Wars video game. We've called it *World discovery* but could as well have called it *Room mapping*, this is the idea: the Create robot is placed somewhere in a room -it doesn't matter if it's the middle or the entrance-, and it's left there to roam on its own; we don't control it remotely, it moves around freely, autonomously. As the robot keeps wandering around the room it will, at some point, hit a wall or an obstacle, be it a chair, a table, a desk, whatever. Upon running into an obstacle, Create gets new information about the environment surrounding it. We can know at all times where in the room Create is, its origin coordinates are used as a reference along with the sensors data -which tell us the traveled distance, speed, angle, etc.-, to calculate the robot's current coordinates. As the robot moves,

it maps out the world that surrounds it, discovering and drawing the obstacles by running into them. So, when Create starts roaming an unknown, unexplored environment, its map of the world will be a blank page. But, after some time, the map will be a very approximate representation of the room's floor, which will keep becoming more and more detailed as the robot wanders around.

Although this project is based on the Exploring component of the Tank Wars video game, there are many real world applications to it. We might be getting too imaginative but, nevertheless, iRobot Create robots could be used to map out the world in an infinity of situations, such as: police raids, rescue operations, prospecting or exploration of planet surfaces, spelunking, etc.. All these suggestions may seem a bit crazy but we are convinced that they are feasible; iRobot Create is a very sturdy and compact robot, and we can attach multiple peripherals/hardware components to its Command module: cameras, infrared sensors, heat sensors, night vision sensors, robotic arms, stun guns... practically anything we can imagine!

Concerning this future line proposal, it's worth mentioning that iRobot Corporation has actually been developing military robots for the US Army and the Pentagon for several years now [40], some of which are -as captured in Figure 6.2.2-:

(a) *PackBot*. This is the first and most popular military robot by iRobot Corporation. The *PackBot* has been successfully used in recent conflicts in both Afghanistan and Iraq, and the US Army currently possesses many *PackBot* units [41]. It's designed to perform a wide variety of tasks: unknown territory reconnaissance, route clearance, explosives detection, explosive ordnance disposal (EOD), hazardous material detection, force protection, sniper detection, etc..

(b) *Warrior 700*. *PackBot*'s big brother, the *Warrior*, had already been under development for two years when iRobot Corporation won a $3.75M army contract to build and deliver two units in October 2008 [42]. With a longer chassis and a much greater payload capacity, the Warrior is capable of explosives disposal, firefighting, clearing buildings, and even extracting casualties from the battlefield.

(c) *Negotiator*. *"See, hear and evaluate dangerous situations from a safe distance... Safely examine suspicious vehicles, packages and buildings... Defeat a variety of SWAT threats..."*, this is how iRobot Corporation advertises the *Negotiator* in their web site. Its low cost makes it ideal for civil response operations such as -in addition to the ones already mentioned- hazardous material detection, surveillance, etc. Lightweight and compact, the *Negotiator* fits into the trunk of any vehicle, easily climbs stairs, and moves swiftly through a range of environments.

6.2.3 Song composition tool

One of the iRobot Create features that I particularly enjoy the most despite its apparent uselessness, is the possibility of composing and playing MIDI songs. Partly, this is due to the fact that, after the first practical demonstrations of my progresses with the Create robot before my colleagues at the School of Computer Science of McGill University, I noticed that they were more impressed by the robot's ability to whistle "The cantina song" than by it chasing students at full speed down the corridors of the second floor of McConnell Engineering building. This gave me the idea of developing an iRobot Create song composition tool for use and enjoyment of Create robot owners and enthusiasts around the world. This idea was highly encouraged by the fact that composing songs for the Create robot is actually really complicated, since it

(a) PackBot. (b) Warrior 700. (c) Negotiator.

Figure 6.2: Some of iRobot Corporation's military robots.

must be done "by ear" and, therefore, a little knowledge of music theory is almost mandatory. All of the songs that I've used in my practical experiments have been taken from the iRobot Create online forums; I decided to compose some songs myself: I read about music theory and MIDI song composition on the internet, and I tried several MIDI composition software applications, but all my attempts were quite disastrous. For all these reasons, I believe that the song composition tool would be very appreciated by iRobot Create owners, and maybe even by the iRobot Corporation itself too.

A

Tank Wars Reading presentation for course COMP 763B's personal project

STATECHARTS MODELLING OF TANK WARS

[*Reading Presentation*]

School of Computer Science
McGill University, Montréal, Canada

COMP 763 / Silvia Mur Blanch

Overview

- Introduction
- Context and goals
- Modelling the "game characters"
- Mapping to an execution platform
- Summary

Acknowledgement

- Entirely based on paper :

"Model-based Design of Computer-controlled Game Character Behaviour"

by

Jörg Kienzle, Alexandre Denault & Hans Vangheluwe

and their homonym presentation given at the

10th MoDELS Conference

5 October 2007

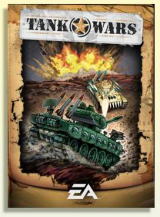
Nashville, TN

Overview

- Introduction
- Context and goals
- Modelling the "game characters"
- Mapping to an execution platform
- Summary

About EA Tank Wars competition

- Develop an AI for a tank that lives in a 2D world
- Simulation environment written in C++
- Manage:
  - Resources of the tank (health, fuel)
  - Map out the world
  - Find the enemy tank
  - Destroy the enemy tank

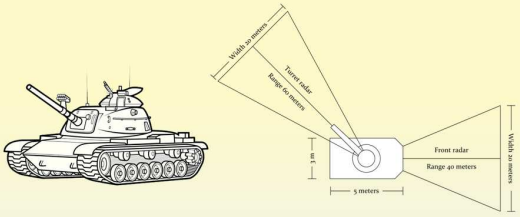


Computer games nowadays

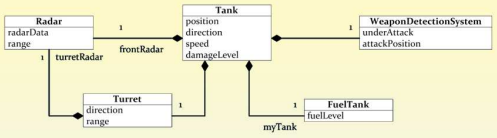
- Computer games sales have experienced a huge growth in the past few years
- Demand for higher realism in computer games leads to development of better AIs
- Necessity for reusability
- AI specification should be done at a higher level of abstraction: not coded, but modeled!
- Models define reactions to game events, so event-based formalisms seem to be the most appropriate

Overview

Abstraction



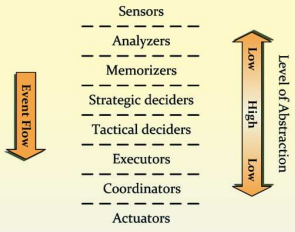
Abstraction: Tank structure (state)



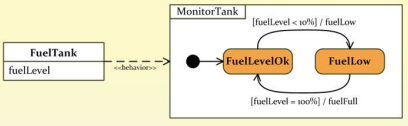
Optimal formalism(s) to model detailed behavior?

