The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

A LANGUAGE-BASED SOFTWARE FRAMEWORK FOR MISSION PLANNING IN AUTONOMOUS MOBILE ROBOTS

A Thesis in

Computer Science and Engineering

by

Mirza A. Shah

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2011

The thesis of Mirza A. Shah was read and approved* by the following:

John Hannan
Associate Professor of Computer Science and Engineering
Thesis Adviser

Mahmut Kandemir
Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

---

*Signatures on file in the Graduate School.

Abstract

Autonomous mobile robots have many uses in both commercial and military applications. Creating the computer software that implements the autonomy of these robots is a daunting task, which has resulted in the development of many tools across both industry and academia to help mitigate development complexity and costs in the form of software libraries, frameworks, and application programmer interfaces. This thesis describes a new software framework, the *Modular Planning Framework and Language (MPFL)*, for developing the mission planning aspect of a mobile robot's autonomy. MPFL is unique in that it looks at the problem of vehicle planning as a programming language with a corresponding runtime compiler that processes the language. This approach to autonomy development leverages concepts from programming language theory to create a planning framework that supports component reuse, strong data type verification, and the ability to reason about complex autonomy solutions in a piecemeal fashion.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank all those that made this work possible. My advisor and committee head, Dr. John Hannan, whose tireless guidance and understanding of programming language theory was crucial to the completion of this thesis. My other committee member, Dr. Mahmut Kandemir. My colleague, mentor, and friend, Mr. Matthew Davis, for all the conversations we had over the years regarding the topics in the thesis and his impeccable and thorough proofreading of several drafts. My manager at Penn State's Applied Research Lab, Dr. Jeff Weinschenk, who suggested the research topic and also helped proofread the thesis. I would like to acknowledge my close friends Deepak, Vivek, Ross, Tom, and Alex whom unknowingly helped and inspired me over the years with our late night debates and conversations regarding computation, artificial intelligence, psychology, economics, philosophy, and countless other topics. Finally, I would like to thank my lovely sister, Alvira, whom perpetually pestered me (in a good way) until I completed this thesis.

For my favorite scientist: my father.

Chapter 1

Introduction and Roadmap

1.1 The Future is Autonomous

In our daily lives we take for granted countless tasks that are autonomously handled by non-human entities. Computers autonomously control our financial transactions, traffic lights, airline schedules, the amount of gasoline injected into our car's engine, and even the temperature of our homes. Everything from our home appliances to our breakfast cereal are made by tireless robots in factories overseen only by a handful of people. These computers and robots are all instances of ***autonomous systems***. What qualifies a system as being *autonomous* is subjective but in general it can be understood as some entity that can operate on its own, with little or no outside intervention, in order to accomplish some set of goals. Each living creature on this planet can be considered an autonomous system with the base goals of survival and reproduction. However, in the context of this thesis, autonomous system refers primarily to an artificial entity, such as a computer or a robot. We have increasingly become dependent on autonomous systems to do tasks for us because those tasks are simply too mundane, complex, time/energy consuming, costly, and/or dangerous if performed by a human. The inevitability is that in order for us to progress as a society, we will have to continuously delegate more work to autonomous systems.

1.2 Autonomy and Robotics

One of the most common examples of autonomous systems are ***autonomous mobile robots***, also commonly referred to as ***autonomous vehicles***. Autonomous mobile robots are robots that are both self-governing (i.e. autonomous) and can move around (i.e. mobile). These robots may operate aerially, on the ground, on water, and/or underwater.

1.2.1 Commercial and Research Autonomous Robots

Many commercial autonomous mobile robots have become mainstream. Some of the notable ones available in the market are iRobot Corporation's *Roomba* vacuum cleaning robot (Figure 1.1), Friendly Robotics' *RoboMow* lawnmower robot (Figure 1.2), Sony Corporation's *AIBO* robotic pet dog (Figure 1.3), and Adept MobileRobot's *PatrolBot* security guard robot (Figure 1.4).

Other robots, though not commercially available, are prototypes for potential commercial products. The well-known Honda *ASIMO* (Figure 1.5) is a *humanoid* robot that attempts to replicate human anatomy and movement. In 2003, history was made

Fig. 1.1 iRobot's *Roomba* autonomous robot for vacuum cleaning. In addition to vacuuming, the Roomba is a popular platform for robot hobbyists and researchers due to its low price and hacker-friendly interface (Tribelhorn and Dodds 2007). *(Courtesy of iRobot Corporation)*

Fig. 1.2 Friendly Robotics' *RoboMow* autonomous lawn mowing robot. *(Courtesy of Friendly Robotics)*

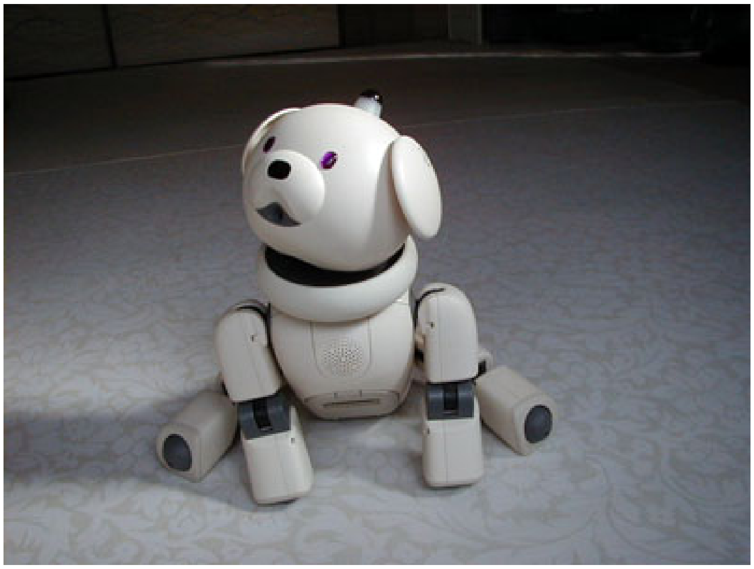


Fig. 1.3 The Sony *AIBO* is a robotic pet designed to mimic a dog. *(Courtesy of Sony Corporation)*

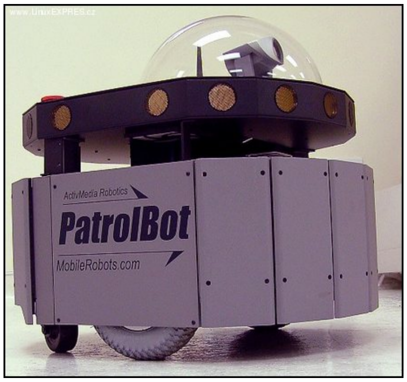


Fig. 1.4 Adept MobileRobots' *PatrolBot* is an autonomous robot for surveillance and security. Note the ultrasonic sensors (the copper mesh patches) circling the chassis and camera on top. *(Courtesy of Adept MobileRobots)*

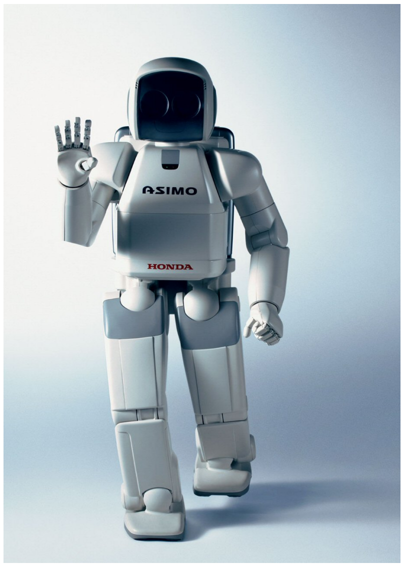


Fig. 1.5 Honda's *ASIMO* (Sakagami et al. 2002) is a humanoid robot; a robot meant to mimic the anatomy of a human. *(Courtesy of Honda)*

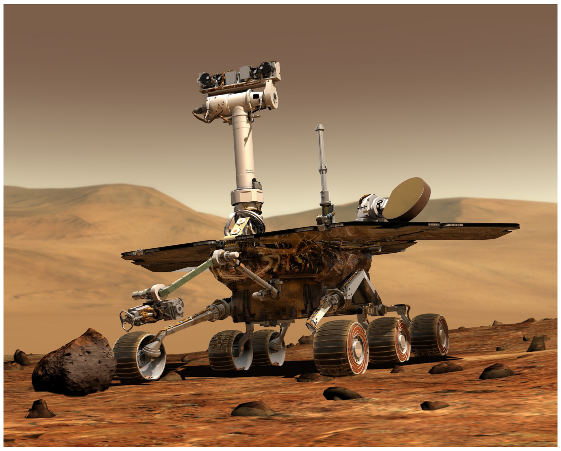


Fig. 1.6 A depiction of one of the twin Mars rovers *Spirit* and *Opportunity* used in NASA's *Mars Exploration Rover Mission. Courtesy of NASA*

during NASA's *Mars Exploration Rover Mission (MER)* when the twin autonomous rovers *Spirit* and *Opportunity* (Figure 1.6) landed on the surface of Mars and are still operational till this day.[1]

1.2.2 Military Autonomous Robots

In the United States, autonomous and remote-controlled unmanned robotics are heavily used in military applications. In the last two decades, the United States Department of Defense (DoD) has become extremely interested in unmanned mobile robots and have invested heavily in the area. In recent years, General Atomics' *Predator* unmanned aerial vehicle (UAV) (Figure 1.7) has been used heavily in combat in the U.S. *War on Terror*. Autonomous underwater vehicles (AUV) such as iRobot's *Seaglider* (Figure 1.8) and Hydroid's *REMUS* (Figure 1.9) have been utilized in applications such as environmental monitoring, mine counter measures (MCM), and hydrographic surveys. Boston Dynamics' *Big Dog* (Figure 1.10), though not yet fielded, will eventually act as a robotic mule that can carry heavy supplies in out and of a battlefield. Robots such as iRobot's *PackBot* (Figure 1.11) are used for explosive ordnance disposal applications(e.g. disabling a bomb), exploring dangerously damaged buildings for survivors, and reconnaissance in urban warfare. Using unmanned robots is cheaper than using manned vehicles, puts military personnel out of harm's way, and allows reaching places that humans cannot get to or safely navigate (e.g. collapsed building). DoD endeavors such as the Defense Advanced Research Projects Agency's (DARPA) *Grand Challenge* and *Urban Grand Challenge* (Figure 1.12) contests were widely publicized events aiming to push the state-of-the-art in autonomous vehicles. In fact, the United States Congress has passed laws that have set a mandate for at least one-third of all aerial vehicles to be unmanned by 2010 and at least one-third of all ground vehicles to be unmanned by 2015 (Laird 2009).

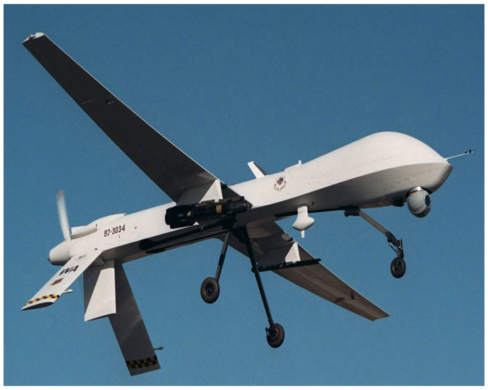


Fig. 1.7 General Atomics' *Predator* aerial unmanned vehicle drone *(Courtesy of General Atomics)*

---

[1] In 2010, *Spirit's* wheels became stuck in soil, turning it into a stationary platform, though it is still operational.

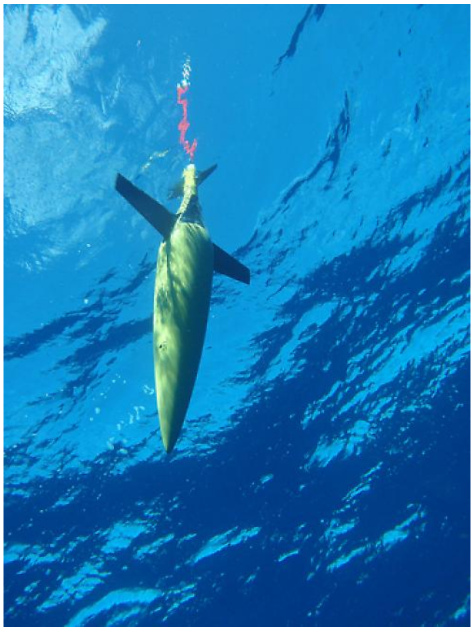


Fig. 1.8 iRobot's *Seaglider* autonomous underwater vehicle (Eriksen et al. 2001) utilizes buoyancy changes to *fly* through the water in contrast to traditional propeller-based thrust *(Courtesy of iRobot Corporation)*

1.3 How Autonomous Robots Work

Though autonomous robots differ greatly in their designs, their form factor, intended use, payload configuration and computer software, the following high-level algorithm describes the control loop followed by many autonomous robots:

1. Observe the world with onboard sensors
2. Perform data fusion on sensory data and update robot's world perception
3. Decide what to do next based on robot's goals and world perception, and issue commands to robot's actuators if necessary
4. Goto 1

This algorithm is not meant to be an accurate depiction for all robots, but gives the reader a starting framework to reason about robot autonomy. In the robotics literature, this algorithm represents what is sometimes referred to as a *sense-plan-act* or *sense-model-plan-act* autonomy model (Alami et al. 1998; Brooks 1991)[2]. Human beings can also be thought to follow this algorithm in the way we function.

---

[2] In military terminology, the concept of an *OODA (Observe-Orient-Decide-Act) loop* (Boyd 1987) is almost identical to the *sense-model-plan-act* concept. It is common amongst defense contractors working in the realm of autonomous systems to utilize the OODA metaphor.

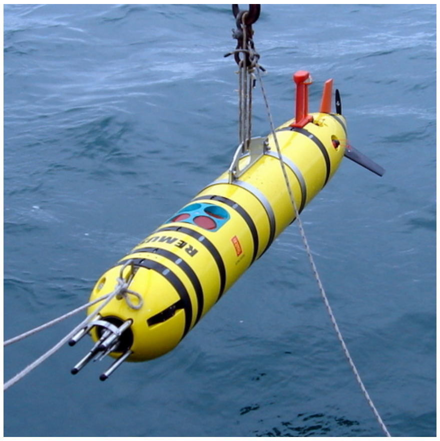


Fig. 1.9 Hydroid's *REMUS 600* Autonomous Underwater Vehicle*(Courtesy of Hydroid)*

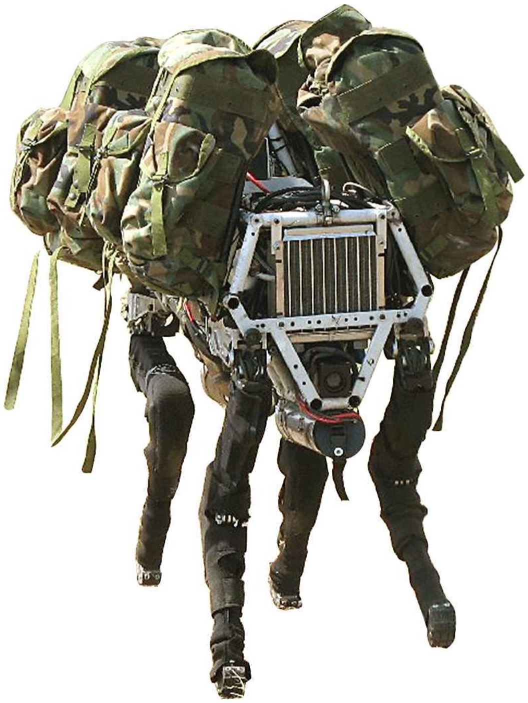


Fig. 1.10 Boston Dynamics' *Big Dog* is an autonomous robot that helps carry heavy supplies across rough terrain like a mule. *(Courtesy of Boston Dynamics)*

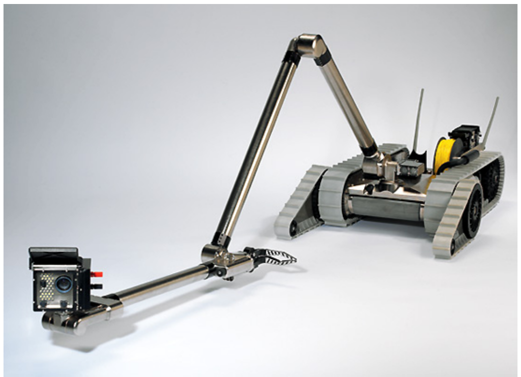


Fig. 1.11 iRobot's *PackBot* is extremely popular in explosive ordnance disposal and other law enforcement/military applications. *(Courtesy of iRobot Corporation)*

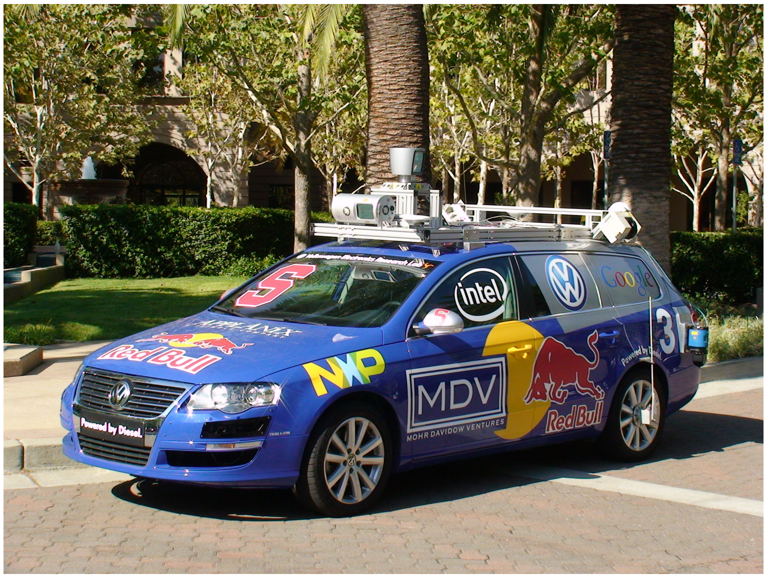


Fig. 1.12 A contender from the DARPA Grand Challenge

1.3.1 Sensors, Actuators, and Payloads

Robots have varying types of ***sensors*** mounted on them for perceiving their surrounding world. Sensors for detecting objects includes infrared, ultrasonic (Figure 1.13), laser, optical cameras, LIDAR (Figure 1.14), sonar, and radar. They act as the eyes of a mobile robot. Other sensors are used for tracking the robot's position and orientation, such as global positional system (GPS), wheel encoders, magnetometers, accelerometers, altimeters, and inertial navigational systems. Environmental sensors are used for measuring properties of the environment, such as thermocouples for temperature and barometers for pressure. Sensors can overlap in responsibility. For example a sensor for pressure can be used to infer either the depth or altitude of the robot. Robots also may carry other types of ***payloads***, such as a robotic arm (Figure 1.15) or grapple to interact with objects, or devices for communication such as a radio transmitter. Robots also have ***actuators*** that are devices to control the mechanical movements of vehicle itself, such as controlling a robotic arm, orienting an onboard camera, or controlling a servo that determines its heading and speed.