- Time-sliced Vs. Event-based approach (more abstract)
- Individual objects' behavior:
  - State/event based
  - Autonomous/reactive behavior
  - Notion of time
  - Modularity
  - Well known
  - Availability of simulators/code generators

Abstraction: Tank behavior (high-level)

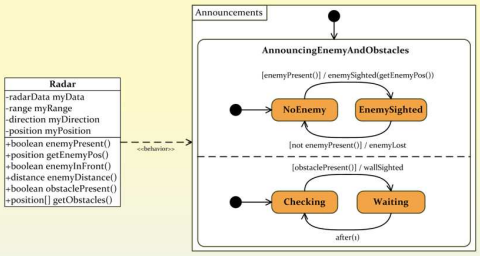


Sensors – Generating important game events



Modelling the "game characters"

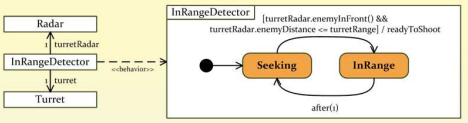
Sensors – Generating important game events



Statecharts modelling of Tank Wars

Modelling the "game characters"

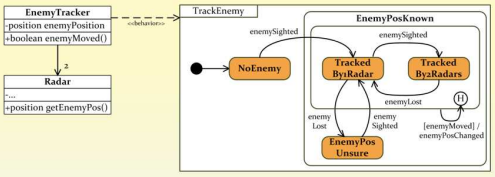
Analyzers – Correlating sensor events



Statecharts modelling of Tank Wars

Modelling the "game characters"

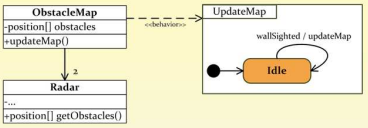
Memorizers – Modelling memory



Statecharts modelling of Tank Wars

Modelling the "game characters"

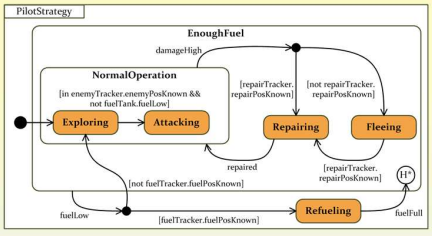
Memorizers – Modelling memory



Statecharts modelling of Tank Wars

Modelling the "game characters"

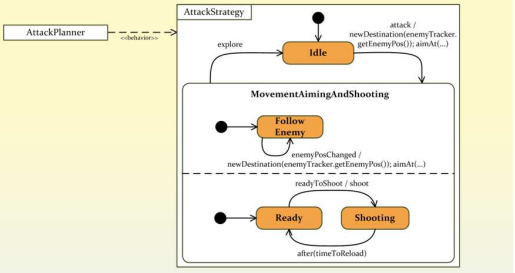
Strategic deciders – High-level goals



Statecharts modelling of Tank Wars

Modelling the "game characters"

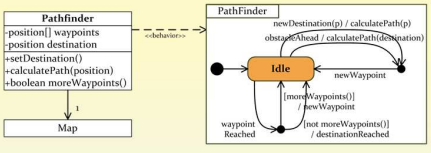
Tactical deciders – Planning how to achieve goal



Statecharts modelling of Tank Wars

Modelling the "game characters"

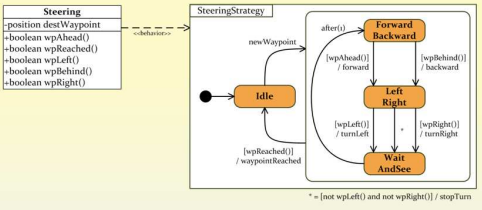
Tactical deciders – Planning how to achieve goal



Statecharts modelling of Tank Wars

Modelling the "game characters"

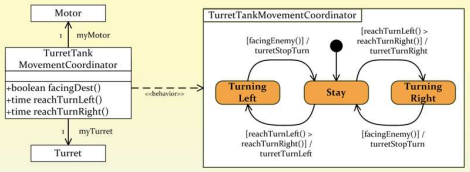
Executors – Mapping decisions to actuator commands



Statecharts modelling of Tank Wars

Modelling the "game characters"

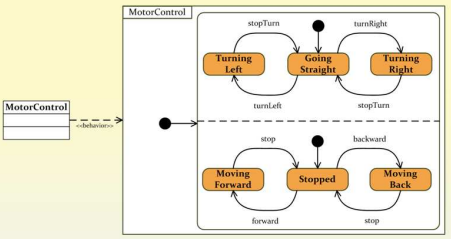
Coordinators – Resolving actuator interactions



Statecharts modelling of Tank Wars

Modelling the "game characters"

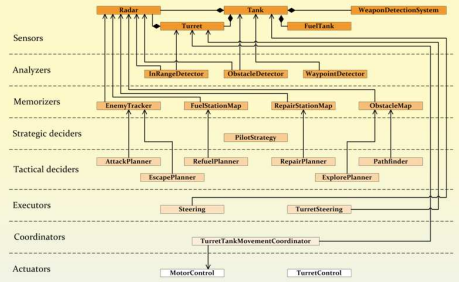
Actuators – Influencing the game state



Statecharts modelling of Tank Wars

Modelling the "game characters"

Putting it all together: (a)synchronous communication



Statecharts modelling of Tank Wars

Overview

- Introduction
- Context and goals
- Modelling the "game characters"
- Mapping to an execution platform
- Summary

Statecharts modelling of Tank Wars

Mapping to an execution platform

Time-slicing Vs. Continuous time

- In Tank Wars simulation, new state of the environment is sent to the AI component every 50ms
- Continuous time is most general and most appropriate at this level of abstraction because:
  - Modelling freedom: modeller not unnecessarily constrained by implementation issues, can focus on the logic of the model
  - Symbolic analysis: analyze model using timed logic
  - Simulation accuracy: due to real numbers

Statecharts modelling of Tank Wars

Overview

- Introduction
- Context and goals
- Modelling the "game characters"
- Mapping to an execution platform
- Summary

Statecharts modelling of Tank Wars

Summary

Summary

- Appropriate level of abstraction:
  - Events
  - Layers
  - Data: filtering
- Appropriate formalism(s):
  - State/event-based
  - Autonomous/reactive behavior
  - Notion of time
  - Modularity
  - Well known
  - Availability of simulators/code generators

Statecharts modelling of Tank Wars

Statecharts modelling of Tank Wars

B

Tank Wars Practical presentation for course COMP 763B's personal project

STATECHARTS MODELLING OF TANK WARS

[Project Presentation]

School of Computer Science
McGill University, Montréal, Canada

COMP 763 / Silvia Mur Blanch

Overview

- Recap
- The static part: the controller
- Designing the tank components' statecharts
- Recreating the simulation environment
- Summary

Recap

- EA TankWars: AI programming contest, C++ environment
- AI specification should be done at a higher lever of abstraction: modelled instead of coded (reusability)
- Since models define reactions to game events, event-based formalisms seem to be the most appropriate
- Formalisms:
  - State / event based
  - Autonomous / reactive behaviour
  - Notion of time
  - Modularity
  - Availability of simulators / code generators
- Time-sliced Vs. Event-based

Overview

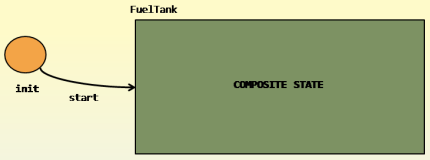
- Recap
- The static part: the controller
- Designing the tank components' statecharts
- Recreating the simulation environment
- Summary

The static part: the controller

- One big statechart (digital watch) Vs. many small statecharts
- Defining each component in different statecharts means event broadcasting is not possible
- Necessity to use controller: it will "contain" all the tank's components and link them

The static part: the controller

- Defining each component of the tank with a statechart, e.g.:



The static part: the controller

- Event "start" to begin the simulation of the statecharts, (pass controller as parameter) e.g.:

```
print "fuelTank start"
ctl=[PARAMS]
```

- Defining an instance of each component / statechart of the tank, e.g.:

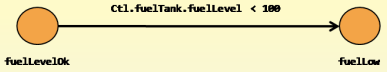
```
self.fuelTank = FuelTank.FuelTank()
self.fuelTank.initModel()
self.fuelTank.event("start", self)
```

- Adding variables to the tank's components, e.g.:

```
self.fuelTank.fuelLevel = MAX_TANK_FUEL
```


The static part: the controller

- Tank's components generate events in other components (statecharts) by using "*narrow cast*", e.g.:
  - FuelTank



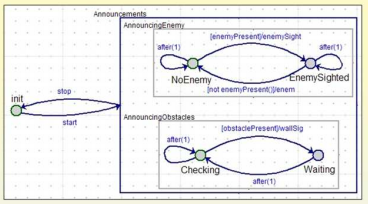
  - Action:

```
ctl.fuelTank.fuelLow = True
ctl.pilotStrategy.event("fuelLow")
```


Overview

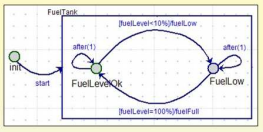
Designing the tank components' statecharts

- Announcements (sensors)



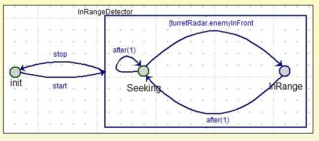
Designing the tank components' statecharts

- FuelTank (sensors)



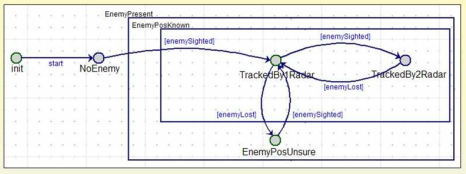
Designing the tank components' statecharts

- InRangeDetector (analyzers)



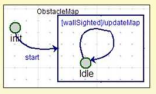
Designing the tank components' statecharts

- EnemyTracker (memorizers)



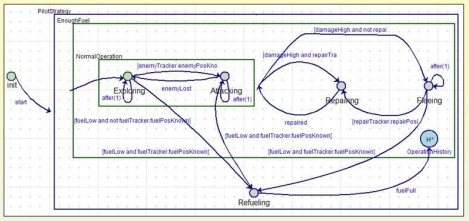
Designing the tank components' statecharts

- ObstacleMap (memorizers)



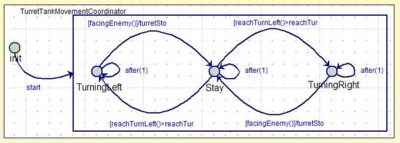
Designing the tank components' statecharts

- PilotStrategy (strategic deciders)



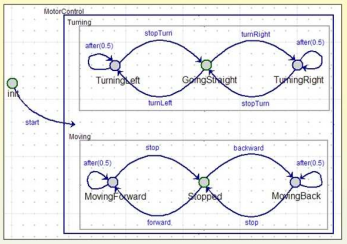
Designing the tank components' statecharts

- TurretTankMovementCoordinator (coordinators)



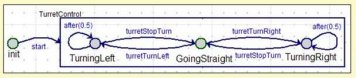
Designing the tank components' statecharts

- MotorControl (actuator)



Designing the tank components' statecharts

- TurretControl (actuator)



Overview

- Recap
- The static part: the controller
- Designing the tank components' statecharts
- Recreating the simulation environment
- Summary

Recreating the simulation environment

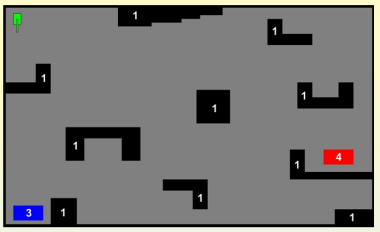
- Time-sliced to event-based: a lot of work!
- Implement a very simple simulation environment instead of using EA Tank Wars environment
- Use Python instead of C++
- Translate EA Tank Wars header files into Python
  - **ai**
  - **coord**
  - **visor**

Recreating the simulation environment

- Using Tkinter to draw world map and tank

Recreating the simulation environment

- World map defined using numbers in a plain text file



Recreating the simulation environment

- Moving the tank (translation)

```
def translatePoint(point, dx, dy):
      x = point.x + dx
      y = point.y + dy
      return Coord2D(x, y)
```

- Turning the tank and the turret (rotation)

```
def rotatePoint(point, origin, angle):
      x = origin.x + ((point.x - origin.x) *
            cos(angle) - (point.y - origin.y) * sin(angle))
      y = origin.y + ((point.x - origin.x) *
            sin(angle) + (point.y - origin.y) * cos(angle))
      return Coord2D(x, y)
```

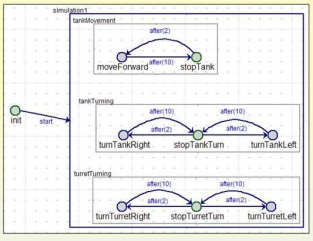

Recreating the simulation environment

- The simulation is also described with a statechart

```
root = Tk()
root.withdraw()
window = newWindow(root)

fr = FileReader.FileReader("input2.txt")
fr.loadFile()

dr = Drawer.Drawer(root, window, fr.lines, 40, 40)
ctl = TankController.TankController(dr)

sim = simulation.simulation()
sim.initModel()
sim.event("start", ctl)

root.mainloop()
```


Recreating the simulation environment

- Simulation example:



Overview

- Recap
- The static part: the controller
- Designing the tank components' statecharts
- Recreating the simulation environment
- Summary