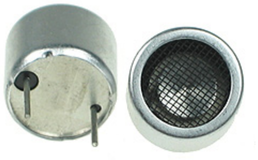


Fig. 1.13 Small ultrasonic sensors commonly used in hobbyist robotics

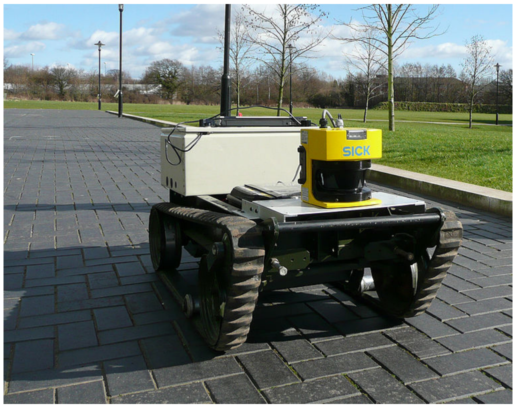


Fig. 1.14 An autonomous robot with front mounted LIDAR

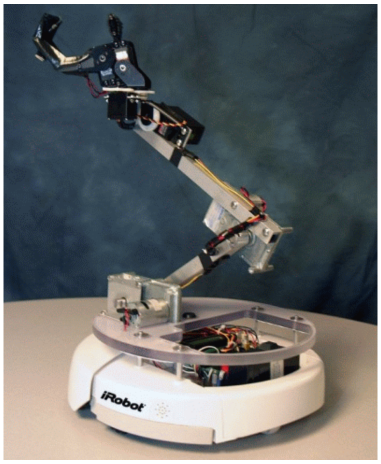


Fig. 1.15 iRobot *Create* mobile robot with a robotic arm (*Courtesy of iRobot Corportation*)

1.3.2 Data Fusion, Reasoning, and Perception

The various data collected by the robot from its sensors are typically processed by a higher-level entity that performs ***data fusion***, such as a digital signal processing (DSP) board and/or a general-purpose computer. Data fusion is the process of taking the clues provided by the various onboard sensors of the robot to help create or update a comprehensive, unified picture of the robot's worldview. This worldview is known as the robot's ***perception***. In artificial intelligence terminology, this is also sometimes known as a ***knowledge base*** which is a place that holds all known facts. Likewise in military terminology, the term ***situational awareness*** is often used. For example, say the robot has two sensors for detecting an object. If both sensors suddenly indicate there is an object bearing X degrees with a range of Y meters, it is a good indicator there is something there. Sensors are not perfect, so sometimes one sensor may say something is there, while another says something is not there or something is there but at a different position. Sensor accuracy also depends on the distance from the object as well as environmental factors. The job of data fusion algorithms is to perform analysis of all sensor data while taking into consideration the accuracy of each sensor, cross-checking facts with other sensors, as well as previous sensor readings in order to create a unified picture of the world (hence the term *fusion*).

The implementation of the knowledge base is important in terms of how knowledge is encoded, stored, indexed, queried, and modified. When human beings are in a given situation, their minds automatically and instantly *prefetch* data that would be relevant/useful in that moment of time. For example, if one is in a car accident, their mind and body are flooded with thoughts and emotions that allow them to get out of that situation and continue on with their life (so as to continue to survive and reproduce). Thoughts may include experience from previous accidents, knowledge of what to do in a car accident (e.g. make sure everyone is alright, call the police, do not admit any fault), worries about how the car will be repaired or replaced, and tasks/meetings/obligations that are affected by the incident. Emotions, which themselves can be considered merely data, may also be invoked by the release of bodily chemicals and neurotransmitters such as adrenaline and dopamine. In the car accident analogy, emotions may include fear and caution so as to create conditions that improve probability of success. The mind reacts so quickly because of the way data is encoded within the brain; facts are not merely stored but they have links between them. In a robot, we attempt to mimic this using tools from computer science. Memory is our physical storage. Thoughts are encoded in the form of programming language constructs and data structures (e.g. records, tuples, abstract data types, classes/objects, linked lists, arrays, binary trees). Thoughts can be cross-indexed via indexes (a common technique in databases) which themselves are metadata in the form data structures (e.g. hash tables, B-trees). Data can also be quickly recalled via data caching and exploiting data locality principles, which is of course a deeply-studied subject in computer science.

The perception/knowledge base of a robot can contain more information than what is processed by data fusion. Just like humans, robots can also infer/deduce other facts with some level of certainty. This act of acquiring new information with already known information via deduction and inference is known as ***reasoning***. For example, if a person sees that it is gray and cloudy outside, they can infer that there is a reason chance of rainfall based on past experience. If thunder is heard, it increases the likelihood even further as it is rare to have thunder without rain. Once drops of rain start falling, a person can infer that it is indeed 'actually raining'. The ability to draw new facts from existing ones is important ability for both humans and autonomous robots in order to interact with world.

1.3.3 Planning and Response

Even though human beings still ponder the purpose of their existence and lives, the purpose of existence for an autonomous robot is very simple: do what is necessary to achieve a set of ***goals*** desired by the robot's user. A goal is a desired state the robot attempts to achieve. For example, a robot may have the goal of reaching some position X. The process of creating the necessary steps to achieve goals is known as ***planning***, and is represented by step 3 of the algorithm mentioned at the beginning of Section 1.3. The result of planning is a ***schedule*** which is a timestamped series of steps needed to be executed at the appropriate times to achieve a goal. For the example of "go to point X", the planning phase may involve determining what heading the robot would need to take in order to get X from its present location. The schedule resulting from planning would indicate the robot should move along the planned heading. The robot may need to continuously replan the heading to take into account obstacles and navigational sensor inaccuracy. When the robot eventually gets to point X, the robot's planning system will recognize the conditions for achieving the goal have been met, and mark it complete. The act of executing the steps in the plan is know as the ***response*** of the vehicle. When a robot completes all of its goals via its responses, it has served its purpose. Goals may also be accompanied by ***constraints*** such as time or energy usage that the robot must stay within when attempting to achieve its goals.

Planning is difficult to develop in an autonomous robot as there are many things to consider:

- What are the smaller steps that need to be taken by the robot to achieve a goal? What steps have already been taken?
- If the robot has two or more goals that, if attempted, would overlap in some resource usage (such as time), how does it interleave these tasks together? If they cannot be interleaved because they conflict, how is that handled?
- How does the robot plan for goals that have interdependencies and require collaboration?
- If the robot does not think it can perform the tasks needed to achieve some goals, for reasons such as hardware failures or resource constraints, how does it handle that? What if later on it changes its mind and believes it can handle the task?
- If the robot plans on doing other things in the future, how does that affect its decisions now? (e.g. if a certain amount of a resource is needed to do a future task, will there be enough of that resource available when the task commences?)
- How does the robot plan when working with other robots each with its own goals as well as shared goals across robots?
- How does the robot deal with incomplete or uncertain information when planning?
- Is the robot planning optimally? As there are often multiple ways to solve a problem, one way may be better in terms of resource usage (e.g. takes less time, uses less power).

1.3.3.1 Reactive versus Deliberative Planning

Some autonomous robots have extremely simple planning systems, such as iRobot Corporation's *Roomba*, whose main goal is *to obtain clean carpets*. It moves about a room in straight lines and when it hits a wall, it moves in a random direction until it hits another wall. Sensors underneath the robot measure dirt level of the carpet. If the sensor detects nothing after a prolonged period of time, the carpet is considered clean. The planning does not take into consideration what areas have been visited, which places are likely to have dirt, etc. Even though the Roomba works in a fully autonomous fashion, its simplistic planning system results in vacuuming times of hours for a single room, versus the mere minutes it would take a human to do it. Other robots, such as the famous NASA Mars exploration rovers, *Spirit* and *Opportunity*, can do more sophisticated things such as path planning to avoid obstacles, as well as filtering of sensory input to send back only the most interesting and relevant telemetry back to Earth (as bandwidth is a limited resource). The planning system of a robot can be described as lying somewhere in a continuous spectrum between being ***reactive*** or being ***deliberative*** (Figure 1.16). Even though the definition is subjective, reactive systems are those that do not take into account previous decisions or anticipate into the future what is to be done when making their next move, making the decision process very simple. Deliberative systems are those that try to take into account information about past moves, as well as try to anticipate what needs to be done in the future. Reactive systems are simpler to implement and debug, but are limited in the scope of what their autonomy can do. The Roomba is an excellent example of a reactive system. Likewise, the Mars rovers are comparatively more deliberative than the Roomba as they *put more thought* into the way they make their decisions.

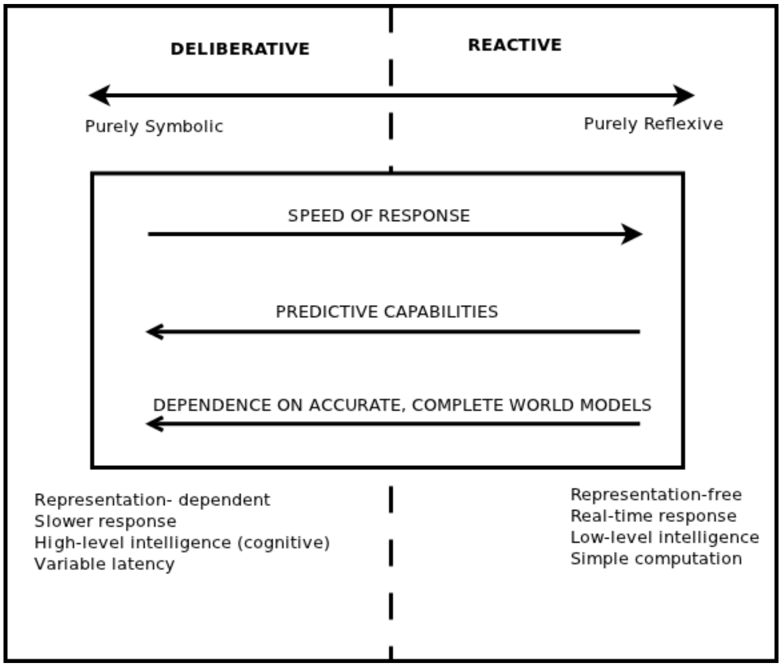


Fig. 1.16 The reactive-deliberative spectrum of robotic planning. (Image recreated from Arkin (1998))

1.3.4 Learning

Learning is the ability for a robot to reason about past experience (e.g. previous sensor readings, current progress in achieving goals, etc) when attempting to achieve goals in order to modify data fusion, reasoning, and planning/response algorithms to increase the probability of achieving current and future goals. For example, a robot with a robotic arm may know how to open doors with door knobs. However, if it encounters a door with a latch, it may not know how to open it. If the robot could somehow understand that not all doors open in the same way and figure out a way to open the door, it could *learn* how to open it.

Learning is a difficult problem. Current techniques for learning typically involve heavy use of probabilistic methods and are extremely limited in comparison to the learning abilities of a human being. Learning that is similar to that of humans almost requires the robot to have a “a sense of purpose” and an understanding of all the dynamics in the world and the consequences of all its actions. Perhaps it requires some form of metacognition and/or self-awareness.

1.3.5 Putting it all Together

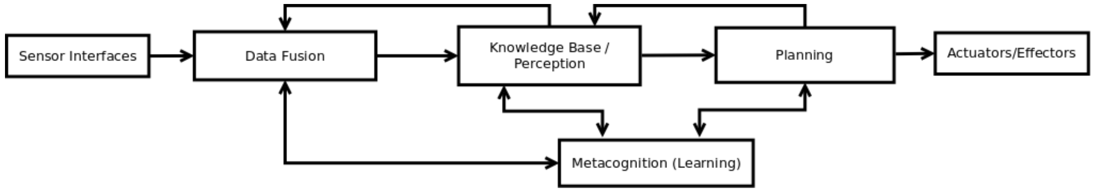


Fig. 1.17 The *sense-plan-model-act* algorithm is implemented as a set of independent agents communicating information.

A typical high-level software architecture used for robotics autonomy is shown in Figure 1.17. It provides a graphical form of the control loop algorithm mentioned at the beginning of Section 1.3. The arrows in the diagram indicate data flows and each box can be thought of as an independent agent. These agents all run in parallel and asynchronously send and receive information from the other agents. It is common place in robotics to implement the software in a similar fashion as it takes advantage of multiprocessing, is more robust because if an agent fails the other agents continue running, and the design is more modular making it flexible to swap out agents.

1.4 The Development and Challenges of Mission Planning in Mobile Robots

Sensors, data fusion, perception, and planning/response are what make up the essential core autonomy of an autonomous robot. Though learning can improve performance of an automous system, it is not necessary to create a useful robot, though the robot will have limited creativity regarding how it can solve a problem and may give up sooner than necessary.

The data fusion, perception, and planning/response components of an autonomous robot’s autonomy tend to be realized in the form of computer software that runs onboard the robot itself. The development of this software is a problem of not only computer science, but of software engineering as well. In general, software complexity and size grows with increasingly sophisticated robot autonomy. More sophisticated software engineering techniques must be employed to manage all the complexity.

This complexity has motivated the development of a myriad of software tools, libraries, frameworks, and application programmer interfaces (APIs) to help develop the autonomy of the robot. This thesis describes a new software framework called the ***Modular Planning Framework and Language (MPFL)*** that focuses on the planning aspect of autonomy development in mobile robots, particularly ***autonomous underwater vehicles (AUVs)***, even though the concept could be generalized to any autonomous system. MPFL distinguishes itself from other autonomy frameworks as it looks at the problem of planning as a programming language. The framework is tied closely to a special domain-specific programming language for describing planning problems that inherently provides many interesting features including:

- Strong static and runtime verification mechanisms for mission execution integrity
- A modular framework that allows component reuse across different robots
- Naturally supports the development of heavily deliberative systems
- Provides an elegant way of handling system and planning failures through exception handling

1.5 Roadmap

Chapter 2 gives some high level background concepts from various disciplines that inspired the design of MPFL. Chapter 3 gives an overview of MPFL. Chapter 4 describes MPFL's programming language: the *Mission Specification Language (MSL)*. Chapter 5 explains how MPFL is utilized and how its works internally. Chapter 6 discusses a proof-of-concept demonstration system that implements the complete autonomy of a simulated robot that uses MPFL. Finally, Chapter 7 is an epilogue that provides analysis of the strengths and weaknesses of MPFL and areas of future research to improve the concept.

Chapter 2

Background Concepts

This chapter introduces some concepts that were used in implementing MPFL. Connections are drawn between these concepts and will eventually be referenced in subsequent chapters.

2.1 The Relationship Between Computation and Artificial Intelligence

The kernel of computer science lies in the theory of computation. No statement summarizes the power and potential of computation more so than the ***Church-Turing Thesis*** that roughly states *any effectively calculable function can be computed by a Turing machine, or one of its [Turing-equivalent] equivalents.* An ***effectively calculable function*** is a function that can be represented as a sequence of discrete steps, or in other words, as an algorithm. A ***Turing machine*** is a hypothetical digital computer that has infinite memory, and [supposedly] can compute any effectively calculable decidable function using only a finite portion of its memory. A ***Turing-equivalent*** computer is a computer that can compute the same set of functions as a Turing machine (Sipser 2006). Typical everyday computers, such as a home personal computer, would be considered Turing-equivalent if they had infinite amounts of memory. However, since any decidable computation uses a finite, bounded amount of memory, one can assume that if we cannot run an algorithm today because of memory or compute time constraints, we maybe able to in the future due to technological advances in computer memory and speed. Even if we cannot compute in the present day, there maybe suboptimal approximations that are sufficient.

Even though there is no formal proof for the Church-Turing Thesis, all evidence to date hints at it being true. Many believe that all processes in our universe are effectively calculable, even though this a subject of debate. If this somehow turned out to be true, it would imply (through the Church-Turing Thesis) that our universe may simply just be a computer program in execution (as depicted by many works of science fiction)[1] or could even be simulated on a computer given sufficient amounts of memory and time. At the same time, it is an exciting prospect that *anything is possible through computation.* This includes being able to create intelligent and sentient beings in our own image as mere computer algorithms.

Artificial intelligence (or ***AI***) is the branch of computer science dedicated to creating such artificial beings. Though the idea of *computing the universe* may seem far

---

[1] *Digital physics* is the area of physics and cosmology that believes that the universe is describable as computable data. *Pancomputationalism* is the belief that the universe is simply a digital computer in execution.

out to many, the idea of at least being able to mimic the mind through computation seems much more plausible. The idea that the *mind is a computer* has now spread to other fields such as neuroscience, psychology, physics, philosophy, linguistics, and biology, and has become a de facto abstraction for reasoning about intelligence. In fact, all of these fields (including computer science and artificial intelligence) have merged into a single field called ***cognitive science*** dedicated to studying the human brain and intelligence.

2.2 The Relationship Between AI and Autonomy

The focus of this thesis is not so much on AI, but rather on ***autonomous systems***. A system that is autonomous is an entity that can operate with limited or no intervention from another entity to achieve a set of goals. For example, automated assembly robots in factories, self-driving cars, autonomous mobile robots (which are the focus of this thesis), home security alarm systems, automatic thermostats, and even human beings are autonomous systems. It is not necessary for an autonomous system to be sentient, intelligent, or even complex. However, as we want more complex autonomy, we find that the system has to exhibit properties that are held only by *intelligent*[2] beings. Hence, creating autonomous systems can be thought of as an artificial intelligence problem.