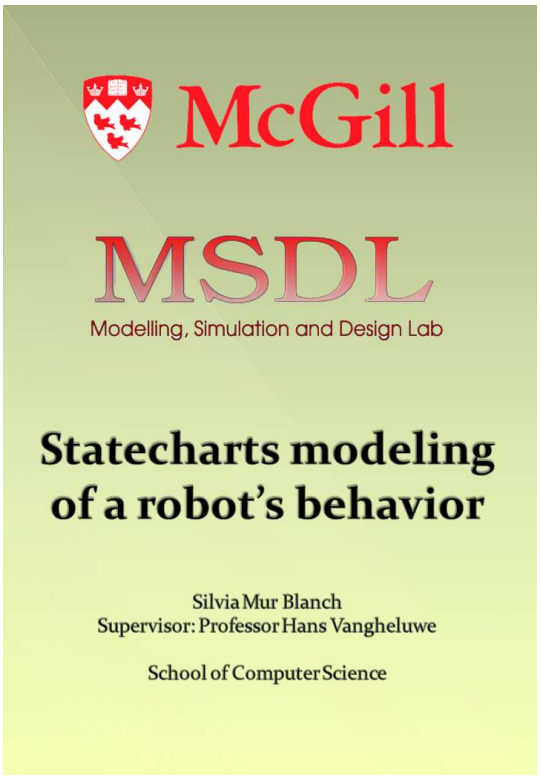
Summary

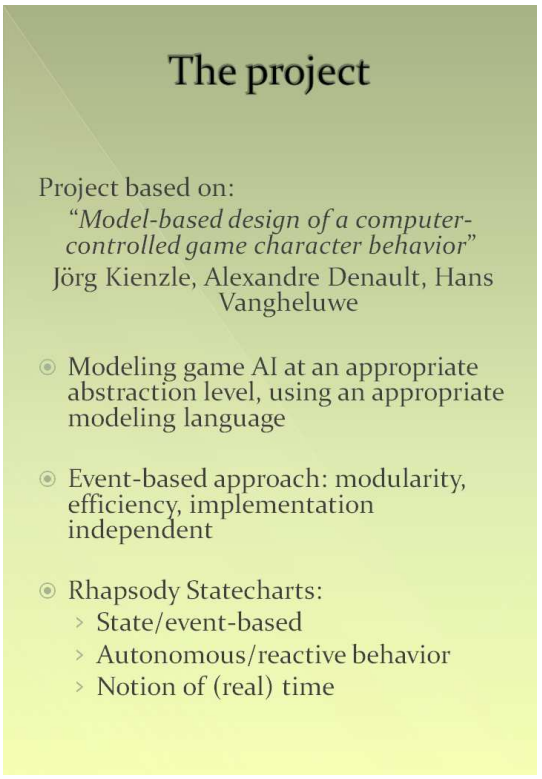
- Succeeded:
  - Model tank's AI at a higher level of abstraction by using event-based formalism: statecharts
  - Necessity to use controller class to link tank's components and allow communication between them: "*narrow cast*"
  - Implementing a very simple environment for simulation instead of using EA Tank Wars
  - Describing simulation with a statechart too
- Not succeeded (yet):
  - Perceiving environment through the sensors (feeding the tank with input: obstacles, enemies, etc.)

Thank you!

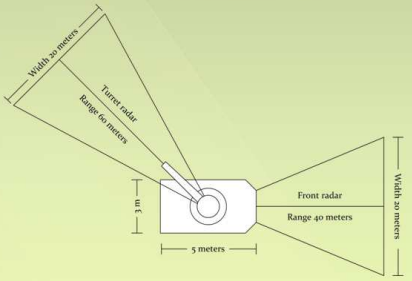
C

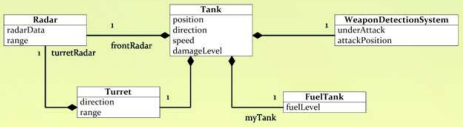
Poster presented at the McGill Computer Science Alumni Open House



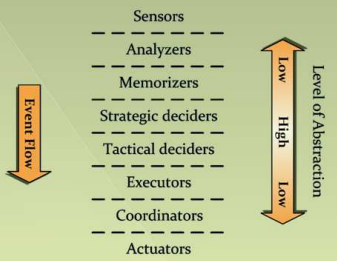


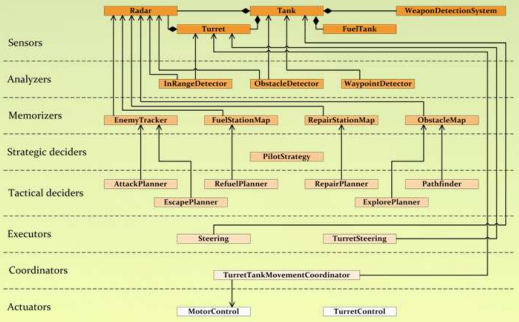
Abstracting the Tank



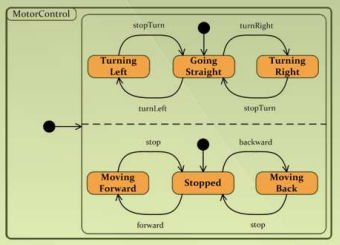


Tank's AI architecture

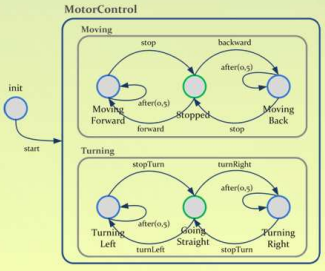




Tank's statecharts: Event-based Vs. Time-based



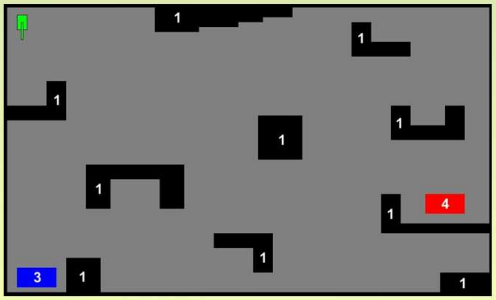
In the original designs, the statecharts are purely event-based



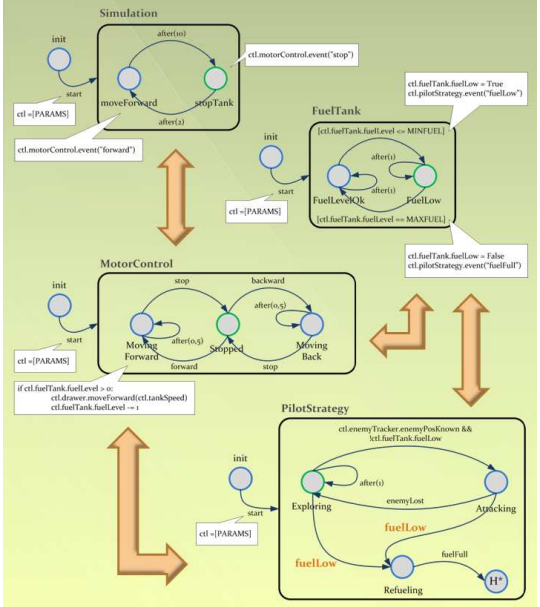
In the final designs, we use time to force guards checks, which will trigger transitions

Building a simple simulation environment

- Map defined in a plain text file using numbers: walls (1), repair station(2), refuel station(3)
- Map drawn using Tkinter canvas rectangles



Simulation using Statecharts



The robot: *iRobot® Create™*

- 32 built-in sensors
- Use serial port to send individual commands
- Store up to 100 Open Interface Commands on robot

- 8-bit microcontroller with 16Kbytes FLASH memory
- Provides on-board control with programs written in C/C++

- High power Class 1 Bluetooth, 300ft (100m) range
- Provides remote communication with the Create robot

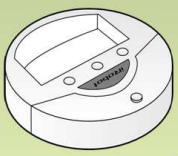
Simplifying the tank, dealing with the robot

- Introduction to the robot: getting to know how it works and its basic commands (10 built-in demos for testing)
- Model robot's behavior based on tank's behavior: simplify statecharts (robot is a lot simpler)
- Define statecharts models using AToM³, then compile into C/C++ and adapt to robot's parameters
- Possibility of developing a "*Statecharts to iRobot Create*" compiler for AToM³ : free and easy-to-use robot behavior modeling tool for everyone!

Making the robot move

1st step

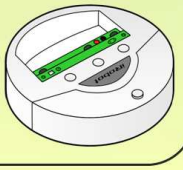
Communication with robot using BlueTooth



Sending *one instruction at a time*

2nd step

```
int main()
{
   printf ("Hello world!");
   return 0;
}
```



Load C code describing robot's behavior into *Command module*

D

Personal contribution to *PyRobot* interface

```
#######################################################
####
#### Silvia Mur Blanch
#### silvia.murblanch@gmail.com
#### MSDL, School of Computer Science
#### McGill University, Montreal, Canada
####
#######################################################

CHARGING_SOURCES = {
   0: 'None',
   1: 'Internal charger present',
   2: 'Home base present',
   3: 'Internal charger and Home base present',
}

CREATE_BUTTONS = {
   0: 'None',
   1: 'Play button pressed',
   4: 'Advance button pressed',
   5: 'Play and Advance buttons pressed',
}

SENSOR_PACKET_GROUP_IDS = (0, 1, 2, 3, 4, 5, 6)

SENSOR_PACKET_GROUPS = (
   (7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26),
   (7, 8, 9, 10, 11, 12, 13, 14, 15, 16),
   (17, 18, 19, 20),
   (21, 22, 23, 24, 25, 26),
   (27, 28, 29, 30, 31, 32, 33, 34),
   (35, 36, 37, 38, 39, 40, 41, 42),
   (7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
         27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42),
)
```

```
SENSOR_PACKETS = {
   7: 'Bumps and wheel drops',
   8: 'Wall',
   9: 'Cliff left',
   10:'Cliff front left',
   11:'Cliff front right',
   12:'Cliff right',
   13:'Virtual wall',
   14:'Low side driver and wheel overcurrents',
   15:'Unused bytes',
   16:'Unused bytes',
   17:'Infrared byte',
   18:'Buttons',
   19:'Distance',
   20:'Angle',
   21:'Charging state',
   22:'Voltage',
   23:'Current',
   24:'Battery temperature',
   25:'Battery charge',
   26:'Battery capacity',
   27:'Wall signal',
   28:'Cliff left signal',
   29:'Cliff front left signal',
   30:'Cliff front right signal',
   31:'Cliff right signal',
   32:'Cargo bay digital inputs',
   33:'Cargo bay analog signal',
   34:'Charging sources available',
   35:'OI mode',
   36:'Song number',
   37:'Song playing',
   38:'Number of stream packets',
   39:'Requested velocity',
   40:'Requested radius',
   41:'Requested right velocity',
   42:'Requested left velocity',
}

SPECIAL_SENSOR_PACKETS = (
   'Bumps and wheel drops',
   'Low side driver and wheel overcurrents'
)

BYTE_1_BOOLEAN = (8, 9, 10, 11, 12, 13, 37)

BYTE_1_UNSIGNED = (7, 14, 17, 18, 21, 32, 34, 35, 36, 37, 38)
```

```
BYTE_1_SIGNED = (15, 16, 24)

BYTE_2_UNSIGNED = (22, 25, 26, 27, 28, 29, 30, 31, 33)

BYTE_2_SIGNED = (19, 20, 23, 39, 40, 41, 42)

#####################################################


class Create(Roomba):
      ...
      def LoadSong(self, chords):
          """Load a song on the Create.

          """
          self.sci.song(*chords)

      def PlaySong(self, song_id):
          """Play a song loaded on the Create.

          """
          self.sci.play(*song_id)

      def GetPacketSize(self, packet_id):
          """Retrieving sensor packet size.

          """
          if packet_id in SENSOR_PACKET_GROUP_IDS:
             return SENSOR_GROUP_PACKET_LENGTHS[packet_id]
          elif packet_id in BYTE_1_BOOLEAN or packet_id in BYTE_1_UNSIGNED or
             packet_id in BYTE_1_SIGNED:
             return 1
          elif packet_id in BYTE_2_UNSIGNED or packet_id in BYTE_2_UNSIGNED:
             return 2

      def _FurtherDecodePacket(self, packet_id):
          """Some packet id's may be further decoded.

          """
          if SENSOR_PACKETS[packet_id] == 'Infrared byte':
             self.sensors.data['Infrared byte']
             = REMOTE_OPCODES[self.sensors.data['Infrared byte']]
          if SENSOR_PACKETS[packet_id] == 'Charging state':
             self.sensors.data['Charging state']
             = CHARGING_STATES[self.sensors.data['Charging state']]
          if SENSOR_PACKETS[packet_id] == 'OI mode':
```

```
            self.sensors.data['OI mode'] = OI_MODES[self.sensors.data['OI mode']]
        if SENSOR_PACKETS[packet_id] == 'Charging sources available':
            self.sensors.data['Charging sources available']
            = CHARGING_SOURCES[self.sensors.data['Charging sources available']]
        if SENSOR_PACKETS[packet_id] == 'Buttons':
            self.sensors.data['Buttons']
            = CREATE_BUTTONS[self.sensors.data['Buttons']]

    def SpecialSensorPackets(self, name, bytes):
        """Check for special sensor packets and treat them.

        """
        if name == 'Low side driver and wheel overcurrents':
            self.sensors.MotorOvercurrents(bytes.pop())
            return True
        if name == 'Bumps and wheel drops':
            self.sensors.BumpsWheeldrops(bytes.pop())
            return True
        if name == 'Angle':
            self.sensors.Angle(high=bytes.pop(), low=bytes.pop(), unit='degrees')
            return True
        return False

    def DecodePacket(self, packet_id, bytes):
        """Decode a sensor packet.

        """
        name = SENSOR_PACKETS[packet_id]
        if packet_id in BYTE_1_BOOLEAN:
            self.sensors.DecodeBool(name, bytes.pop())
        elif packet_id in BYTE_1_UNSIGNED:
            if not self.SpecialSensorPackets(name, bytes):
                self.sensors.DecodeUnsignedByte(name, bytes.pop())
        elif packet_id in BYTE_1_SIGNED:
            if not self.SpecialSensorPackets(name, bytes):
                self.sensors.DecodeByte(name, bytes.pop())
        elif packet_id in BYTE_2_UNSIGNED:
            if not self.SpecialSensorPackets(name, bytes):
                self.sensors.DecodeUnsignedShort(name, high=bytes.pop(), low=bytes.pop())
        elif packet_id in BYTE_2_SIGNED:
            if not self.SpecialSensorPackets(name, bytes):
                self.sensors.DecodeShort(name, high=bytes.pop(), low=bytes.pop())
        self._FurtherDecodePacket(packet_id)

    def DecodePacketGroup(self, packet_id, bytes):
        """Decode a sensor packet or a sensor packet group (packet id's 0-6).
```

```
        """
        ## Reverse the list because it's easier to simply pop the elements
        ## from the end.
        bytes.reverse()
        if packet_id in SENSOR_PACKET_GROUP_IDS:
           packets = SENSOR_PACKET_GROUPS[packet_id]
           for packet_id in packets:
              self.DecodePacket(packet_id, bytes)
        else:
           self.DecodePacket(packet_id, bytes)

    def RequestSensorData(self, packet_id):
        """Request a sensor packet from the robot.

        """
        self.sci.sensors(*packet_id)
        num_bytes = self.GetPacketSize(packet_id[0])
        ## Wait 1 second after reading serial port.
        print '...requesting sensor packet', packet_id[0]
        time.sleep(0.1)
        """Read response from the robot."""
        bytes = list(self.sci.ser.read(num_bytes))
        if len(bytes) != num_bytes:
           print "\n"
           print "Uh oh! Wrong data length!"
           print "\n"
        else:
           self.DecodePacketGroup(packet_id[0], bytes)

    def GetSensorData(self, packet_id):
        """Return a sensor packet data (must have been previously requested).

        """
        if packet_id[0] > 6:
        return self.sensors.data[SENSOR_PACKETS[packet_id[0]]]

    def RequestAndGetSensorData(self, packet_id):
        """Request a sensor packet and return the obtained data.

        """
        self.RequestSensorData(packet_id)
        return self.GetSensorData(packet_id)

    def PrintSensorData(self):
        """Print sensor data obtained from the robot.

        """
```

```
            system("cls")
            print "\n"
            print "-------------------------"
            for key,value in sorted(self.sensors.data.iteritems()):
               if key in SPECIAL_SENSOR_PACKETS:
                  ## Special sensor packets contain detailed data so their keys have
                  ## dictionaries as values.
                  print "|+",key,":"
                  for skey,svalue in sorted(value.iteritems()):
                  print " |+",skey,"=",svalue
               else:
                  print "|+",key,"=",value
                  print "-------------------------"
                  print "\n"

      def RequestAndPrintSensorData(self, packet_id):
         """Request a sensor packet from the robot and print the obtained data

         """
         self.RequestSensorData(packet_id)
         self.PrintSensorData()

####################################################
```