When developing the software and hardware that act as the *mind* of an autonomous system, one needs to look at various aspects of intelligence that AI researchers have attempted to model computationally. Key areas that are commonly highlighted in the AI literature and depicted in Figure 2.1 are:

- *Sensing* - How does an intelligent entity perceive its environment and how it affects it?
- *Data Fusion* - How does an intelligent entity interpret raw sensor data from multiple sources? How does it merge together information when sensors agree as well as contradict each other?
- *Knowledge Representation* - How does an intelligent entity store, relate, modify, update, and query knowledge to do its tasks?
- *Reasoning* - Using what is already known to an intelligent entity, how can it infer or deduce other facts?
- *Learning* - How can an intelligent entity adjust its behavior without intervention, factoring in previous experience and knowledge, to handle situations unforeseen to it?
- *Planning and Response* - How does an intelligent entity break down complex tasks into simpler steps to accomplish those tasks, while taking into account other tasks

---

[2] The definition of *intelligence* is controversial in the fields of artificial intelligence and cognitive science. The thesis purposely does not define this term. The reader is encourage to go with their intuitive notion of the term.

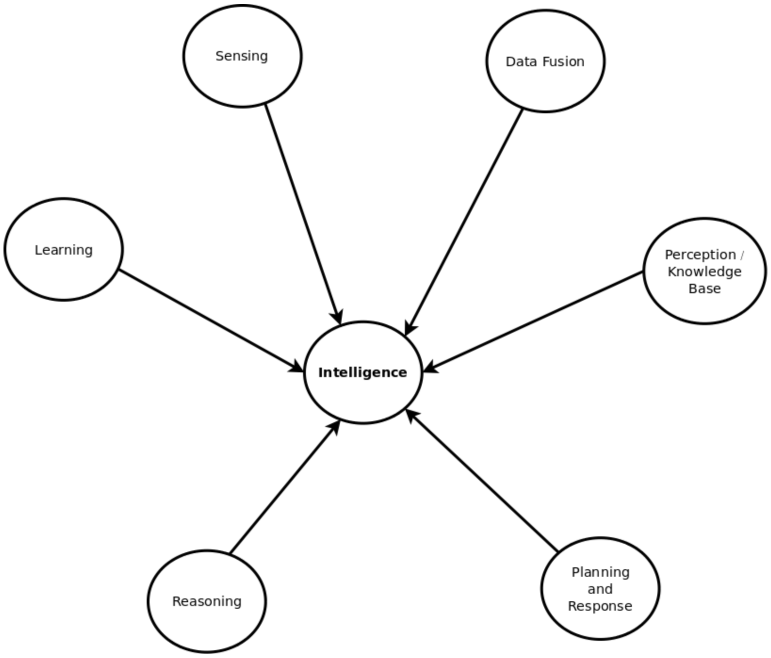


Fig. 2.1 Artifacts of Intelligence

that must be done concurrently or in the future, while managing resource allocation?

Even though to date AI researchers have failed to create ***strong AI*** (also known as ***general AI***), such as sentient computers that parallel or exceed human intelligence (i.e. *the Singularity*), many useful tools that we take for granted in computer science and mathematics emerged from, were improved by, or were popularized by AI research. These include expert systems, constraint solvers/automatic theorem provers, natural language and symbolic processors, fuzzy logic, probabilistic logic, evolutionary computation (e.g. genetic algorithms, particle swarm optimization), database query engines, and neural networks (Russell and Norvig 2010).

2.3 The Relationship Between Intelligence, Natural Languages, and Programming Languages

In linguistics, psychology, anthropology, and sociology, the ***Sapir-Whorf Hypothesis*** (also known as the ***Linguistic Relativity Hypothesis***) states that thought and behavior are a function of language (Kay and Kempton 1984). Essentially it is saying that a person's thoughts and behavior are shaped by the expressive constructs of the language in which they articulate said thoughts (the level to which this is true remains controversial). For example, the *Pirahã* tribe of South America has no numeric system, or even words for quantification such as *each, many, some,* etc. There was never any need for any sort of quantification system in that tribe, hence it never evolved into the language (Colapinto 2007; von Bredow 2006). At the same time, that tribe has not and will likely not discover the concepts and tools that come with more advanced mathematics as it is difficult to reason about or express those concepts without some sort of language (e.g. equations and variables). The extent to which the Sapir-Whorf Hypothesis is true is a highly controversial topic along researchers. However, there is a general consensus that language and thought are indeed intertwined; the controversy lies in the *extent* to which language affects thought (Carruthers 2002). Another example supporting the Sapir-Whorf Hypothesis is that multiple studies have shown people who know more than one language have certain advantages in terms of intelligence such as better problem solving abilities and longer information retention in the face of neurological diseases such as Alzheimers (Bialystok 2010, 2007; Craik et al. 2010). Psychologists suspect this is the case because people familiar with more than one language are able to reason about the same concepts from multiple perspectives as well as have redundant neural encodings of information sets.

The Sapir-Whorf Hypothesis applies to programming languages as well. Kenneth Iverson indirectly asserted this in his Turing Lecture on the importance of mathematical notation in reasoning about computation (Iverson 1980). The notion is quite obvious to a person that has programmed in multiple programming language paradigms. Take for example a computer program that takes as input an algebraic expression as a string and then outputs a reduced expression. To a programmer familiar with logic-based programming languages (such as Prolog), such a task is trivial; they would encode the reduction rules of algebra as logical predicates, and they would be done. The same problem in an imperative language such as C would likely baffle even the most adept imperative programmers; a parser would have to be created to break the expression into smaller subexpressions recursively, then each subexpression would have to be matched against the known reduction rules and evaluated, and finally the result of each evaluated subexpression would have to be passed up during the recursive unwind to the larger expression it constitutes to complete that expression's evaluation. Even if the programmer knew how to solve the problem, the amount of code it would take would be significantly more than that of a logic-based language. Though the Church-Turing Thesis implies that all Turing-equivalent models of computation are equivalent in what they can compute, it says nothing about how easy it is to express an algorithm in one type of computational model versus another. That is where the Sapir-Whorf Hypothesis comes into play; the reason why the example above favors the logic-based language is because the computational model of logic-based languages maps naturally to the problem of algebraic expression reduction, whereas the imperative computational model does not. Like natural languages, it is common wisdom amongst programmers that learning more programming languages, particularly those in a new programming language paradigm, broadens one's programming abilities. It allows one to tackle a wider variety of problems more quickly. This is because it gives one multiple vantage points from which to view a concept, just as those who can speak multiple natural languages.

2.4 The Relationship Between AI and Programming Languages

There is a rich history and relationship between artificial intelligence and programming languages. In the early days of AI, a common approach was to look at intelligence as a symbolic processing problem. The concept came from Allen Newell and Herbert Simon's famous ***Physical Symbol System Hypothesis*** which theorized that a physical symbol system (i.e. a closed formal system) is all that is really necessary to create any type of intelligent agent (Newell and Simon 1976). What Newell and Simon mean is that thinking is really just symbolic manipulation where thoughts and actions can be encoded as a closed system of symbols and rules to manipulate said symbols (where rules themselves can be encoded as symbols). AI researchers refer to this symbolic approach to AI as ***classical AI***, ***Good Old-Fashioned AI*** or simply ***GOFAI*** (pronounced "go-fy") (Russell and Norvig 2010). This is in contrast to other paradigms such as the non-symbolic ***connectionist*** approach. Connectionists believe that intelligence can be modelled from a set of interconnected simple computational units that result in an emergent intelligence (e.g. neural networks). John McCarthy, another major AI pioneer and the one who first coined the term *artificial intelligence*, was also a believer in the symbolic approach. This is one of the main reasons why he developed the famous programming language, *Lisp*, which is designed for list processing. Symbolic expressions are just lists of symbols, hence Lisp is an excellent language for doing symbol processing. *Prolog*, a logic-based language built years later, also was used heavily in AI applications as it allowed the creation of physical symbol systems in the form of logic rules (logic systems are an example of a physical symbol system). Lisp, its descendant Scheme, and Prolog were among the most dominant languages for implementing GOFAI algorithms for many years and are among the most common languages used to teach AI in universities today.

2.4.1 Symbol Processing, Languages, and Semantics

The notion of physical symbol systems is another way of looking at computation. In a sense, the physical symbol system hypothesis is restating the Church-Turing Thesis in that any physical process can be encoded as an algorithm that is computable by a Turing machine. The idea of symbol processing can be seen as creating a language interpreter, or in other words generating a programming language. Every symbolic expression can be manipulated based on a rule that is pattern matched against the expression's symbolic structure, and result in a new output expression. The rules encapsulate the meaning of each symbolic expression, and define what are known as the ***semantics*** of a programming language. Essentially, creating a programming language is analogous to creating a physical symbol system, such as an intelligent system.

2.4.2 Knowledge Representation as Types

As stated earlier, one of the challenging areas of artificial intelligence is knowledge representation; how does one represent, organize, access, and modify knowledge in a computer? One way to approach this problem is reasoning about knowledge in terms of a programming language's ***type system***. Virtually every programming language has a notion of ***types***. Types are classifiers for values/objects/expressions encountered in the language which restrict the operations that can be performed on or by those values/objects/expressions. A type system is a closed formal system (i.e. a physical symbol system) that defines all of those restrictions. An easier way to think about it is looking at types in our own world. When we look at objects, we distinguish them by their properties. For example, when we see a *chair*, we assume certain things about it; it has a seat for us to sit on, and some mechanism to support it (such as four legs), and possibly a back to it. We can sit on it, stand on it, place other objects on it. We cannot use it to *drive to work* or *listen to music*; such operations do not make sense. In programming languages, types are valuable because they restrict invalid operations on values (Pierce 2002). They also allow the user to reason about their program more easily as types give structure and meaning to data, just as in the real world our mind ascribes types to give objects structure and meaning. Some of the tools that type systems provide which could be useful in implementing a knowledge representation scheme are listed in the following subsections.

2.4.2.1 Composite types and abstract data types

Almost every programming language has the ability for users to define their own types in the form of ***composite types***. These types are typically constructed as a composition of the basic, or ***primitive types***, provided by the language. Some languages take this a step further and allow the user to define a set of operations that can be performed by/on values of that type. These types are called ***abstract data types*** (Sethi 1996).

2.4.2.2 Hierarchical Knowledge Representation Through Subtyping

Many type systems allow the notion of ***subtyping***, where data can be considered to have more than one type through a type hierarchy (Pierce 2002). For example, consider our real world example of the type *chair*. Each of us probably imagines a different image when we think of a chair. Some may think of an office chair which swivels and sits on wheels. Others may think of a lawn chair, a living room chair, or a dining chair. Languages that support subtyping allow programmers to define a type such as *lawn chair* as a subtype of *chair*. This means a *lawn chair* has all the properties of a general *chair* (along with any operations if *chair* is an abstract data type), as well as any additional properties unique to a *lawn chair* (e.g. being waterproof). What makes this interesting is that in the course of the program, wherever a *chair* is expected, a value of type *lawn chair* can be used because *a lawn chair is a chair*. However, a *chair* is not necessarily a *lawn chair*. Subtyping allows users to define knowledge in a hierarchical fashion, which turns out to be a valuable abstraction as cognitive scientists believe that the mind stores and relates many concepts in this manner (Minsky 1986).

2.4.2.3 Reasoning Through Type Inferencing

The previously mentioned aspects of intelligence, *reasoning* and *knowledge representation* are not disjoint concepts. One of the interesting features of some type systems is ***type inferencing***. With type inferencing, a user does not have to explicitly classify data with a type. Rather the language interpreter/compiler can infer the type associated with a value based on how the value is used (Sethi 1996; Pierce 2002). For example, if someone was talking about baseball and said “He hit *it* way out of the park.”, one can infer *it* refers to a baseball. Programming languages with type inferencing work in a similar way. For example, in the language *Standard ML* which has type inferencing, we can define a function `addthree` that takes two values, and returns their sum plus 3:

```
fun addthree(a, b) = a + b + 3
```

The compiler infers that `a`, `b`, and the returned value of `addthree` must have type *int* because of the "`+ 3`" at the end of the expression. Standard ML does not allow arithmetic addition between different types of numbers and because the '`3`' is an integer the other operands must be integers. The way Standard ML is able to reason about the structure of data (i.e. knowledge) through inferencing makes it a potentially useful way to implement aspects of reasoning in an intelligent system.

2.4.3 Hierarchical Error Handling through Exceptions

Most computer programs of sufficient complexity require the programmer to handle errors or unforeseen situations. For example, in most languages, attempting to divide a number by zero will cause an error that may result in program termination. Some languages do not have any explicit or sophisticated form of error handling. For example, in C, a typical approach to seeing if a function caused any errors during its invocation is done by either checking the return value of the function or observing a global variable that is manipulated via a side-effect (e.g. *errno* in C) caused by the function that contains the latest error code. A more interesting and structured approach to error handling that is found in many high-level languages is known as ***exception handling*** (Sethi 1996). Exception handling is a technique that separates the normal code from the error handling code. Whenever the programmer wants to indicate an error, they can ***raise*** (also called ***throw***) an ***exception***. Exceptions are entities in the language that represent an exceptional situation, such as an error. Programmers can indicate in the language which blocks of code they want to handle exceptions for by associating an ***exception handler*** with them. When an exception is raised, the program stops its normal path of execution and jumps to the nearest handler defined within the scope of the current call stack. Exceptions are typically values in a language, so they have associated with them data and/or a type. Not all handlers necessarily handle all errors, in which case the type information is used to match the nearest handler for that type within scope. If no handlers are found, the program will crash with some sort of *unhandled exception* error. What is interesting about exception handling is that it creates a hierarchical way to deal with errors. As an analogy, if you have a problem with a home appliance, you may call the support line of the company that manufactures the appliance. If the customer service representative cannot help you, they may contact their supervisor and pass the problem up to them to handle. If they cannot help you, the problem will keep rippling up, perhaps to the owner of the company. If nobody helps you, your problem is not resolved like an unhandled exception. In a computer program, handling an error at the place it occurs may not be the best place to do it as somebody higher up in the call stack would have a better context of what to do.

2.4.4 The Power of Abstraction - Sapir-Whorf Revisited

When John McCarthy created Lisp, he did so because there was something lacking in then existing languages like FORTRAN. FORTRAN did not have the tools needed to express certain types of computations; in this case it lacked facilities for symbolic processing. Not having tools to deal with symbolic processing makes it difficult to create complex symbolic processors. This relates back to the Sapir-Whorf Hypothesis which argues that thought is a function of language. There are thousands of programming languages in existence; each designed to look at computational problems from a particular point of view. Some problems like the algebraic expression reduction example mentioned in Section 2.3 is trivial in Prolog versus C, whereas some problems in C are trivial compared to Prolog. Looking at computational problems as programming languages gives one the ability to create natural abstractions to express problems for a particular problem domain.

2.5 The Relationship Between AI and Domain-Specificity

In the early days of AI, researchers who took the symbolic approach to AI assumed they could implement intelligence as a single, unified formal system. One of the famous attempts at doing this was the *General Problem Solver (GPS)* program created by Newell and Simon (Newell et al. 1959). GPS allowed a user to specify any problem in any domain as a combination of a start state, state transformation rules, and goal states. GPS attempts to then create a path, using means-ends analysis, from the start state to the goal state via the state transformation rules. In other words, GPS builds a proof tree by applying rules sequentially until it can conclude the goal state can be reached. The problems with this concept were that 1) it was difficult to come up with all the rules for a particular problem and more importantly 2) as soon as there were a significant number of rules, the performance worsened. The reason the performance went down was because of a phenomenon known as ***rule explosion*** that occurs in automated provers. GPS performs a depth-first search when constructing its proof. It has to examine every sequence of transformations from the start state that may lead to the goal state. If a path proves to be unfruitful, the prover has to backtrack and explore another path. The problem is that as more rules are added, the number of paths increases significantly. This increase is known as rule explosion. Rule explosion is an instance of the ***search problem*** in artificial intelligence: how does one efficiently examine the problem space to find a solution? For example, if a person wants to infer whether it is going to rain outside, they are going to check the sky to see if it is cloudy. However, another rule that person may have in their head that has the same antecedents such as *if the sky is cloudy, it's a bad day to go tanning* would be completely irrelevant. GPS would not distinguish between *if cloudy, it's going to rain* and *if cloudy, it's bad to go tanning* as it does not distinguish between the relevance of rules and therefore does inefficient search. Examples like GPS have led many AI researchers to believe that the fundamental notion of implementing intelligence in a single, closed system is virtually impossible primarily because those systems cannot inherently do efficient searches. However, many still believe that the mind is implemented using a limited set of base constructs (i.e. a kernel) from which all human cognitive abilities emerge.

However, cognitive science has found evidence that the mind is a set of domain-specific entities interacting with each other. In other words, different portions of the brain are dedicated to doing very specific tasks. For example, in recent years neuroscientists isolated certain areas of the brain with neuron *patches* dedicated purely to facial recognition (Buchen 2008). If these *patches* were manipulated, we would not recognize people or perhaps recognize them as someone else. Evolutionary psychologists believe that everything the mind is stems from our basic instincts: to survive and reproduce through the process of natural selection and evolution. Evolutionary psychologists predominantly believe the human mind is a series of domain-specific entities dedicated to helping us simply survive and reproduce and were formed through the process of natural selection (Cosmides and Tooby 1994; Cosmides and Tooby). The idea of domain-specificity has spread to AI as well, mainly through the study of ***agents***. Agents are individual entities (usually in the form of a computer program) that are responsible for handling some task, typically isolated to a narrow domain, on behalf of a user (Russell and Norvig 2010; Minsky 1986). Agents can talk to other agents to provide or receive services, and may have the power to take actions autonomously. One of the advantages of isolating problems to narrow domains is that the problem is much easier to reason about. Even using a formal system like GPS in a domain-specific situation may not be a bad idea as 1) the rule set is reasonably small enough to avoid rule explosion and 2) the domain is small enough to describe as a rule-based system. In general, by breaking up AI problems into smaller, domain-specific problems (such as agents), it is easier to reason about and create intelligent entities.