Bibliography

[1] HANS VANGHELUWE. *The Modelling, Simulation and Design Lab (MSDL).* [web page], 15 September 2008. McGill University. [Visited: 14 January 2008]. URL: `http://msdl.cs.mcgill.ca/`.

[2] Kienzle, J., Denault, A., Vangheluwe, H., "Model-Based Design of Computer-Controlled Game Character Behavior", *MoDELS*, pp.650-665, 2007.

[3] ELECTRONIC ARTS. *EA Tank Wars.* [web page], 4 November 2005. Electronic Arts. [Visited: 14 January 2008]. URL: `http://www.info.ea.com/company/company-tw.php`.

[4] HANS VANGHELUWE. *AToM^3 A Tool for Multi-formalism Meta-Modelling.* [web page], 17 January 2008. McGill University. [Visited: 14 January 2008]. URL: `http://atom3.cs.mcgill.ca/`.

[5] THOMAS FENG. *SCC (StateChart Compiler) 0.1 Documentation.* [web page], 11 November 2003. McGill University. [Visited: 23 January 2008]. URL: `http://msdl.cs.mcgill.ca/people/tfeng/uml/scc/doc/index.html`

[6] IROBOT CORPORATION. *iRobot: iRobot Create® Programmable Robot.* [web page], December 2008. iRobot Corporation. [Visited: 14 January 2008]. URL: `http://store.irobot.com/product/index.jsp?productId=2586252`.

[7] BETTINA KEMME. *gr@m - Games Research at McGill.* [web page], 31 January 2009. McGill University. [Visited: 18 February 2007]. URL: `http://gram.cs.mcgill.ca/`.

[8] HANS VANGHELUWE. *Modelling and Simulation Based Design (COMP 763B).* [web page], 9 September 2008. McGill University. [Visited: 14 January 2008]. URL: `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/`.

[9] TIHAMER LEVENDOVSZKY. *Visual Modelling and Transformation System - VMTS.* [web page], 18 June 2008. Budapest University of Technology and Economics - Department of Automation and Applied Informatics. [Visited: 18 December 2008]. URL: `http://avalon.aut.bme.hu/~tihamer/research/vmts/`.

[10] IBM CORPORATION. *Telelogic is now IBM - Telelogic Statemate: Embedded Systems Design Software.* [web page], January 2009. IBM Corporation. [Visited: 18 December 2008]. URL: `http://modeling.telelogic.com/products/statemate/index.cfm`.

[11] Harel, D., Naamad, A., "The STATEMATE semantics of statecharts", *ACM Trans. Softw. Eng. Methodol.*, Vol.5, Issue 4, pp.293-333, 1996.

[12] IBM CORPORATION. *Telelogic is now IBM - Telelogic Rhapsody: Model-Driven Development Software with UML 2.0* [web page], January 2009. IBM Corporation. [Visited: 18 December 2008]. URL: `http://modeling.telelogic.com/products/rhapsody/index.cfm`.

[13] Harel, D., Kugler, H., "The Rhapsody Semantics of Statecharts (or, On The Executable Core of the UML)", *In Integration of Software Specification Techniques for Application in Engineering, number 3147 in Lecture Notes in Computer Science*, pp.325-354, 2004.

[14] PARALLAX, INC. *Boe-Bot Robot Information.* [web page], December 2008. Parallax, Inc. [Visited: 3 June 2008]. URL: `http://www.parallax.com/tabid/411/Default.aspx`.

[15] GREGORY DUDEK. *Mobile Robotics Lab @ McGill.* [web page], September 2008 . McGill University. [Visited: 10 December 2008]. URL: `http://www.cim.mcgill.ca/~mrl/`.

[16] PARALLAX, INC. *SumoBot Robot Information.* [web page], December 2008. Parallax, Inc. [Visited: 3 June 2008]. URL: `http://www.parallax.com/tabid/412/Default.aspx`.

[17] IBOT LIMITED. *About picoBotz.* [web page]. iBOTZ Limited. [Visited: 3 June 2008]. URL: `http://www.ibotz.com/html/AboutPicoBotz.html`.

[18] DAMON KOHLER. *PyRobot: A Python interface to iRobot's Roomba and Create.* [web page], December 2007. Google Code. [Visited: 20 May 2008]. URL: `http://code.google.com/p/pyrobot/`.

[19] VARIOUS AUTHORS. *Modeling language.* [web page], 30 March 2009. Wikipedia, the free encyclopedia. [Visited: 5 May 2009]. URL: `http://en.wikipedia.org/wiki/Modeling_language`.

[20] Harel, D., "On visual formalisms", *Commun. ACM*, Vol.31, Issue 5, pp.514-530, May 1988.

[21] Harel, D., "Statecharts: A visual formalism for complex systems", *Sci. Comput. Program.*, Vol.8, Issue 3, pp.231-274, June 1987.

[22] THOMAS FENG. *Home - Thomas Huining Feng.* [web page], 7 May 2009. U.C. Berkeley. [Visited: 20 January 2008]. URL: `http://www.eecs.berkeley.edu/~tfeng/`.

[23] VARIOUS AUTHORS. *Breakout (arcade game).* [web page], 23 December 2008. Wikipedia, the free encyclopedia. [Visited: 9 June 2008]. URL: `http://en.wikipedia.org/wiki/Breakout_(arcade_game)`.

[24] VARIOUS AUTHORS. *Pac-Man.* [web page], 8 January 2009. Wikipedia, the free encyclopedia. [Visited: 9 June 2008]. URL: `http://en.wikipedia.org/wiki/Pac-man`.

[25] VARIOUS AUTHORS. *Metal Gear Solid.* [web page], 7 January 2009. Wikipedia, the free encyclopedia. [Visited: 9 June 2008]. URL: `http://en.wikipedia.org/wiki/Metal_gear_solid`.

[26] PC WORLD COMMUNICATIONS, INC. *U.S. Video Game Industry Grows 43 Percent Over 2007.* [web page], 18 January 2008. PCWorld. [Visited: 29 May 2008]. URL: `http://www.pcworld.com/article/141493/us_video_game_industry_grows_43_percent_over_2007.html`.

[27] IROBOT CORPORATION. *iRobot: Robot® Roomba® Vacuum Cleaning Robot cleans routinely so you don't have to. Shop the iRobot Store for Roomba Vacuum Cleaning Robots.* [web page], December 2008. iRobot Corporation. [Visited: 14 January 2008]. URL: `http://store.irobot.com/category/index.jsp?categoryId=3334619`.

[28] IROBOT CORPORATION. *iRobot: iRobot Create® Command Module*. [web page], December 2008. iRobot Corporation. [Visited: 14 January 2008]. URL: `http://store.irobot.com/product/index.jsp?productId=2586253`.

[29] SIMON BRIDGER. *SourceForge.net: RealTerm: Serial/TCP Terminal*. [web page], 28 February 2008. SourceForge.net. [Visited: 12 April 2008]. URL: `http://realterm.sourceforge.net/`.

[30] JOE LUCIA. *iRobotCreate - iRobot Create Project Page HOME*. [web page], 20 August 2008. Google Pages. [Visited: 28 April 2008]. URL: `http://irobotcreate.googlepages.com/`.

[31] IROBOT CORPORATION. *Create - iRobot Create Community* [web page], 16 July 2008. iRobot Corporation. [Visited: 28 April 2008]. URL: `http://createforums.irobot.com/irobotcreate/`.

[32] IROBOT CORPORATION. *iRobot: BAM Wireless Accessory*. [web page], December 2008. iRobot Corporation. [Visited: 14 January 2008]. URL: `http://store.irobot.com/product/index.jsp?productId=2649971`.

[33] IROBOT CORPORATION. *iRobot: Accessories: Roomba: Original Series: Original Series Add Ons: Standard Remote*. [web page], December 2008. iRobot Corporation. [Visited: 14 January 2008]. URL: `http://store.irobot.com/product/index.jsp?productId=2172879`.

[34] IROBOT CORPORATION. *iRobot Corporation: iRobot Create Projects*. [web page], December 2008. iRobot Corporation. [Visited: 1 June 2008]. URL: `http://www.irobot.com/hrd_right_rail/create_rr/create_fam/createFam_rr_projects.html`.

[35] JÖRG KIENZLE. *SoCS Colloquium*. [web page], November 2008. McGill University. [Visited: 20 January 2008]. URL: `http://www.cs.mcgill.ca/announcements_and_events/colloquium`.

[36] VARIOUS AUTHORS. *Alan Emtage*. [web page], 21 November 2008. Wikipedia, the free encyclopedia. [Visited: 16 May 2008]. URL: `http://en.wikipedia.org/wiki/Alan_Emtage`.

[37] MELISSA ROLLOCK. *Before Google ever was*. [web page], 26 October 2008. The Nation Newspaper. [Visited: 15 December 2008]. URL: `http://www.nationnews.com/story/292944299389588.php`.

[38] SILVIA MUR. *latex-bibitemstyler*. [web page], 13 January 2009. Google Code. [Visited: 13 January 2009]. URL: `http://code.google.com/p/latex-bibitemstyler/`.

[39] TINO WEINKAUF. *TeXnicCenter*. [web page], 7 December 2008. Tino Weinkauf. [Visited: 25 February 2008]. URL: `http://www.texniccenter.org/`.

[40] IROBOT CORPORATION. *iRobot Corporation: Government & Industrial Robots*. [web page], January 2009. iRobot Corporation. [Visited: 23 January 2009]. URL: `http://www.irobot.com/sp.cfm?pageid=109`.

[41] BEN KAGE. *Roomba maker iRobot also developing military robots for Pentagon.* [web page], 14 December 2006. NaturalNews. [Visited: 23 January 2009]. URL: `http://www.naturalnews.com/021301.html`.