2.6 The Relationship Between AI and Domain-Specific Programming Languages

The ideas of domain-specificity and programming languages can be combined together to create a powerful approach to solving AI problems. Most mainstream programming languages can be classified as being general-purpose programming languages. These languages are designed to solve any general problem. Languages such as *Ada, FORTRAN, C, C++, Java, Smalltalk, StandardML, Ruby, Perl, PHP, Python,* and *OCaml* are classified as *general-purpose languages*. This is in contrast to ***domain-specific programming languages*** or ***DSPLs*** which are meant to solve a problem for a particular problem domain such as *Standard Query Language (SQL), Make, VHSIC Hardware Specification Language (VHDL), A Mathematical Programming Language (AMPL), regular expressions, LaTeX,* and *Unix Shell Scripts* (e.g. *Bash, Tsh*). These languages are not meant to solve every problem, but solve problems in their domain well, even though they may be Turing-equivalent. Domain-specific languages are an advantageous approach to computation problems because they:

- *Provide language constructs that naturally fit to a domain and its experts in order to make it easier to write programs for that domain.* For example, researchers at MIT designed a domain-specific programming language for microfluidic chips that could be programmed to perform chemical experiments for biology applications involving the mixing of chemicals within the chip (Thies et al. 2008). The domain-specific language is used to describe the experiment to be performed: which chemicals to mix, which quantities and concentrations to use, and specifying which intermediate products should be used as reagents in subsequent reactions. The compiler for the language then generates a set of commands for a special microfluidic chip "printer" that in turn fabricates a disposable, single-use microfluidic chip. The chip is then connected to a special interface into which the chip inputs valves are connected to a supply of reactants and the experiment commences. Output chambers fill with final products as they are formed. The syntax of the language is tailored to that of biologists so that it is easier for them to build and reason about their experiments. A simple syntax that is specific to their domain does not require them to understand a more complex general-purpose language.

- *Provides verification at the domain-level to help better increase operational correctness and reduce development costs.* One form of this could be a domain-specific type system which would likely be able to do more rigorous checking of values in the language that could only be known at the domain level. For example, in the future, a hypothetical domain-specific programming language that instructs a hypothetical *robot surgeon* how to perform surgery may be able to find mistakes that a surgeon might have forget such as *Anesthetic X cannot be used on the patient because they have an allergy to it according to their patient history.* Mistakes like this could be caught by the interpreter or compiler for the language.
- *Increases reuse and better system-to-system interfacing through simplicity.* When a language is both expressive and simple to use, it will catch on faster and be more timeless than its analog in the form of a library-based application programmer interface. As the language stays relatively static, the implementation can change freely as long as it complies with the syntax and semantics of that language. Examples of languages that exemplify simplicity, elegance, and notable flexibility and ease in interfacing include:
    - *UNIX Shell Script* - A typical UNIX shell scripting language, such as *Bash*, is a language that is designed to control the operating system and the programs it runs within an interpreter for that scripting language (e.g. the Bash shell is the interpreter for the Bash scripting language). UNIX shell scripts have a reputation to be able to do incredibly powerful things with very simple programs and have been in mainstream computing for over 30 years.
    - *Standard Query Language (SQL)* - Virtually every website on the Internet that has a database backend uses a *relational database management system* or *RDBMS* such as *MySQL, Oracle, IBM DB/2, PostgreSQL* or *Microsoft SQL Server*. RDBMSs are databases that store and query data based on a formal database model known as the ***relational model***. SQL is a language designed for performing data queries in RDMBSs by incorporating the relational model in its underlying computational framework. Virtually every RDBMS utilizes or can utilize SQL as the user interface to query data. The language allows

a developer to communicate with any database, regardless of how it is implemented, via an extremely simple language. Since 1979, relational databases have dominated the database world and the SQL domain-specific language takes a great role in that achievement.

- *Decoupling from a General-Purpose Programming Language.* A domain-specific language does not need to be tied to a specific programming language. For example, the most common way to talk to an SQL-based RDBMS from application code is by forming SQL statements as strings and sending them to the database's query engine via a database-provided API. The database then processes the query using the SQL interpreter/compiler within the query engine and returns the result.

2.6.1 Libraries versus Domain-Specific Languages

The purpose of domain-specific languages is to provide an interface to a user to solve some problem in a domain. However, most domain-specific interfaces do not come in the form of their own language, but rather more commonly come in the form of ***libraries*** that have an ***application developer interface (API)*** defined as constructs native and natural feeling to the language (such as a collection of classes in an object-oriented language, or a collection of procedures for procedural languages). Libraries provide a uniform and natural way for developers to utilize domain-specific tools in their programs that fits in with their thinking when developing applications. Even though libraries tend to be bound to a specific language, it is now common in modern software development to create ***bindings*** for popular libraries across many languages, making them even more accessible. Even without bindings, most languages provide a means to utilize code from other programming languages. It is also typically cheaper to develop software libraries versus languages as 1) creating programming languages is not a skill typical to software developers, 2) developing a language that really encapsulates the domain well is extremely difficult and 3) it is time consuming and burdensome to implement features of a language, such as its type system, properly. When attempting to build interfaces for a domain, deciding when to use a domain-specific language versus a library is a subjective choice based on what solution fits the problem best (Heering and Mernik 2002; Mernik et al. 2005). The following are some criteria from Mernik et al. (2005) that would qualify a domain-specific language as a better approach to solving a problem versus a software library:

- A relatively simple language can tightly encapsulate the essential aspects of the domain.

- The primary users will be people that are not comfortable with learning a general-purpose programming language to perform their desired tasks. The users are likely experts in the domain the DSPL is designed for.

- Higher costs of developing a language instead of a library must be able to return investment by making it easier, cheaper, and faster to build and maintain solutions for a domain's problems.

2.6.2 Looking at Software Development as a Programming Language Design Problem

Looking at problems as the implementation of a programming language may result in better quality software. Programming language development is a rigorous process that requires everything to be specified (either formally or informally) to guarantee that what a programmer types in actually happens based on the semantics of the language (Mawu et al. 2002). The language developer is required to define a syntax for the language as well as static and runtime semantics. If these are not defined, then a programmer cannot write a program in the language, either because it's impossible or they cannot define what an expression in the language does. In other words, developing a programming language forces the language developer to think very hard about what the operational meaning of every construct in their language is. In the software development world, it is common when developing software (such as libraries) to ignore certain cases that the application may come across either out of time constraint, laziness, or oversight. It is common in the development world to see incomplete or frequently-changing APIs. These ignored cases are what cause a vast portion of software bugs. Tools to aid in developing software such as flow charts, block diagrams and the graphical *Unified Modeling Language (UML)* provide vague specifications of how some software application is meant to operate. The tools used by programming language developers have no such ambiguity. The grammar (i.e. specification of the syntax) and the semantics of a programming language can be formally specified in mathematical notation. Any person familiar with those mathematical formalisms can then create an implementation of the language in the form of an interpreter or compiler that behaves exactly as the specification says it should. In contrast someone trying to describe the behavior of simple module of a program in UML could create dozens of UML diagrams, but still end up with an ambiguous specification.

2.7 The Relationship between Intelligence, Planning, and Scheduling

The specific focus of this thesis is on implementing a framework that focuses around planning, mentioned briefly in Chapter 1. ***Planning*** is the process of creating the steps necessary to achieve a set of ***goals***. Goals are end states the autonomous system would like to be in by affecting itself and its surrounding world. For example the goal of being at some position X is achieved when the autonomous system is actually located at position X. The resulting series of steps required by the autonomous system in order to achieve its goals are called a ***schedule***. The autonomous system can break up a goal into several ***subgoals*** that may not be disjoint from each other in order to make it easier to develop its final schedule. For example, when you wake up each morning, you have a set of goals at hand that you would like to achieve. On a typical day, you wake up, decide you need to get ready for work, eat breakfast, then go to work, and eventually come home. You will think about what steps need to be taken to accomplish these goals, focusing more closely on the near term goals (like getting ready) rather than the long ones (like what to do when you get home), but keeping the long term ones in the back of your mind. You will also take into account your time requirements, if you have to be at work by 9 am, you attempt to schedule the getting ready, breakfast, and drive to work parts of your schedule to meet that requirements. If you're in a hurry, you may have to skip somethings, like eating breakfast which may have lower priority, than say, taking a shower. You might find that even though you have planned things, the situation around you changes as the environment is dynamic, so you have to replan, and modify your schedule. For example, you may find your car does not start, so you need to get an alternate ride, which causes you to be late for work. Being late for work may alter your others plans as well, affecting the schedule you have laid out. You will then again have to replan and rebuild your schedule.

Planning is an important part of intelligence. It allows human beings to make decisions with care in order to maximize chances of successfully achieving goals. Any *interesting* autonomous system needs to have some sort of mechanism to plan, whether it is reactive, deliberative, or somewhere in between.

2.8 Planning as Processes

In English, the word ***process*** refers to a series of actions/occurrences that result in some change and/or development in some environment. We use the term process to describe how physical events play through their execution. The way steel is manufactured, how rocks weather, how software is developed, why and how it rains, etc, can all be described as processes. In a sense, a process is similar to the notion of an algorithm.

2.8.1 Processes and Operating Systems

The term *process* has special meaning in computing, particular in the realm of operating systems. In a typical multitasking computer operating system, there are two important concepts: ***programs*** and ***processes***. A computer program typically comes in the form of a binary file containing machine-level instructions that are executed by the computer. Alternatively it may come in the form of files containing source code or bytecode that is interpreted by a runtime interpreter. A process is an operating system abstraction that represents a running instance of a particular program and its associated execution context. This context comes in the form of a data structure, typically denoted as the ***process control block (PCB)***, and contains data used by both the running program and the operating system (Silberschatz et al. 2002). The PCB contains information about the process such as the process identifier, data stack, page table, register contents, stack pointer and program counter. When a user runs a program, the OS creates a new process by creating a new PCB and putting it on the OS' scheduler queue.

The purpose of having processes is two-fold. Having a context associated with every running program allows the creation of multiple running instances of the same program. One can run five instances of the same web browser; each is unique through the contents of its PCB. The other purpose is to make running programs manipulatable objects. The PCB captures the state of the entire machine for a running program, essentially providing a "virtual machine environment" for that program. In a multitasking operating system, processes do not realize they are sharing resources, such as the processor, with other processes. For example, in a single processor system, only one program can execute at a time on the processor. The OS allows many programs to run at the same time by utilizing timesharing. When a process has run for its given timeslice, it is put to sleep in order to allow another process to be revived and run. In order for this to work, the OS saves the machine state for the process that is to be put to sleep within its PCB. The OS then sets the real machine state (e.g. register contents, program counter, virtual-to-physical page mappings) to that contained within the incoming process' last saved PCB. The revived process continues running where it left off previously, not realizing that it was put to sleep and revived.

2.8.2 Modeling Planning in Terms of Processes

When developing planning systems, it can be helpful to think of concepts in terms of processes in the operating system sense. There are several ways to do this. For example, a goal can be thought of as a computer program, and *an instance of an attempt to achieve that goal* can be thought of as a process associated with that goal. If you have the goal of going to several waypoints, you can break each waypoint into its own subgoal. A "program" designed to solve the *get to waypoint* problem can then be reused by creating several instances of that program (i.e. each represented as a process). The output of each process could represent part of the final schedule of what the vehicle is to do in order to achieve its goals. The input represents the goal specification along with the relevant pieces of perception information needed to solve the problem. Another way of looking at it is thinking of programs as individual planning systems, where a running instance of that program waits for goals to be input and outputs schedules to solve the problems for those goals. Multiple types and instances of these planning systems could communicate with each other to develop a final, cohesive schedule.[3]

The advantage in thinking about planning in terms of processes is the same as the reason why multitasking operating systems reason about programs instances as processes: a process is a manipulatable and inspectable entity that can exist in parallel with other processes:

- The operating system can easily perform concurrent execution of several programs, even if its simulated through timesharing.
- The operating system can provide certain degrees of validation, such as making sure processes do not violate their address space or perform invalid operations.
- The operating system optimally schedules usage of limited system resources such as memory and processor time.
- The operating system can provide a uniform and safe interface for processes to access system features through standard libraries and system calls.
- The operating system can reason about concurrency more easily as each process encapsulates an isolated system.

---

[3] This concept is an example of a *multi-agent system*. Agents were mentioned briefly earlier and represent a popular approach to developing intelligent systems.

If a planning system is thought of as an operating system that thinks of goal achievement attempts as processes, the planning system can leverage these advantages. The planning system can validate, to a certain extent, each goal achievement attempt, depending on the richness of the content of the PCB-like context associated with that attempt. It can make sure goals do not step on each other, provides a uniform interface to reach perception information, validate schedules, error handling, etc. Many goal attempt instances can be created from the same goal, reducing the amount of code needed.

2.8.3 Processes and Programming Language Semantics

Most mainstream languages encourage sequential, rather than concurrent, thinking — particularly imperative languages. This is one of the reasons why many programmers have difficulty implementing multi-threaded applications: the Sapir-Whorf Hypothesis bites them as their language does not naturally incorporate concurrency into its computational model and style.

One of the things that make it difficult to add constructs into a language for concurrency is that the semantic models people use are inherently sequential. For example, a popular type of semantics is ***operational semantics***. This semantics describes how expressions are evaluated in a language in a sequential manner. In pure functional languages, any subexpression in an expression can be evaluated in any order[4], meaning all the subexpressions could be evaluated in parallel, but is not an inherent requirement of the computational model. With certain types of operational semantics, such as *big-step operational semantics*, you could imply that several subexpressions can be evaluated concurrently, but there is no explicit way of doing this.

Several types of mathematical models have been created for modeling concurrent systems, including programming languages with constructs for concurrent computation. Examples include the *Actor model* and several ***process calculi*** also known as ***process algebras*** (Baeten 2005). Process calculi are interesting in this context because semantics are expressed in terms of processes and the communications and relationships (such as sequential versus parallel execution) between those processes.

2.9 Segue: The Big Picture

This chapter has described several concepts; particularly intelligence, language, computation, domain-specificity, and autonomy. What makes these concepts important in this thesis is how they molded the thinking process in designing MPFL. The next chapter will describe what MPFL really is and how combining many of these concepts together can provide a powerful way to develop autonomous robots with sophisticated planning capabilities.

---

[4] This is one of the consequences of the *Church-Rosser Theorem.*

Chapter 3

Overview of the Modular Planning Language and Framework

This chapter gives a high level overview of the central topic of this thesis, the *Modular Planning Language and Framework.*

3.1 Redundancies in Autonomy Development

Developing sophisticated autonomy software for a robot is a very difficult and expensive task. One snag is that it is common practice for developers to create customized autonomy software for each unique robot, its particular payload, and its intended purpose. One cannot take the autonomy software that runs on mobile robot A and put it on a completely different mobile robot B in a transparent manner, and expect B to behave like A (or even to work at all). This is because B may have completely different sensors, propulsion system, onboard computer, and/or payload from A.

To solve this problem, developers of autonomy development frameworks try to create software abstractions that create a unified interface across multiple vehicles so software can be reused. For example, rather than worry about the details of how the drive system of a mobile robot works, one possible abstraction is to create an interface that allows the user to control the heading and speed of the robot, perhaps in the form of a software library. The developer of the drive system would be expected to provide the underlying implementation that maps to this common interface. If this interface is available across several different types of robots, higher level autonomy layers that utilize the interface do not need to be changed and can be ported transparently across to other robots. The concept is similar to the idea of system libraries and drivers in operating systems. The operating system defines high-level abstractions for the various types of devices that are expected to be used on the computer (e.g. mouse, video card). Hardware developers are expected to write drivers that match the interface defined by the operating system. Application developers can then utilize the device through the abstraction in the form of a standardized API , such as sockets for network communication and file I/O for non-volatile data storage.

Creating common abstractions is more difficult in robotics than with operating systems. This is because robots are so varied in their design and application. Also unlike computer components that are highly standardized (e.g. Intel x86 ISA, USB, PCI, SCSI, VGA, Ethernet, etc), robotic hardware has not reached the same level of commoditization. However, on the other hand there is significant commonality across robots particularly when one focuses on a specific class of robots. For example, there is a large amount of redundancy in the area of autonomous underwater vehicles (AUVs)[1]. Even though AUVs differ greatly in design, payload configurations, and their applications, their primitive functions are essentially the same: move around, sense things (primarily with sonar), operate various onboard payloads, deploy things, and collect things. There is even overlap with higher-level functionality. For example, it is typical to use AUVs to search for and/or monitor objects of interest. Take for instance an undersea warfare application where an AUV is searching for enemy submarines versus a marine biology experiment that is tracking whale migration patterns. Both will likely use similar techniques to detect, classify, and track those objects, whether it be a whale or an enemy sub[2]. Another example is bottom floor mapping where an AUV is used to create a detailed map of the ocean floor. In undersea warfare applications, this can be used to give manned submarines detailed navigational information so that they do not collide with the ocean floor, or perhaps to find more covert routes to traverse. In an energy exploration application, bottom mapping is used to look for potential places to drill for oil.

3.2 Overview of the Modular Planning Framework and Language

The ***Modular Planning Framework and Language (MPFL)*** is yet another attempt to create a tool to address the redundancy in the area of autonomous robot mission planning software. MPFL is a software framework that allows one to develop sophisticated, deliberative planning capabilities for an autonomous mobile robot in a straightforward, guided manner while providing guarantees regarding the operational correctness of the robot. MPFL not only gives a framework for control software developers to build robot autonomy, but it has explicit focus on component reuse. For example, if somebody wants to create a component for executing a search behavior in some autonomous robot X, that component should have the ability to be reused transparently in another autonomous robot Y, giving Y the ability to perform that search behavior. A robot that utilizes MPFL can then be controlled via a special high-level, domain-specific programming language used to specify the goals, operating constraints, and error handling routines for the robot. This domain-specific programming language is known as the ***MPFL Mission Specification Language (MSL)*** and is the foundation of the entire framework.

3.2.1 Planning is to Schedules as Compiling is to Machine Code

The MSL is what makes MPFL powerful and unique. MPFL is a planning system embodied in the form of a compiler for a programming language (the MSL). Utilizing an artificial language to control the robot brings with it many of the advantages of domain-specific programming languages mentioned in the previous chapter, including strong domain-level verification and error handling mechanisms, an intuitive and natural way

---

[1] AUVs are a subset of unmanned underwater vehicles (UUVs), which can also include non-autonomous vehicles primarily *remotely-operated underwater vehicles (ROVs)*

[2] In the example of a whale versus a submarine, both the whale and submarine emit a unique sound signature that can be used to track them with passive sonar.

to control the vehicle, decoupling from a specific general purpose programming language, and a formal operational specification. MPFL's design was influenced by thinking of robotic planning as the compilation of code written in a high-level language (the MSL) to a lower-level representation. Compilers for a high-level general purpose programming languages (such as C or Java) typically compile to machine code for a particular machine instruction set architecture (ISA)[3]. In contrast, MPFL translates a description of the mission specified by the user in the MSL into a set of ***schedules*** (Figure 3.1). In MPFL, a schedule is simply a table of fine-grained, timestamped commands intended for a particular vehicle actuator or subsystem (e.g. sensors, drive system) (Figure 3.2). If the robot follows the schedules, the result will be what the user specified in their MSL program.

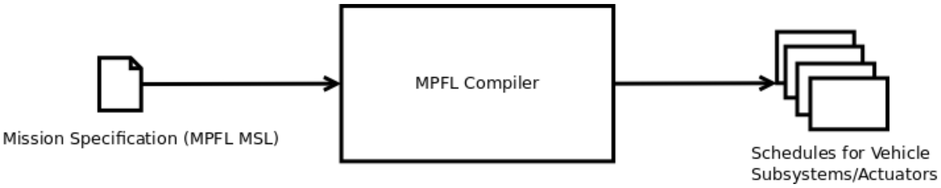
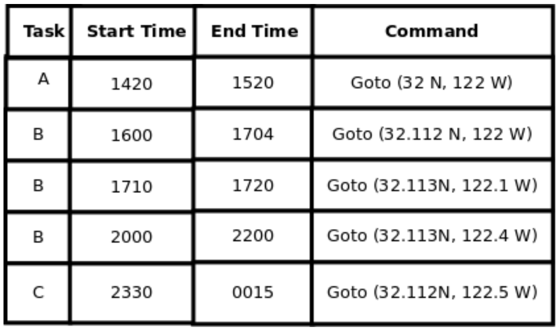

Mission Specification (MPFL MSL)
MPFL Compiler
Schedules for Vehicle Subsystems/Actuators

Fig. 3.1 High level view of MPFL.



Task	Start Time	End Time	Command
A	1420	1520	Goto (32 N, 122 W)
B	1600	1704	Goto (32.112 N, 122 W)
B	1710	1720	Goto (32.113N, 122.1 W)
B	2000	2200	Goto (32.113N, 122.4 W)
C	2330	0015	Goto (32.112N, 122.5 W)

Fig. 3.2 A typical MPFL schedule.

Unlike typical compilers which are standalone applications, MPFL comes in the form of a software library where the compiler is invoked from an application that links against the library. An application which utilizes MPFL is referred to as an ***MPFL client application*** or simply ***client***. The client is the glue between MPFL and the other components of the robot's autonomy software. The client is intended to run physically onboard the robot itself but may also run on a remote computer communicating planning data with the robot via a network link if onboard processing resources are not

---

[3] Languages that compile to an intermediate representation (e.g. Java bytecode) that is to be processed by a virtual machine (i.e. runtime interpreter) are also considered compiled.

sufficient. The client may encompass all the autonomy of the vehicle or maybe one of many operating system processes running onboard the vehicle that constitute the whole autonomy software stack. Within the client, the user initializes the MPFL framework via an API call passing the name of a single source file containing their mission description written in the MSL. MPFL then parses the specification while performing static checks (e.g. type checking, enforcing scoping rules). If successful, the ***MPFL compiler/runtime*** (also can be referred to as the ***MPFL compiler*** or the ***MPFL runtime***) is loaded into memory which stays resident until the client application has ended. The user can then ask MPFL to compile the mission specification code into a set of schedules. The schedules that are returned by MPFL to the client can then be used to control the vehicle. If the user follows the schedules carefully, the mission that was specified in the MSL can be accomplished.

What makes MPFL different from a normal compiler is that the job is not complete after a single compilation. The environment of robots is constantly changing and information the robot has can be erroneous or incomplete requiring replanning as newer and more complete information is acquired. For example, if a robot is asked to go to some waypoint within some time constraint, it may initially think the task can be done. However, halfway to the point the robot's sensors may pick up an obstacle requiring replanning so that the robot can move around the object. In MPFL, one can perform replanning simply by reinvoking the compiler from within the client periodically. The MPFL compiler/runtime is stateful so rescheduling takes into account partially and fully completed goals. The control software developer writing the MPFL client can choose to invoke the compiler as needed depending on their requirements.

3.2.2 Expressiveness of the Mission Specification Language

What makes MPFL unique is the MSL. The MSL is a highly declarative language where one can specify problems that seem really difficult but can be solved in a straightforward manner by MPFL. For example, with the MSL one can specify a seemingly complicated mission such as the following:

I have three physical regions `A`, `B`, and `C`. `A` and `B` are defined as a cubic region whereas C is defined as a cylinder. I want the robot to search all three regions for objects of interest while in parallel reporting what they find to base at least once a minute. The robot can perform `A` and `B` in any order, but can only search `C` after finishing the former two. After searching all three areas, the robot needs to return to one of several potential pickup points, which one does not matter but the robot must get there before noon the day after tomorrow and must not arrive at the pickup point before midnight of the current day. In addition, region `C` must be searched before 8 pm tonight.

One can specify this mission in the MSL quite easily and MPFL will be able to develop a set of schedules to solve it. In fact, it does not matter how complicated the mission is, MPFL will try to solve it without violating constraints. In the event the task is impossible or tasks conflict, MPFL provides constructs to handle the situation both within the framework and within the MSL. In addition to being expressive, the MSL like other programming languages has a notion of types as well as exception handling. These features help users create specifications that are both clear and correct.

3.2.3 Customization and Reusability Via Plugins

The core MPFL library is not a complete out-of-the-box solution for planning by itself. To use MPFL, developers must build or acquire prebuilt plugin modules known as ***planners*** that define and implement some of the primitive scheduling capabilities of the robot. Planners are plugins for the compiler/runtime (Figure 3.3) that are responsible for solving a set of problems of a specific type, and generating a schedule to solve all those problems[4]. The planners are not provided with the compiler/runtime, but are intended to be created by a third party, most likely the vehicle software developers. of the language.

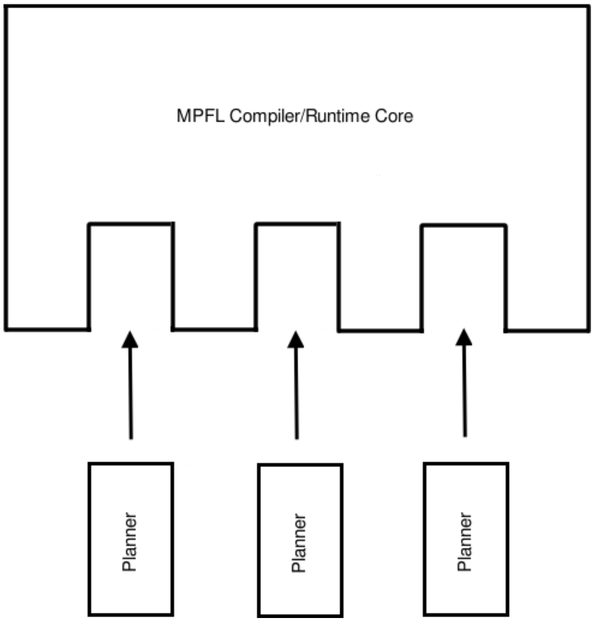


Fig. 3.3 Planners are plugins for the MPFL compiler/runtime that define primitive capabilities of the robot.

To implement a planner, MPFL provides an object-oriented application programmer interface API which requires subclassing from a set of provided base classes. Users pass a set of planners (in the form of objects) to MPFL in addition to their mission specification when initializing the framework from the client. These planners are linked into the MPFL runtime in an analogous fashion to how drivers are linked into an operating

---

[4] For those that have encountered the term *planner* in AI literature, the term has a different connotation as used here. Chapter 7 talks about this more.

system kernel on system bootup. Just as drivers provide the means for the kernel to communicate with the computer's components and peripherals, planners provide a means to the MPFL runtime for scheduling tasks. However by no means are building planners as difficult as writing an operating system driver and requires only implementing a handful of typed callback functions. MPFL's API is designed to be easy to understand and use while additionally providing safeguards to prevent the programmer from implementing a faulty planner. It will be shown later that not only do planners allow one to customize the underlying semantics of the MSL, but also to mix and match planners transparently without having to recompile or relink code.

Planners are what give the MPFL framework its reusability feature. Once someone implements a planner, it can be reutilized in another vehicle (with certain restrictions detailed in later chapters) in a transparent manner without recompiliation. The MPFL framework isolates planners from each other as completely sandboxed systems; this is what makes planners reusable across vehicles. Hence, once a sufficient collection of planners has been created, a vehicle developer can mix and match previously developed planners (even those made by other people), along with any new planners the vehicle requires. Make no mistake, the MPFL compiler/runtime by itself really does not do much in terms of planning. The planners do the hard work of solving the problem; the compiler/runtime itself just provides a well-defined framework that create the bridge between the MSL and planners, such as facilities for verification, error handling, and the actual delegation of problems to planners. A planner developer does not have to worry about the details of how the compiler/runtime works or how any other planners work. All they are focused on is writing their specific planner.

3.2.4 A Software Framework

The MPFL library implements a ***software framework*** in contrast to an ordinary library which is typically a collection of types and functions that can be used by an application. A software framework distinguishes itself from a normal library as it utilizes ***inversion of control*** where control flow of the program is managed by the framework rather than by the one dictated by the user. The client interacts with the framework via callback functions/methods that are fired by the framework as necessary (triggered by some event), but the framework dictates the thread of execution. For example, most widget toolkits such as *Qt, Gtk+, WinForms, WxWindows*, and *Swing* used for creating GUI applications follow this style; the programmer relinquishes control of their application to the framework. As GUI events occur such as resizing the window or clicking on a menu, the framework invokes callback functions/class methods to handle the events. If a user does not provide a callback then typically a default version is used. The advantage of frameworks is that they allow one to more rapidly develop software as it provides guide rails for building an application and good defaults for unspecified behavior.

3.3 Segue: A Language of Planning

In order to understand how the MPFL compiler/runtime and planners work, the first step is to understand the MPFL MSL. The next chapter describes the MPFL MSL and informally explains the constructs in terms of language syntax and semantics, and how they came to be. Later chapters will focus on how the MPFL runtime compiles user specifications written in the MSL, and the significance of planners in the framework.

Chapter 4

A Language of Planning

This chapter details the special language, the *Mission Specification Language (MSL)*, that not only composes the external interface of the framework, but is the foundation of the framework internal implementation.

4.1 The Requirements of a Language for Planning

The MPFL framework focuses on a particular area of autonomy development: *planning*. The other components of autonomy development, (e.g. data fusion, perception, reasoning) are largely ignored though there are hooks in the framework to help a developer using MPFL integrate those components into their total autonomy solution. In MPFL, planning is articulated and reasoned about via the MSL. Before looking at the design of the MSL, it is useful to think about the requirements and features that one would like in a language to describe and encode concepts pertaining to *planning*. One approach is to introspectively reflect on how we humans perform and reason about planning ourselves and how it is reflected in our own natural languages (e.g. English).

In the previous chapter, an example was given describing how a human being might plan their tasks for some given day. When designing the MSL, examples like this were analyzed to determine what sort of language constructs people use to plan. These language constructs are useful in capturing the domain of autonomous robot planning and helped shape the syntax of the MSL. Some of these observations were:

- Humans use phrases like *"I have to shower before I go out"*, *"I have to do this task by 3:00 pm"*, *"I need to go shopping for food, but cannot spend more than 40 dollars"*, and *"If I finish my homework, I can watch the game tonight or read a book"*. These phrases represent ***constraints*** on the way we set out to achieve goals. Some of these constraints are ***resource constraints*** that limit a particular resource, such as time, energy, or money. Other constraints are subordinate clauses that explain relationship constraints between and across goals, such as A happens *after* B, A *can only happen if* B happens, and A *must occur while* B is occurring.

- Humans break down complex goals into smaller subgoals to make it easier to reason about solving problems.[1] For example, the goal of *making a cake* can be broken into goals such as *create batter*, *place batter in baking pan*, *heat baking pan in oven*, *remove baking pan from oven*, and *add icing to cake*. Each of these subgoals can

---

[1] This is a feature that is common to many AI/autonomy techniques and discussed further in Chapter 7.

be broken down further to make it even easier to solve the problem. If one were to graphically depict goals and subgoals, they would form a tree.

- When unexpected problems occur during planning, humans try to circumvent them utilizing the path of least resistance while attempting to limit the involvement of other entities in solving said problems. For example, if a person has a headache, they do not immediately go to a neurologist to get an MRI, as that would be premature and unnecessary; more likely they will take some aspirin. If the headache persists, they may then go to their family doctor. If the family doctor cannot solve the problem, the doctor may pass the problem on to a specialist, and so on. In other words, we try to solve our problems on our own, and pass them onto outside entities only when we cannot solve them in the case we do not have enough information, skill, and/or other resources. When an outside entity cannot solve a problem, they may in turn may pass on the problem to different entity further up the chain.
- Humans can immediately recognize syntactic and semantic errors in sentences allow them to recognize nonsensical or ambiguous statements. Take for example the following sentence: *"There is a gas station ten hertz past the diner."* The sentence does not make sense as *hertz* is not a measurement of distance, but rather of frequency. As another example, take the sentence: *"I am going to make some motorcycle for lunch."* This sentence also does not make sense as *motorcycle* is not a food item. Programming languages can enforce similar sanity checks enforced during syntactic and semantic analysis stages of a compiler/interpreter.

These observations were important in designing MPFL's language of planning and are incorporated into the constructs the language provides.

4.2 The MPFL Mission Specification Language

The language that constitutes the foundation of the MPFL framework is known as the ***MPFL Mission Specification Language (MSL)***. This language is what is used to *specify the missions* the robot is to perform – hence the name. Users of the robot encode the goals they want their robot to achieve along with any constraints in an MSL program. The MPFL compiler/runtime will then process this specification and attempt to help the robot achieve the specified goals.

4.2.1 Language Overview

The MSL is a strongly, statically typed domain-specific programming language encompassing the domain of mission planning in AUVs. The language is highly declarative with very little imperative control. The concept of *verification* is extremely important in the MPFL world; the compiler verifies as much as it can statically before runtime, and whatever cannot be statically checked is continuously verified during runtime. Programs written in the MSL are contained within a single file, though breaking up a specification into multiple files could be achieved easily by employing a simple preprocessor such as the one used in C compilers.

The grammar of the language is defined by an *LALR(1)* grammar with a simple and consistent syntax. Parentheses are used extensively and represent block constructs in the language. White spaces, newlines, and tabs have no significance in the language and are ignored. One will notice that the syntax of the language is heavily influenced by variant types found in functional languages, namely *ML*. The grammar is specified later in the chapter.

4.2.2 Plans and Plan Instances

The primary construct in the MSL is called a ***plan***. A plan represents a set of goals and their corresponding constraints. The MSL has a set of built-in plans called ***primitive plans*** and allows users to create their own plans called ***user-defined plans***.

The primitive plans represent the most basic capabilities of the robot; each encapsulating a singular goal. The MSL's current set of primitive plans is built specifically for ***autonomous underwater vehicles (AUVs)*** but can easily be extended to serve a broader range of autonomous robots. The currently-available primitive plans in the MSL are described in Table 4.1.

Primitive Plan	**Description**
`Search`	Searches a specified area using an onboard sensor
`UseSonar`	Activates an onboard sonar device
`UseModem`	Activates an onboard acoustic modem device
`Transit`	Requests the vehicle to move through a set of waypoints
`PhoneHome`	Sends status and sensory reports back to a command ship or station
`Loiter`	Tells the vehicle to sit at a position and do nothing
`UseAutopilot`	Requests the vehicle's autopilot to move to a waypoint
`UseAcoustic`	Requests allocation of the acoustic channel

Table 4.1 Primitive Plans in MPFL's MSL

4.2.2.1 Plan is a Type

In the MSL, *Plan* is a built-in MSL type. Recall *types* are tags ascribed to values in a language restricting the usage of those values primarily for the purpose of safety and correctness (Pierce 2002). All of the primitive plans mentioned in Table 4.1 are values which can be used in the MSL whereever a *Plan* is expected (i.e. *Plan* is a ***variant type***[2] which can take one of the values from Table 4.1). Values of type *Plan* are called ***plan instances***. *Plans* describe tasks, whereas *plan instances* refer to an actual instance of a task. Another way of thinking about it is the difference between a *class* and an *object* in an object-oriented language; *an object is an instance of a class.* Likewise in the MSL,

---

[2] A ***variant type*** (also known as a ***set type***, ***tagged union*** or ***discriminated union***), is a value representing a value that could take on several different, but fixed types. The value is typically represented as a tag along with some typed value.

a *plan instance is an instance of a plan.* For example, if one wants the robot to move to some waypoint using the MSL, it can be achieved by creating a plan instance of the *Plan* variant type value `UseAutopilot` depicted by the following code segment:

```
UseAutopilot goToPoint(Destination = GeoPosition(Lat = Degrees(32.0),
                                                 Lon = Degrees(-122.0),
                                                 Depth = Meters(10.0)))
```

Example 1 - Declaration of a `UseAutopilot` plan instance

In Example 1, we create a plan instance representing the primitive plan `UseAutopilot` with the identifier `goToPoint`. Following the identifier is a list of parameters enclosed in parentheses. This is referred to as the ***plan instance constructor***. This constructor describes the input parameters for the task represented by the plan instance. In the case of the example above, there is one parameter: `Destination`. Hence, the instance `goToPoint` represents transiting to a geographical position of 32° N, 122° W, and a depth of 10 meters.

4.2.2.2 Plan Instance Constructors and Other Types

Every type of primitive plan in MPFL has a constructor associated with it. Each constructor is different for each primitive plan. For example, the `Search` plan takes parameters such as the area to be searched and the sensor to utilize within the search. Each type of plan may have more than one constructor, allowing several ways to define the problem parameters. Table 4.2 gives the complete syntax for plan instances declarations along with their constructors. Note that values with bars over them (e.g. $\overline{integer}$) refer to the ***metatypes*** used in the implementation language (typically referred to as the ***metalanguage***) for the MPFL compiler.