[42] WADE ROUSH. *iRobot Wins $3.75M Army Contract to Develop Warrior Robot | Xconomy.* [web page], 2 October 2008. Xconomy Boston. [Visited: 23 January 2009]. URL: `http://www.xconomy.com/boston/2008/10/02/irobot-wins-375m-army-contract-to-develop-warrior-robot/`.

[43] XJ TECHNOLOGIES. *AnyLogic - Xjtek.* [web page], 29 December 2008. XJ Technologies Company. [Visited: 18 December 2008]. URL: `http://www.xjtek.com/anylogic/`.

[44] THE MATHWORKS. *Simulink - Simulation and Model-Based Design.* [web page], January 2009. The Mathworks, Inc. [Visited: 18 December 2008]. URL: `http://www.mathworks.com/products/simulink/`.

[45] MARK HAMMOND. *SourceForge.net: Python for Windows ex,ensions.* [web page], 31 July 2008. SourceForge.net. [Visited: 14 April 2008]. URL: `http://sourceforge.net/projects/pywin32/`.

[46] NOAH SHACHTMAN. *Danger Room What's Next in National Security iRobot Scores $200 Million Army Deal; Could Include Former Foe's Machines | Danger Room.* [web page], 3 September 2008. WIRED. [Visited: 23 January 2009]. URL: `http://www.wired.com/dangerroom/2008/09/last-month-irob/`.

[47] SCHOOL OF COMPUTER SCIENCE OF McGILL UNIVERSITY. *McGill Computer Science Alumni Open House Event.* [web page], 16 May 2008. McGill University. [Visited: 16 May 2008]. URL: `http://www.cs.mcgill.ca/community/alumni/`.

[48] THOMAS FENG. *DCHARTS, A FORMALISM FOR MODELING AND SIMULATION BASED DESIGN OF REACTIVE SOFTWARE SYSTEMS.* [web page], 28 April 2004. McGill University. [Visited: 20 January 2008]. URL: `http://msdl.cs.mcgill.ca/people/tfeng/thesis/thesis.html`.

[49] PYTHON COMMUNITY. *Python Programming Language – Official Website.* [web page], 23 December 2008. Python Software Foundation. [Visited: 14 January 2008]. URL: `http://www.python.org/`.

[50] CHRISTIAN SCHENK. *MiKTeX Project Page.* [web page], 30 December 2008. Christian Schenk. [Visited: 25 February 2008]. URL: `http://miktex.org/`.

[51] CHRIS LIECHTI. *SourceForge.net: pyserial » pySerial: Python Serial Port Extension.* [web page], 6 July 2008. SourceForge.net. [Visited: 14 April 2008]. URL: `http://pyserial.wiki.sourceforge.net/pySerial`.

[52] UPI. *iRobot to develop Warrior 700 for Army - UPI.com.* [web page], 3 October 2008. UPI. [Visited: 23 January 2009]. URL: `http://www.upi.com/Security_Industry/2008/10/03/iRobot-to-develop-Warrior-700-for-Army/UPI-74301223062313/`.

Prof. Hans Vangheluwe
School of Computer Science
McConnell Engineering Bldg, Office 328
McGill University
3480 University Street
Montreal, QC H3A 2A7

Ecole d'Informatique
Pavillion McConnell, Bureau 328
Université McGill
3480, rue Université
Montréal, QC H3A 2A7

Tel. : +1 (514) 398 44 46
Fax. : +1 (514) 398 38 83
E-mail : hv@cs.mcgill.ca
URL : www.cs.mcgill.ca/∼hv

Ms. Silvia Mur Blanch
Enginyeria i Arquitectura La Salle
Universitat Ramón Llull
Barcelona, España

17 May 2007

Dear Ms. Mur Blanch,

This letter is to formally invite you to spend the period of *5 January – 31 July 2008* in the School of Computer Science at McGill University in Montreal, to perform work on your Master's thesis under the immediate co-supervision of Professor Hans Vangheluwe. For the duration of your visit, your status at McGill will be that of an *Academic Trainee*. This means that you will not have official student status and will not receive any degree from McGill, but that you must be officially registered as a student at Universitat Ramón Llull during your entire stay, which should be less than six months. However, you will have full access to Professor Vangheluwe's lab facilities (i.e., the Modelling, Simulation and Design Lab). If needed, you may also have library borrowing privileges, and a SOCS E-Mail account. You will not be invited to participate in conferences and will not receive any funding or payments from Canadian sources. Finally, please note that you must get private insurance coverage before leaving your country. We hope that these arrangements will be convenient, and are looking forward to your arrival in Montreal.

Sincerely,

Prof. Dr. Hans Vangheluwe

Prof. Hans Vangheluwe
School of Computer Science
McConnell Engineering Bldg, Office 328
McGill University
3480 University Street
Montreal, QC H3A 2A7

Ecole d'Informatique
Pavillion McConnell, Bureau 328
Université McGill
3480, rue Université
Montréal, QC H3A 2A7

Tel. : +1 (514) 398 44 46
Fax. : +1 (514) 398 38 83
E-mail : hv@cs.mcgill.ca
URL : www.cs.mcgill.ca/∼hv

7 May 2009

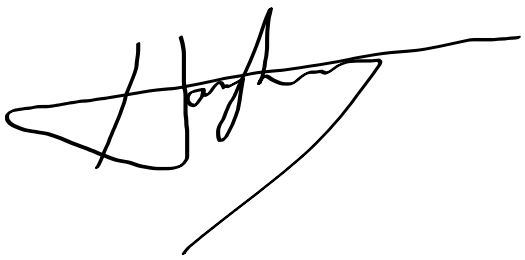
To whom it may concern,

This letter is to certify that Ms. Silvia Mur Blanch of La Salle Universitat Ramón Llull in Barcelona was a Visiting Student in my Modelling, Simulation and Design Lab (MSDL) in the School of Computer Science at McGill University in Montréal, Canada for a period of 6 months: 14 January – 15 July 2008.
During that period, Ms. Mur Blanch was active at McGill in a number of ways:

- during the McGill Winter Term, she audited my COMP 763 advanced graduate course "Modelling and Simulation Based Design". She completed all assignments as well as the course project. Her solutions were excellent and her score for the course is equivalent to "A" (the maximum grade).
- during the latter part of her stay, she worked under my supervision on an infrastructure to use Statechart models to specify the behaviour of the "iRobot" robot. Such models can then be used to simulate behaviour as well as synthesize real-time code. The work was excellent and will form the basis for our further developments in this area.
- during a School of Computer Science Open House day for alumni and industry, she represented the Modelling, Simulation and Design Lab, presenting a poster of her work. The poster was well-made and appreciated by the attendees.

If you require any additional information, please do not hesitate to contact me.

Sincerely,



Prof. Dr. Hans Vangheluwe