Every constructor requires each parameter to be named (in the case of Example 1 only one parameter called `Destination` is required for a `UseAutopilot` constructor) followed by an '=', followed by the value. Each parameter name/value pair separated by a comma. The value in each pair has type associated with it. In Example 1, the expected parameter type for the value of the field name `Destination` is the MSL type *Position*.

Besides the *Plan* type, the MSL has many types built into it (such as *Position*). The majority of these remaining types are used for encoding the parameter values in plan instance constructors. All of these types are variant types, which can take one value from a discrete set of differently-typed values described by their own constructors. Table 4.3 describes the majority of types in the language along with the different values they can take (either a constructor, an expression consisting of a binary operator, or a primitive value such as an integer or string) without the constructor. Table 4.4 give the complete syntax for these types in the language along with their corresponding type constructors. Besides constructors, there are also ***operators*** that can be used to create typed expressions. All of the MSL operators are binary and use infix notation. These are also included in Table 4.4 and can be thought of as special constructors where the

Value	**Type**	**Syntax**
Loiter	*Plan*	Loiter <$\overline{string}$>(LoiterPosition = <*Position*> LoiterDuration = <*Duration*>)
PhoneHome	*Plan*	PhoneHome <$\overline{string}$>(CommDeviceName = <*String*>, PhoneHomeRate = <*Frequency*>
Search	*Plan*	Search <$\overline{string}$>(SonarName= <*String*>; SearchArea = <*Area*>, LaneWidth = <*Length*>)
Transit	*Plan*	Transit <$\overline{string}$>(Waypoints = <*Position*>, <*Position*>, ...)
UseAcoustic	*Plan*	UseAcoustic <$\overline{string}$>(AcousticDeviceName = <*String*>, StartTime = <*Time*>, EndTime = <*Time*>, TaskDuration = <*Duration*>, MinGap = <*Duration*>, MaxGap = <*Duration*>)
UseAutopilot	*Plan*	UseAutopilot <$\overline{string}$>(Destination = <*Position*>)
UseModem	*Plan*	UseModem <$\overline{string}$>(ModemName = <*String*>, ModemMessage = <*String*>)
UseSonar	*Plan*	UseSonar <$\overline{string}$>(SonarName = <*String*>, PingRate = <*Frequency*>)

Table 4.2 Declaring Plan Instances in MSL

operator is the constructor tag and the two operands are parameters. The remaining types in the language will be defined as needed.

A *Position* can take on several values such as the value `GeoPosition` which refers to an absolute geographical position (i.e. latitude/longitude). `GeoPosition`, just like a plan instance, has a constructor. The `GeoPosition` constructor consists of a latitude (parameter name `Lat`), longitude (parameter name `Lon`), and a depth (parameter name `Depth`), which must be passed into its constructor, where each of these parameters also has to be named, following an '=' with the respective value on the right. `Lat` and `Lon` have to be defined as values of the MSL variant type *Angle* which can take a value of `Degrees`. Depth requires a value of MSL variant type *Length*, which can take a value of `Meters`. Notice the constructors `Degrees` and `Meters` simply take a number, indicating a terminal type constructor.

Types	**Values**
Angle	`Degrees, Radians`
Area	`RectangularArea, CircularArea, PolygonalArea`
Boolean	$\overline{\mathit{boolean}}$, $>=, ==, != , <=, <, >$, `LookupBoolean`
Constraint	`TimeConstraint, PowerConstraint`
Duration	`Seconds, Minutes, Hours`
Energy	`Joules, KilowattHours`
Float	$\overline{\mathit{float}}$, $+$, $-$, $/$, $*$, `LookupFloat`
Frequency	`Hertz, Kilohertz`
Integer	$\overline{\mathit{integer}}$, $+$, $-$, $/$, $*$, `LookupInteger`
Length	`Feet, Meters, Yards`
Plan	`Loiter, PhoneHome, Search, UseAcoustic, UseAutopilot, UseModem, UseSonar, ExecutePlan`
Position	`GeoPosition, RelativePosition`
Power	`Watts, Kilowatts, Horsepower`
String	$\overline{\mathit{string}}$, `LookupString`
Time	`UnixTime, DHMSMTime`

Table 4.3 Type Constructor Tags in MSL

4.2.2.3 Explicit Parameter Naming and Unit Specification

The reader may have noticed that the syntax in constructors is unusually verbose. The MSL is designed to ensure there is no ambiguity in what the user wants to do and accomplishes this through a very strong type system. Again one of the central tenets of MPFL is providing as much automated verification as possible. When software is running on an extremely expensive vehicle alone in the middle of the ocean potentially for

Value	**Type**	**Syntax**
Degrees	*Angle*	Degrees(<*Float*>)
Radians	*Angle*	Radians(<*Float*>)
RectangularArea	*Area*	RectangularArea(TopLeft = <*Position*>, BottomRight = <*Position*>)
CircularArea	*Area*	CircularArea(CenterOfArea = <*Position*>, Radius = <*Length*>)
PolygonalArea	*Area*	PolygonalArea(BoundaryPoints = <*Position*>, <*Position*>, ...)
TimeConstraint	*Constraint*	TimeConstraint <*String*>(<*Time*> <= StartTime <= <*Time*>, <*Time*> <= EndTime <= <*Time*>)
PowerConstraint	*Constraint*	PowerConstraint <*String*>(MaxPowerLevel = <*Power*>, MaxEnergyToUse = <*Energy*>)
Seconds	*Duration*	Seconds(<*Float*>)
Minutes	*Duration*	Minutes(<*Float*>)
Hours	*Duration*	Hours(<*Float*>)
Joules	*Energy*	Joules(<*Float*>)
KilowattHours	*Energy*	KilowattHours(<*Float*>)
Hertz	*Frequency*	Hertz(<*Float*>)
Feet	*Length*	Feet(<*Float*>)
Meters	*Length*	Meters(<*Float*>)
Yards	*Length*	Yards(<*Float*>)
AbsolutePosition	*Position*	AbsolutePosition(Lat = <*Angle*>, Lon = <*Angle*>, Depth = <*Length*>)
RelativePosition	*Position*	RelativePosition(Center = <*AbsolutePosition*>, X = <*Float*>, Y = <*Float*>, Z = <*Float*>)
Watts	*Power*	Watts(<*Float*>)
Horsepower	*Power*	Horsepower(<*Float*>)
ClockTime	*Time*	ClockTime(Day = <*Integer;*>, Time = <*Integer*>::<*Integer*>::<*Integer*>)
UnixTime	*Time*	UnixTime(UTCSeconds = <*Float*>)

Table 4.4 Non-*Plan* Constructors in MSL

months, a single incorrect value could cause severe problems such as mission failure or the damaging/destruction of the vehicle. By specifying all parameters within a constructor along with their units where applicable, trivial mistakes are avoided.

4.2.3 User-defined Plans

Plan instances have to be declared within an MSL language construct called a ***user-defined plan***. User-defined plans allow a user to extend the set of built-in primitive *Plans* with their own. User-defined plans are similar to primitive plans in the sense that they represent some sort of task/goal description. The difference is that primitive plans represent a singular goal whereas user-defined plans are composed of several primitive plan instances as well as plan instances of other user-defined plans. User-defined plans are the primary construct for abstraction and encapsulation in the MSL and allow a user to organize their missions in a logical way. User-defined plans, like primitive plans, have the type *Plan*.

4.2.3.1 User-defined Plan Syntax

User-defined plans have the following basic syntax

```
Plan <plan identifier>
(
    <plan instance declaration 1>
    <plan instance declaration 2>
    ...
    <plan instance declaration n>

    Do(<plan expression>)
)
```

An example of a very simple user-defined plan taking from Example 1 is as follows:

```
Plan doStuff
(
   UseAutopilot goToPoint(Destination = GeoPosition(Lat = Degrees(32.0),
                                                    Lon = Degrees(-122.0),
                                                    Depth = Meters(10.0)))
   Do(goToPoint)
)
```

Example 2 - A user-defined plan in the MSL consisting of a single goal

Example 2 depicts a complete MSL program consisting of a singular goal (depicted by the `UseAutopilot` plan instance `goToPoint`) to reach a waypoint. If one were to save this plan to a text file and feed it to the MPFL runtime, the robot would go to the waypoint specified.

The final part of the plan declaration is the keyword `Do` followed by some expression. This part of the plan is known as the ***Do Expression*** and contains within it an expression describing the temporal relationships and constraints for goals (i.e. plan instances) defined within a user-defined plan. The example above is rather trivial so we need to expand on the example.

4.2.3.2 Do Expression and Plan Operators

Example 3 expands Example 2 by adding in three more goals:

```
Plan doStuff
(
   UseAutopilot goToPoint(Destination = GeoPosition(Lat = Degrees(32.0),
                                                    Lon = Degrees(-122.0),
                                                    Depth = Meters(10.0)))
   Search lookForThreats(
             SonarName = sideSonar,
             SearchArea = RectangularArea(
                             TopLeft     = GeoPosition(Lat = Degrees(32.231),
                                                       Lon = Degrees(-122.112),
                                                       Depth = Meters(0.0)),
                             BottomRight = GeoPosition(Lat = Degrees(32.231),
                                                       Lon = Degrees(-122.112),
                                                       Depth = Meters(0.0))),
            LaneWidth = Meters(100))

   PhoneHome reportStatus(CommDeviceName = benthos100, PhoneHomeRate = Hertz(0.01))

   Loiter waitForPickup(LoiterPosition = GeoPosition(Lat = Degrees(-32.93),
                                                     Lon = Degrees(-121.991)
                                                     Depth = Meters(0.0)),
                        LoiterDuration = Hours(5.5))

   Do(goToPoint > (lookforThreats || reportStatus) > waitforPickup)
)
```

Example 3 - A user-defined plan in the MSL consisting of multiple goals

The Do Expression in this plan is more complicated. The expression contained within the `Do` constructor represents a relationship between various plan instances (i.e. tasks). Each plan instance is referenced via its identifier. The angle bracket (>) represents a ***serial operator***, meaning that the task on the left-hand side of the > must finish before the task on the right-hand side can begin. In Example 3, the plan instance `goToPoint` must complete before `lookForThreats` and `reportStatus`, which both in turn must finish before `waitForPickup` can commence. The two bars (`||`) between `lookForThreats` and `reportStatus` is known as the ***parallel operator*** indicating that `lookForThreats` and `reportStatus` must execute in parallel. These operators are referred to in the MSL as ***planning operators***. The MSL has several planning operators that can be used in the Do Expression depicted in Table 4.5. These operators are left associative, but can be grouped using parentheses as well. The MPFL compiler verifies that all references within the Do Expression are valid identifiers with corresponding plan instance declarations within the user-defined plan. If the references are invalid it will result in a compile-time error.

Plan Operator	**Meaning**
`Serial (a > b)`	Execute task *b* if and only if task *a* is complete.
`Parallel (a || b)`	Execute task *a* and *b* in parallel. Once either task is started, the other must be forced to start as well.
`Group (a & b)`	Execute both tasks *a* and *b*, however there is no dependency between the two (in contrast to the parallel operator), and both do not need to run at the same time.

`Exclusive Or (Xor)`
`(a ^ b)`
	Execute either task *a* or task *b*, but execute exactly only one of them. Give task *a* priority over task *b*.

Table 4.5 Plan Operators in MSL

The Do Expression is a typed value in the language of type *DoExpression*. The expression inside of the Do Expression has a different type *PlanExpression*. In Table 4.6, we add to the list of types and type constructors from Table 4.3 and Table 4.4.

Value	**Type**	**Syntax**
`Do`	*DoExpression*	`Do(`<*PlanExpression*>`)`
`operator >`	*PlanExpression*	<*PlanExpression*> `>` <*PlanExpression*>`)`
`operator ||`	*PlanExpression*	<*PlanExpression*> `||` <*PlanExpression*>`)`
`operator &`	*PlanExpression*	<*PlanExpression*> `&` <*PlanExpression*>`)`
`operator ^`	*PlanExpression*	<*PlanExpression*> `^` <*PlanExpression*>`)`
`plan instance identifier`	*PlanExpression*	<*string*>

Table 4.6 *DoExpression* and *PlanExpression* Types MSL

Here are some examples of more complex Do Expressions using the operators from Table 4.5:

1. `Do(moveAround || sendStatusHome > goHome > stayPut)` - This example wants us to do the tasks `moveAround` and `sendStatusHome` in parallel, followed by the task `goHome`, then followed by the task `stayPut`.

2. `Do(((a > b) ^ (c || d)) || e)` - This example wants us to either do the tasks `a` followed by `b`, OR do the tasks `c` and `d` in parallel (only one or the other). While it is doing all of that, it wants to do task `e` in parallel.

3. `Do(a & ((b & c) > d))` - This example wants to group the task `a` with the following task: do `b` and `c` together, not necessarily in parallel. Once `b` and `c` are complete, then and only then, perform task `d`.

We can see from these examples that one can create significantly complex temporal relationships between the tasks needed to be done by the AUV. An important note about the Do Expression is that one is not allowed to reference a plan instance in the expression more than once.

4.2.3.3 Creating Plan Instances from User-Defined Plans

The kind of plan instances we have created so far are instances of the various primitive plans built into the language. Creating a user-defined plan results in extension of the set of valid values constituting the variant type *Plan*. We can create instances of user-defined plan types by using the special plan instance constructor: `ExecutePlan` (see Table 4.7).

Value	**Type**	**Syntax**
`ExecutePlan`	*Plan*	`ExecutePlan(UserPlanName = <`*`String`*`>)`

Table 4.7 ExecutePlan Constructor in MSL

We can extend Example 3 with another plan that creates an instance of a user-defined plan as shown here:

```
Plan doStuff
(
    ...
    Do(goToPoint > (lookforThreats || reportStatus) > waitforPickup)
)

Plan master
(
    Loiter waitForMissionStart(LoiterPosition = GeoPosition(Lat = Degrees(-32.44),
                                                            Lon = Degrees(-122.213)
                                                            Depth = Feet(0.0)),
                              LoiterDuration = Hours(5.5))

    ExecutePlan primaryMission(UserPlanName = doStuff)

    Do(waitForMissionStart > primaryMission)
)
```

Example 4 - Creating an Instance of a User-Defined Plan

The `ExecutePlan` constructor takes the identifier of a previously declared (i.e. higher up) user-defined plan. The plan instance created with `ExecutePlan` is an instance of the user-defined plan and like all primitive plan instances also has the type *Plan*. In Example 4, we create a plan instance with user-defined plan `doStuff` (which we declared in Example 3), called `primaryMission`. Note that whenever the '...' notation is used (as in `Plan doStuff`), it refers to omitted, *don't care* code that is not shown for the sake of brevity. The plan instance `primaryMission` acts just like any primitive plan instance now, but represents the aggregate of multiple plan instances.

4.2.3.4 The Sortie Plan Instance and Children Plan Instances

MPFL internally creates a single plan instance of the last user-defined plan in the MSL program known as the ***sortie plan instance***[3]. The ability to create plan instances of user-defined plans results in the creation of a tree structure, which internally within MPFL, is known as the ***plan instance tree*** with a root node of the *sortie* instance. In the case of Example 4, think of it as having two additional imaginary lines of code after the last user-defined plan:

```
ExecutePlan sortie(UserPlanName = master)
Do(sortie)
```

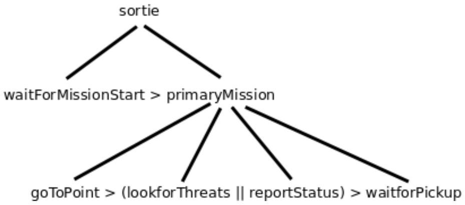

sortie
waitForMissionStart > primaryMission
goToPoint > (lookforThreats || reportStatus) > waitforPickup

Fig. 4.1 Depiction of the plan instance tree for Example 4

Figure 4.1 graphically depicts the tree, also encoding the planning operators. The plan instance tree contains within it all plan instances declared within the program. In the example, the sortie plan instance has two children `waitForMissionStart` and `primaryMission`. As `waitForMissionStart` is a primitive plan instance (`Loiter`), it is a leaf node in the plan instance tree. However, `primaryMission` is an instance of the `doStuff` user-defined plan, meaning that it is a non-leaf. As intuition suggests, `primaryMission` has four leaf children: `goToPoint, lookforThreats, reportStatus,` and `waitForPickup`.

---

[3] The term *sortie* comes from military terminology and refers to the deployment of a single military unit. In the MSL, the sortie plan instance is a representation of what the robot is to do during its deployment – hence the name.

4.2.3.5 Multiple Instances of User-Defined Plans

User-defined plan types behave just like primitive plan types. Multiple plan instances of these types can be created both within and across different plans. In Example 4, it would be perfectly acceptable to create more than one instance of a `doStuff` user-defined plan with the caveat that the identifiers have to be unique within the enclosing user-defined plan declaration.

4.2.3.6 Circular Dependencies

In order to create an instance of a user-defined plan, the MSL requires the user-defined plan to already be declared. This means it is not possible to create a circular dependency.

4.2.4 Run-time Value Lookups

Once MPFL commences, it is not possible to modify the mission specification written in the MSL. This is extremely limiting as many of the parameters used within various type constructors may not be known until runtime. To get around this limitation, the MSL contains a set of built-in functions that can lookup external values from a *knowledge base* (explained more in the next chapter) that can be substituted as needed during runtime.

For example, we can modify Example 1 so that the latitude and longitude are dynamically loaded from the knowledge base:

```
UseAutopilot goToPoint(Destination = GeoPosition(
                                      Lat = Degrees(LookupFloat(myLatitude)),
                                      Lon = Degrees(LookupFloat(myLongitude)),
                                      Depth = Meters(10.0)))
```

Example 5 - A `UseAutopilot` plan instance with dynamic lookup

In this example, the `Degrees` constructors corresponding to the latitude and longitude takes a `LookupFloat` value instead of a hard-coded number. Internally within MPFL, whenever the value is queried for `goToPoint`'s destination latitude or longitude, it will be looked up. The parameters within parentheses following `LookupFloat` refers to a ***key*** (which is simply a *String* value) that is used by MPFL to find the appropriate value. The user must be aware of what keys are available while writing their specification and their corresponding types. There are currently only four different `Lookup*` calls for the most primitive types, but more will be added for all the types specified in Table 4.3. The lookup calls are described in Table 4.8.

Note that the lookup calls are typed expression so wherever a value of one of the types in Table 4.8 is needed, it can be substituted with the corresponding lookup call. For example, because the parameter represented the key for all the lookup calls is of type *String*, we can nest another `LookupString` within the original `LookupFloat` call we issued to determine the key as shown in Example 6:

MSL Type	Lookup Callname
Boolean	`LookupBoolean`
Integer	`LookupInteger`
Float	`LookupFloat`
String	`LookupString`

Table 4.8 Lookup Calls in MPFL MSL

```
UseAutopilot goToPoint(Destination = GeoPosition(
                                      Lat = Degrees(LookupFloat(LookupString(foobar))),
                                      Lon = Degrees(LookupFloat(myLongitude)),
                                      Depth = Meters(10.0)))
```

Example 6 - Nested dynamic lookup calls

4.2.4.1 An Important Note on the Safety of Lookup Calls

Even though lookup calls are type safe, their use can be dangerous for the following reasons:

- Since the lookup is done in a lazy fashion, it is not evaluated until needed. This means range checking cannot be performed on lookup values the way it can on literal values. For example, a latitude of 800 does not make sense as the value must fall between -90 and 90. The MPFL compiler verifies all literal values so such an error would not occur if a latitude was hardcoded.
- Lookup calls do not necessarily refer to constant values, rather they typically refer to dynamic values that change with time (as is their purpose). Hence the behavior of the AUV maybe more difficult to predict.
- If the key is not defined in the knowledge base (described in the next chapter), it will cause the MPFL runtime to throw an exception indicating error.

All-in-all, this issue really cannot be avoided in any [interesting] computer program written in any language. The user is just advised to be careful when using lookup calls, and to use them sparingly as possible.

4.2.5 Conditional Expressions

Besides the planning operators, users can use ***conditional expressions*** within their Do Expression. This gives users more control over how the tasks they want their AUV to accomplish are executed, in contrast to having the MPFL runtime make the decisions. The conditional expression looks similar to the *if-then-else-endif* structure found across many programming languages and has type *PlanExpression* so that it can be used within a Do Expression. We can expand Table 4.6 with an additional entry which is conveyed in Table 4.9.

Value	Type	Syntax
`If Then Else Endif`	*PlanExpression*	`if(<`*`Boolean`*`>) then (<`*`PlanExpression`*`>) else (<`*`PlanExpression`*`>) endif`

Table 4.9 Conditional expressions are of type TypePlanExpression

The subexpression for each branch can be any value of type PlanExpression including another conditional expression or values of type *PlanExpression* in Table 4.6. Unlike conditional expressions in most languages, the MSL conditional expression requires the definition of both branches (i.e. each conditional expression has an `else` block).

4.2.5.1 Meaning of Conditional Expression

The purpose of conditional expressions is to allow users to make decisions about the makeup of their Do Expression. If the condition in the conditional expression is true, the expression reduces to *PlanExpression* in the first branch, otherwise it reduces to the expression in the second (i.e. `else`) branch. The unused branch is thrown away.

Example 7 below gives us a simple user-defined plan with a conditional expression:

```
Plan
(
    Transit a(...)
    Search b(...)
    PhoneHome c(...)
    Loiter d(...)
    Loiter e(...)

    Do(a > (if(LookupFloat(EnergyLeftInHours) > 20)(b || c) else (d ^ e) endif))
)
```

Example 7 - Example of a conditional expression within the Do expression

In Example 7, we have the plan instance `a` which refers to a `Transit` plan instance on the left side of a serial (>) operator with a conditional expression on the right-hand side. The condition determines if a looked up value of type `float` with key `EnergyLeftInHours` is greater than 20. If during runtime the condition is true, the entire expression reduces to:

`Do(a > (b || c))`

Likewise if the condition is false, the result is:

`Do(a > (d ^ e))`

4.2.5.2 Fluctuation of Condition

Recall that lookup values are not necessarily constant and are lazily evaluated. In the case that the condition in the expression changes between compilations, it may result in a changing resulting Do Expression. In the case of Example 7, it may turn out that in the beginning, the value of `LookupFloat(EnergyLeftInHours)` is indeed greater than 20 causing `b` and c to occur in parallel after `a` is completed. However, it could turn out that while `b||c` are being executed, the value of the lookup is less than or equal to 20 (likely due to lack of residual energy as implied by the key). This causes `d^e` to pop into the mix, even though `b` and `c` have partially been attempted or perhaps even been completed already. The user of MPFL must be wary of this fluctuating condition phenomenon and should plan for that in their mission specification.

4.2.5.3 Defining the Condition

As the condition within the conditional expression can be any expression of type *Boolean*, any value of this type can be used for the condition. We saw in Table 4.3 that values of the *Boolean* type can be defined in several ways. Table 4.10 shows the different ways one can construct the condition of a conditional expression. Users can compare values for equality, inequality, greater than, greater than and equal, less than, and less than or equal to for the four core primitive types (*Boolean*, *String*, *Integer*, *Float*).

Value	**Type**	**Syntax**
$\overline{boolean}$	*Boolean*	`true, false`
LookupBoolean	*Boolean*	`LookupBoolean(`*String*`)`
>=	*Boolean*	$<X> \; >= \; <X>$
>	*Boolean*	$<X> \; > \; <X>$
==	*Boolean*	$<X> \; == \; <X>$
! =	*Boolean*	$<X> \; != \; <X>$
<=	*Boolean*	$<X> \; <= \; <X>$
<	*Boolean*	$<X> \; < \; <X>$

Table 4.10 Boolean Type Values (note that X can be values of types *Integer*, *String*, *Float*, or *Boolean)*

4.2.5.4 Pitfalls of Conditional Expressions

Even though conditional expressions are quite powerful and useful, their use should be avoided. Conditional expressions are imperative (in contrast to declarative) constructs that move responsibility away from the MPFL runtime into the hands of the user. However, this deviates from the intention of the MSL. The goal of the MSL and

MPFL runtime is to allow users to not focus too much on *how* the tasks are to be executed, but rather *what* tasks need to be executed. Regardless, conditional expressions cannot be avoided all-together as fine-grained control maybe required, but keep in mind that they might not be necessary in order to accomplish the robot's goal. In the case of Example 7, rather than using a conditional, an alternative might be to replace it instead with with an exclusive or (ˆ):

```
Do(a > ((b || c) ^ (d ^ e)))
```

MPFL would *figure out* how to best execute the mission, attempting to accomplish `b||c`, but resorting to `(d^e)` only if necessary. However, the user does not have fine-grained control over how that exclusive or (i.e. `d ^ e`) decision is made.

4.2.6 Declaring and Binding Constraints

The MPFL language is a constraint-based language. Users define problems as a series of declarative tasks that are bound by certain rules, namely the planning operators specified in the Do Expression, as well as the problem definition that is passed in the constructor of each plan instance. The language runtime is responsible for solving the problems based on these constraints. The MPFL language takes this a step further by creating stand-alone constraints that can be bound to plan instances referenced in the Do Expression. To illustrate, here is an example:

```
Plan anExample
(
    UseSonar a(...)
    Search b(...)
    Loiter c(...)

    TimeConstraint timeLimit(ClockTime(Days=0, Time=19::00::00) <= StartTime <=
                       ClockTime(Days=0, Time=20::00::00),
                       ClockTime(Days=1, Time=19::00::00) <= EndTime <=
                       ClockTime(Days=1, Time=21::30::00))

    Do(a & (b with timeLimit) > c)
)
```

Example 8 - Binding a constraint within the Do Expression

In this example, we construct a `TimeConstraint` value with identifier `timeLimit`. This value has type *Constraint* and can now be bound to any expression of type *PlanExpression* within the Do Expression. In the example, the operator `with` is utilized to bind `timeLimit` to the plan instance `b`. Table 4.11 extends the type system of the MSL and defines the different values of type *Constraint* as well as defines the `with` operator.

The syntax of a user-defined plan declaration can be enhanced thus:

Value	**Type**	**Syntax**
`TimeConstraint`	*Constraint*	TimeConstraint <$\overline{string}$>(<*Time*> <= StartTime <= <*Time*>, <*Time*> <= EndTime <= <*Time*>)
`PowerConstraint`	*Constraint*	PowerConstraint <$\overline{string}$>(*String*)
`operator with`	*PlanExpression*	<*PlanExpression*> with <*Constraint Identifier*>

Table 4.11 Constraint Type Values

```
Plan <plan identifier>
(
    <plan instance declaration 1>
    <plan instance declaration 2>
    ...
    <plan instance declaration n>

    <constraint declaration 1>
    <constraint declaration 2>
    ...
    <constraint declaration n>

    Do(<plan expression>)
)
```

4.2.6.1 Meaning of a Constraint

When a plan expression is bound to a constraint via the `with` operator, it means that the MPFL runtime must plan to carry out the task represented by that plan instance under the limitations of that constraint. The constraint in Example 8 is a ***time constraint***. The time constraint in Example 8 tells us that plan instance `b` must start sometime between 7 PM and 8 PM of the current day and complete between 7 PM to 9:30 PM of the following day.

The other type of *Constraint* in the language is a ***power constraint*** which enforces the amount of energy that can be used by a plan instance to accomplish it's task, along with the maximum energy usage rate (i.e. power) .

4.2.6.2 Binding to an Entire Subexpression

The `with` operator, as described in Table 4.11, allows binding a constraint to any subexpression of type *PlanExpression*. Like all other operators in the language, it is left-associative. For example, the Do Expression from Example 8 can be modified to bind the constraint to both `a` and b in the expression:

```
Do(((a & b) with timeLimit)) > c)
```

Equivalently one can just use the `with` operator twice and achieve the same result:

```
Do((a with timeLimit) & (b with timeLimit) > c)
```

4.2.6.3 Binding Multiple Constraints and Constraint Intersection

One can bind as many constraints to a *PlanExpression* as they would like. With multiple constraints, MPFL takes the *intersection* of all constraints. A modified version of Example 8 that has more than one `TimeConstraint` is as follows:

```
Plan anExample
(
    UseSonar a(...)
    Search b(...)
    Loiter c(...)

TimeConstraint timeLimit1(ClockTime(Days=0, Time=19::00::00) <= StartTime <=
                          ClockTime(Days=0, Time=20::00::00),
                          ClockTime(Days=1, Time=19::00::00) <= EndTime <=
                          ClockTime(Days=1, Time=21::30::00))

TimeConstraint timeLimit2(ClockTime(Days=0, Time=15::00::00) <= StartTime <=
                          ClockTime(Days=0, Time=19::30::00),
                          ClockTime(Days=1, Time=19::30::00) <= EndTime <=
                          ClockTime(Days=1, Time=22::00::00))

    Do(((a & b) with timeLimit1) > c) with timeLimit2)
)
```

Example 9 - Binding multiple constraints

In this case, `a` and `b` have both `timeLimit1` and `timeLimit2` bound to them. Plan instance `c` has only `timeLimit2` bound to it. The intersection of `timeLimit1` and `timeLimit2` ends up in a new time constraint with a start time between 7 pm and 9:30 pm and an end time between 7:30 pm and 9:30 pm the following day. In the case the intersection of time constraints is empty, it means the constraints do not overlap and the task to which the constraint is bound will be infeasible (this is discussed in later sections). For the case of power constraints, the intersection is simply the minimum values for the maximum energy allowed and maximum power level.

4.2.6.4 Constraints are Hierarchical

When constraints are bound to instances of user-defined plans (i.e. instances that are `ExecutePlan` values), those constraints are implicitly bound to all its descendants in the plan instance tree. This is one of the interesting features of the MSL; allowing constraints to be hierarchically arranged. This is a natural way to reason about resource allocation during planning; allocate large chunks of resources for the high-level goals which are then broken into smaller chunks to be distributed to subgoals. It is likely when creating the mission specification that higher-level (i.e. coarse) level plan instances that are closer to the *sortie* plan instance will have the larger portion of resources than those divvied up amongst the children plan instances. The children in turn can create tighter constraints for their children and so on.

4.2.7 Handling Errors: Infeasibilities and Conflicts

One issue that arises is what MPFL does when it cannot achieve a goal or set of goals defined within an MSL program. In the MPFL world, there are two types of problems that can occur during planning respectively known as ***infeasibility errors*** and ***conflict errors***.

4.2.7.1 Infeasibility Errors

Infeasibility errors occur when a task the user requested is physically impossible for the AUV to achieve. For example, if one was to create a `UseAutopilot` plan instance requesting the vehicle move from the Chesapeake Bay to Hawaii with a time constraint of an hour, it would be an impossible task. This task could be impossible for several reasons. Obviously time is one of the issues. Another reason it could be infeasible is that the vehicle does not have enough power to accomplish the mission.

4.2.7.2 Conflict Errors

Conflict errors, in contrast to infeasibility errors, come into play when the AUV cannot do two or more requested tasks at the same time due to a resource scheduling conflict between them. For example, the vehicle could have onboard both a sonar and an acoustic modem, both require using a shared communication medium (the water). If each device utilizes the same frequency band, it may not be possible to use them both at the same time, causing a conflict error. At this point, a choice would have to be made on which device gets to be used and when.

4.2.7.3 Graceful Degradation and Exception Handling

When humans encounter problems achieving our own infeasibilities and conflicts, we make decisions to keep progressing even if it is in a suboptimal manner. This may mean giving up on some of our goals for the time being so we can achieve others. Sometimes not achieving one or more goals in your set of goals can be a show stopper, causing one to throw away the remainder of the goals. Essentially, humans continue to persevere even though they fail in life. With robotics, it is no different. One of the problems with design of autonomous robots, or any complex system for that matter, is dealing with failures of subcomponents. Well-designed systems can keep functioning, albeit in a potentially degraded mode, in the event of failures. This is known as ***graceful degradation***. For example, just because a sensor fails does not mean a robot cannot still continue its mission utilizing the remainder of its sensors and accounting for the amount of error introduced by losing said sensor.

In the MSL we can support graceful degradation by replicating the decision making capability humans use when having to dealing with their own infeasibilities and conflicts. In many modern programming languages, the language provides the ability to handle errors via ***exception handling***. Exception handling allows programmers at any point to signal an exceptional situation (known as *raising* or *throwing* an exception). Programmers can catch errors by associating an ***exception handler*** to a block of code that may raise an exception somewhere within its thread of execution. If an exception is raised, the call stack is unwound until an exception handler in scope that can handle the exception is found. Once the exception is handled, code execution returns to the first line of code after the handler. Exceptions are useful because:

- They are typically typed values in the language that can store information about the exceptional situation
- They can be handled in a hierarchical fashion at any level along the call stack. Sometimes it makes more sense to handle the error nearer to the point where the exception was raised and sometimes it makes more sense to deal with at a higher level.
- They separate error handling code from the application logic of the program. Not only does this make the code cleaner, but it is easy to add in new exceptions and exception handlers as the code evolves without having to modify the application logic.

4.2.7.4 Infeasible and Conflict Handlers

In the MSL, a form of exception handling is used to deal with infeasibilities and conflicts. For each user-defined plan, one has the ability to specify up to two handlers, one being an ***infeasibility handler*** and the other being a ***conflict handler***. When infeasibilities and conflicts occur within/between plan instances, the MPFL runtime jumps to the nearest handler and attempts to resolve the issue.

To illustrate the use of infeasibility and conflict handling in the MSL, observe the following example:

```
Plan meow
(
   Search a(...)
   Transit b(...)
   UseSonar c(...)
   PhoneHome d(...)
   UseModem e(...)

   Do(b > (a & c) || (d ^ e))

   OnInfeasible
   (
      Case(a) (Disable(a))
      Case(b) (Disable(b))
      Case(c)
      (
         if (LookupBoolean(GiveUpOnC)) then
            Retract(c)
         else
            Disable(c)
         endif
      )
      Case(d) (Retract(d))
      Case(e) (Disable(e))
   )

   OnConflict
   (
      Case(a,c)(Disable(c))
      Case(a,b,c,d,e) (Disable(a,d))
      Case(b,d,e)(Retract(e))
   )
)
```

Example 10 - A user-defined plan specifying both an infeasibility handler and conflict handler

Example 10 illustrates the use of both an infeasibility handler and a conflict handler. When MPFL cannot generate schedules that would lead to the completion of some task due to a problem, the runtime generates either an infeasibility or conflict exception depending on the type of problem that occurred. This event triggers MPFL to resolve the infeasibility or conflict via the nearest exception handler. Infeasibility exceptions are handled by the `OnInfeasible` handler and conflict exceptions are handled by the `OnConflict` handler. Each handler consists of a set of ***case*** expressions. Each expression has two parts: a ***signature*** indicating which plan instance(s) caused the problem (enclosed in parentheses after the keyword `Case`) and then a ***handler expression*** which resolves the problem.

The signature indicates which plan instances caused the problem. In the case of the infeasibility handler, each case signature can only have a single entry as each infeasibility maps to a single plan instance. In contrast conflicts have a list of plan instances in their signature indicating which tasks have a conflict between them. Within the handler expression, the user must decide to either ***disable*** or ***retract*** one of the plan instances referenced in the case signature. Disabling means to *temporarily disable* the task for the current MPFL compilation cycle whereas retract means *permanently remove* the task from consideration. The MPFL compiler forces the user to disable or retract at least one plan instance referenced in the signature (in the case of infeasibilities, there is only one option) using the `Disable` and `Retract` functions respectively. Users can also use conditional expressions to make their decision just like in the Do Expression. Table 4.12 describes all the different valid handler expressions that can be used in an infeasibility or conflict handler.

Value	**Type**	**Syntax**
`If Then Else Endif`	*HandlerExpression*	`if` <*Boolean*> `then` <*HandlerExpression*> `else` <*HandlerExpression*>
`Disable`	*HandlerExpression*	`Disable(`<*HandlerExpression*>`)`
`Retract`	*HandlerExpression*	`Retract(`<*HandlerExpression*>`)`
plan instance chain	*HandlerExpression*	<$\overline{String}$>`->...->`<$\overline{String}$>

Table 4.12 MSL handler expression types

We can now finalize the syntax of a user-defined plan declaration:

```
Plan <plan identifier>
(
    <plan instance declaration 1>
    <plan instance declaration 2>
    ...
    <plan instance declaration n>

    <constraint declaration 1>
    <constraint declaration 2>
    ...
    <constraint declaration n>

    Do(<plan expression>)

    <infeasible handler>
    <conflict handler>
)
```

4.2.7.5 Handlers Are Hierarchical

Just like exceptions in most languages, when an exception is raised/thrown, the execution path jumps to the nearest handler that can handle that particular type of exception. If the nearest exception handler cannot handle the case, it jumps up to the next one in the plan that contains the conflicting/infeasible instance(s) parent instance declaration. Eventually, if no handlers are found in the sortie plan, the exception has not been handled by the user and results in an unhandled exception error. This causes the MPFL runtime to exit with an error. It is the responsibility of the user to handle all infeasibility and conflict cases. The MPFL compiler will warn the user when cases are unhandled during compile time. The following example illustrates the hierarchical nature of MSL handlers:

```
Plan makeNoise
(
   UseModem a(...)
   UseSonar b(...)

   Do(a & b)

   OnInfeasible
   (
      Case(a)(Retract(a))
   )
)

Plan distractTarget
(
   Transit x(...)
   ExecutePlan y(UserPlanName = makeNoise)

   Do(x > y)

   OnInfeasible
   (
      Case(x)(Disable(x))
      Case(y)(Disable(y))
  )
)
```

Example 11 - Hierarchical exception handling

In Example 11, the user-defined plan `makeNoise`, plan instance `b` has no infeasible handler. However, as `b` is a child of the instance `y` declared within `distractTarget`, when the case is unmatched in `makeNoise`, it will jump to the handler in `distractTarget` for `y`. When `y` is disabled, the child plan instance `b` (and children it may have) will be disabled. The ability to handle errors at any level allows MSL programmers to handle errors within the scope that is most appropriate.

4.2.7.6 Case Signature Matching and Plan Instance Chains

To take hierarchical exception handling further, MSL handler case signatures do not necessarily have to reference plan instances within the same user-defined plan as the infeasibility or conflict handler(s), but also any of those plan instances' descendants (i.e. children, grandchildren, and so forth).

In each case expression, the case signature is actually not matching on the name of a plan instance, but rather it is matching against its ***plan instance chain***. Every plan instance in the MPFL runtime can be uniquely identified as a sequence of plan instance identifiers starting from the root instance in the plan instance tree, the *sortie* instance. The notation for this involves sequentially naming the plan instances forming the chain with arrows (->) delimiting the instances. Take for example the following code:

```
Plan pasta
(
   PhoneHome pasta1(...)
   Transit   pasta2(...)

   Do(...)
)

Plan bread
(
   ExecutePlan bread1(UserPlanName = pasta)
   ExecutePlan bread2(UserPlanName = pasta)
   Search      bread3(...)

   Do(...)
)

Plan rice
(
   UseAutopilot rice1(...)
   Loiter       rice2(...)
   ExecutePlan  rice3(UserPlanName = pasta)
   ExecutePlan  rice4(UserPlanName = bread)

   Do(...)
)
```

Example 12 - Understanding plan instance chains

Example 12 shows multiple instances of the same user-defined plan. By having multiple instances of the user-defined plans `pasta` and `bread`, there are multiple instances of the plan instances declared within them. However, each plan instance can be uniquely identified using its plan instance chain. For example, there are multiple instances of the plan instance `pasta1` declared within the plan `pasta`:

```
sortie->rice3->pasta1
sortie->rice4->bread1->pasta1
sortie->rice4->bread2->pasta1
```

The plan instance chain is analogous to the *process identifier* in operating systems such as Unix where each process has a unique integer identifier. Even though there may be multiple processes of the same program, each one is unique with its own identifier.

What makes plan instance chains useful in the MSL is that they can be used in the case signatures of handlers. This allows for fine grained error handling control by allowing users to write handlers for problems caused by descendants of plan instances declared within their user-defined plans. The intuitive reason for this feature is that it sometimes may make sense to handle an error for a child at a higher level as there is more contextual information there. To illustrate, Example 13 modifies Example 12 to utilize the plan instance chain notation:

```
Plan makeNoise
(
   UseModem a(...)
   UseSonar b(...)

   Do(a & b)

   OnInfeasible
   (
      Case(a)(Retract(a))
   )
)

Plan distractTarget
(
   Transit x(...)
   ExecutePlan y(UserPlanName = makeNoise)

   Do(x > y)

   OnInfeasible
   (
      Case(x)(Disable(x))
      Case(y->b)(Disable(y->b)
      Case(y)(Disable(y))
   )
   OnConflict
   (
      Case(y->a, y->b)
      (
         if (LookupFloat(energyLeftHours) > 500) then
         (
            Disable(y->b)
         )
         else
         (
            Disable(y->a)
         )
      Case(x,y)(Disable(y))
   )
)
```

Example 13 - Plan instance chain used in handlers

In Example 13, the infeasibility and conflict handlers utilize plan instance chains to directly reference child instances `y→a` and `y→b`. In the infeasibility case, if `b` gives problems, it will end up triggering the infeasibility handler in `distractTarget` with signature `y→b` causing it to be disabled. The last case in the same block (`Case(y)(Disable(y))`, is actually unreachable. If `y→a` had an infeasibility, it would be handled directly in `makeNoise`, and the latter case with `y→b` is handled in the previous case. This brings up an interesting point about how case signature pattern matching works. If in Example 12 the order of the infeasible cases matters; if `Case(y→b)` came after `Case(y)` in `distractTarget`'s infeasibility handler, it would actually fire first. This is because `y` and `y→b` are both valid matches for the case as they are in the same lineage. Matching is therefore based on the first-matching signature rather than the one that matches most closely in the structural sense.

Example 13 also demonstrates how conflicts can be handled in the same way by adding a conflict handler to `distractTarget`. In this handler, if `y→a` and `y→b` were to conflict, it will be handled there. However, with conflicts, the chains cannot be any arbitrary tuple of plan instance chains; chains must be exactly the same except for the last hop. The reason for this is because of a physical limitation of the MPFL compiler and the difficulty in having the ability to compare arbitrary plan instances for conflict.

4.3 Segue: Making it Work

This concludes the chapter on the MPFL Mission Specification Language. Hopefully, it is evident how using a domain-specific programming language can be a powerful means of control of an AUV. The language allows one to focus on the task at hand: planning missions for AUVs. The syntax is simple, consistent, and can only be used to express planning problems. The declarative nature of the language allows AUV operators to specify *what* they want to do and not *how* they want to do it. The language brings with it strong static typing to prevent errors, a type system with dimensions checking to prevent unit errors, the ability to handle system failures in a graceful and hierarchical way through exceptions, and a uniform and intuitive way of reasoning about the problem.

Those that work in robotics may think such a language is a great idea, but may seem difficult or even infeasible to build. However, it is possible and a straightforward approach is described in the next chapter. By using a simple *model of planning*, one can implement an MPFL compiler that obeys the semantics informally described in this last chapter. The next chapter will describe an implementation of MPFL and how it relates to concepts in Chapter 2. Understanding the MSL is important to understanding how MPFL works and why the language was described first. The next chapter demonstrates that the MSL is not only the external interface to a robot that uses MPFL, but also the foundation of the MPFL compiler/runtime's internal implementation.

MPFL is not a standalone compiler; it depends on plugins that define some of the primitive capabilities of the robot. These plugins are defined through an simple API which is tied closely to the implementation of the core MPFL compiler/runtime. The next chapter will describe the API and describe how autonomy software developers can harness MPFL in their autonomous robots.

Chapter 5

Understanding and Utilizing MPFL

5.1 Overview

The previous chapter depicted how the MSL allows a user to describe complex missions consisting of multiple goals and constraints in a logical manner. The MPFL compiler ensures that either each goal that is attempted meets all constraints bound to it or informs the user when constraints will be violated while giving them an opportunity to handle the problem through exception handling. Though implementing a compiler for such a language may seem difficult, the approach utilized to implement a prototype compiler was quite straightforward and simple to reason about. This chapter describes how the prototype compiler and API is implemented and how one utilizes it in their own autonomous robot.

5.2 How MPFL is Meant to Be Used

As mentioned in Chapter 3, MPFL is not quite like compilers for typical high-level languages where the compiler is an application that takes source files as input (often in the form command line arguments) and then generates corresponding object files. Rather MPFL is a library that a client application must link against and invoke from within their application utilizing calls provided in the API (Figure 5.1). The application could encompass all the autonomy software onboard a robot or represent one module/agent in a larger system. The prototype reference implementation of MPFL is written in the *OCaml* programming language and for the time being MPFL client applications must utilize an OCaml API until other bindings are written. In addition to be invoked from a client application, the MPFL compiler/runtime requires a set of plugins that define the primitive scheduling abilities as well as hooks to perception/knowledge base information to be fed to it. The way MPFL is utilized within a client application is described by the following algorithm:

1. Initialize the MPFL compiler/runtime passing in path of the file containing the mission specification written in the MSL along with with a set of plug-ins via the function `initialize_mpfl`.
2. Invoke the MPFL compiler from the client code which returns a set of *schedules* via the function `build_schedules`.
3. Use the schedules (i.e. parse and process them) to control actuators and subsystems of the robot in order to accomplish goals specified in the mission specification.
4. Goto 2.

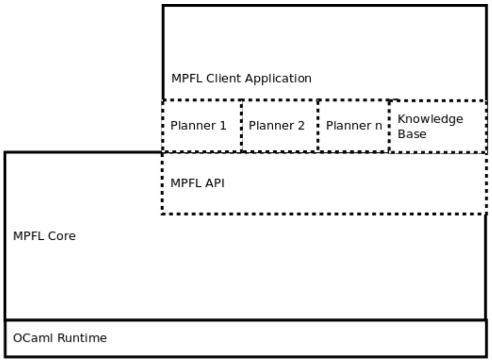


Fig. 5.1 Relationship between MPFL core library and client application

Unlike a typical compiler, the MPFL compiler is meant to constantly be reinvoked, hence the need for the last step in the algorithm. As a robot performs its mission, the environment around it may change causing perception information to change. For example, the compiler may generate a schedule for the navigational subsystem that assumes a clear, obstruction-free path. However as the mission progresses, sensors discover an obstacle is in the way. This means the current set of schedules is no longer valid and replanning needs to occur. Replanning is carried out by simply reinvoking the compiler and generating a new set of schedules. Due to this, the client application must constantly reinvoke the compiler (as in step 2) and issue new commands to its subsystems. One way to do this is as described in the above algorithm; reinvoke the compiler in an infinite loop ensuring the freshest set of schedules available. A smarter, less computationally-intensive MPFL client would only reinvoke the compiler when environmental information has changed to some significant degree.

5.3 The Simplified MPFL Compiler/Runtime Engine

The mission specification just describes what the robot is to do. It is input by the user in the form of a text file. It is up to the compiler to figure out either how to accomplish the tasks in the specification or inform the user when something cannot be done due to task infeasibilities/conflicts. The compiler's job is to take a high-level specification in the MSL and translate it to a series of low-level schedules (Figure 3.1) Schedules are simply tables of timestamped commands for the low-level subsystems and actuators, such as commands for the drive system or activating a payload such as a camera or robotic arm.

This section describes the core stages of the MPFL compiler while omitting some details for the sake of simplicity. The remaining details will be discussed in the next section. The MPFL compiler is broken into the following stages that are also illustrated in Figure 5.2:

- Parser
- Plan Instance Tree (PIT) Builder

- Lifetime State Transition (LST) Evaluator
- Planner Invocation (PI) Evaluator

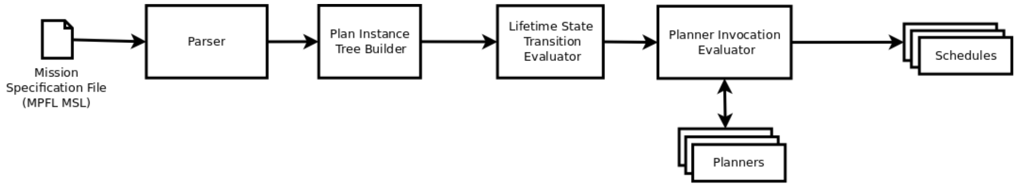


Fig. 5.2 Simplified MPFL Engine

5.4 Parser

The ***parser*** for most programming language interpreters/compilers has the responsibility of reading source code and generating an internal representation of the code for subsequent interpreter/compiler stages. In other words, it transforms code specified with a ***concrete syntax*** (i.e. the MSL) into an abstract representation, such as an *abstract syntax tree* which a computer program (the compiler/runtime) can process more easily. The MPFL parser takes the specification written in the MSL and generates a simple abstract syntax tree encompassing the entire specification. If there are syntax errors in the specification, the parser emits a description of the errors and their respective line numbers to the console resulting in the termination of the MPFL compiler/runtime.

The grammar of the language is an LALR(1) grammar (a subset of *context-free grammars*) for which a *shift-reduce* parser was generated utilizing the OCaml equivalents of the popular *lex* lexer generator and *yacc* parser generator (called *ocamllex* and *ocamlyacc* respectively).

5.4.1 Grammar Specification

The complete grammar specification is defined as follows using ***Backus-Naur Form (BNF)*** notation.

```
(** Main Program **)
Program            := PlanDeclarations
PlanDeclarations   := PlanDeclarations PlanDeclaration
                    | PlanDeclaration
PlanDeclaration    := ‘Plan’ <string> ‘(’ PlanInstDeclarations
                       ConstraintDeclarations DoExpDeclaration ‘)’
                     | ‘Plan’ <string> ‘(’ PlanInstDeclarations
                      ConstraintDeclarations DoExpDeclaration
                      InfeasibleHandler ‘)’
                    | ‘Plan’ <string> ‘(’ PlanInstDeclarations
                      ConstraintDeclarations DoExpDeclaration
                      ConflictHandler ‘)’
                    | ‘Plan’ <string> ‘(’ PlanInstDeclarations
                      ConstraintDeclarations DoExpDeclaration
                      InfeasibleHandler ConflictHandler ‘)’

(** Error Handlers **)
InfeasibleHandler := ‘OnInfeasible’ ‘(’ InfeasibleCases‘)’
ConflictHandler   := ‘OnConflict’ ‘(’ ConflictCases‘)’

InfeasibleCases   := InfeasibleCases+
ConflictCases     := ConflictCases+

InfeasibleCase    := ‘Case’ ‘(’ PlanInstChain ‘)’ ‘(’ HandlerExp ‘)’
InfeasibleCase    := ‘Case’ ‘(’ PlanInstChains ‘)’ ‘(’ HandlerExp ‘)’

PlanInstChains    := PlanInstChain | PlanInstChain ‘,’ PlanInstChains
PlanInstChain     := PlanInstChain ‘->’ <string> | <string>

HandlerExp        := ‘Disable’ ‘(’ PlanInstChains ‘)’
                   | ‘Retract’ ‘(’ PlanInstChains ‘)’
                   | ‘if’ ‘(’ MPFLBool ‘)’ then
                   ‘(’ HandlerExp ‘)’ else ‘(’ HandlerExp ‘)’

(** Primitive Types **)

MPFLString        := ‘LookupString’ ‘(’ MPFLString ‘)’ | <string>
MPFLInteger       := MPFLInteger ‘+’ MPFLIntegerTerm
                   | MPFLInteger ‘-’ MPFLIntegerTerm
                   | MPFLInteger ‘*’ MPFLIntegerTerm
                   | MPFLInteger ‘/’ MPFLIntegerTerm
                   | ‘(’ MPFLInteger ‘)’
                   | MPFLIntegerTerm
MPFLIntegerTerm   := ‘LookupInteger’ ‘(’ MPFLString ‘)’ | <integer>

MPFLFloat       := MPFLFloat ‘+’ MPFLFloatTerm
                 | MPFLFloat ‘-’ MPFLFloatTerm
                 | MPFLFloat ‘*’ MPFLFloatTerm
                 | MPFLFloat ‘/’ MPFLFloatTerm
                 | ‘(’ MPFLFloat ‘)’
                 | MPFLFloatTerm
MPFLFloatTerm   := ‘LookupFloat’ ‘(’ MPFLString ‘)’ | <float>

MPFLBool        := ‘LookupBool’ ‘(’ MPFLString ‘)’
```

```
                  | <bool>
                  | MPFLInteger ‘>=’ MPFLInteger
                  | MPFLInteger ‘>’ MPFLInteger
                  | MPFLInteger ‘==’ MPFLInteger
                  | MPFLInteger ‘<=’ MPFLInteger
                  | MPFLInteger ‘<’ MPFLInteger
                  | MPFLFloat ‘>=’ MPFLFloat
                  | MPFLFloat ‘>’ MPFLFloat
                  | MPFLFloat ‘==’ MPFLFloat
                  | MPFLFloat ‘<=’ MPFLFloat
                  | MPFLFloat ‘<’ MPFLFloat
                  | MPFLString ‘==’ MPFLString

(** Plan Instance Declarations **)
PlanInstDeclarations := PlanInstDeclarations PlanInstDeclaration
                      | PlanInstDeclaration

PlanInstDeclaration := ‘ExecutePlan’ <string> ‘(’ ExecutePlanParams ‘)’
                     | ‘Loiter’ <string> ‘(’ LoiterParams ‘)’
                     | ‘PhoneHome’ <string> ‘(’ PhoneHomeParams ‘)’
                     | ‘Search’ <string> ‘(’ SearchParams ‘)’
                     | ‘Transit’ <string> ‘(’ TransitParams ‘)’
                     | ‘UseAcoustic’ <string> ‘(’ UseAcousticParams ‘)’
                     | ‘UseAutopilot’ <string> ‘(’ UseAutopilotParams ‘)’
                     | ‘UseModem’ <string> ‘(’ UseModemParams ‘)’
                     | ‘UseSonar’ <string> ‘(’ UseSonarParams ‘)’

ExecutePlanParams  :=  ‘UserPlanName’ ‘=’ <string>

LoiterParams       := ‘LoiterPosition’ ‘=’ Position

PhoneHomeParams    := ‘ModemName’ ‘=’ MPFLString ‘,’ ‘PhoneHomeRate’
                      ‘=’ Frequency

SearchParams       := ‘SonarName’ ‘=’ MPFLString ‘,’ ‘SearchArea’
                      ‘=’ Area ‘,’  ‘LaneWidthToken’ ‘=’ Length

TransitParams      := ‘Waypoints’ ‘=’ Positions

UseAcousticParams  := ‘AcousticDevice’ ‘=’ MPFLString ‘,’ StartTime ‘=’ Time ‘,’
                      ‘EndTime’ ‘=’ Time ‘,’ TaskDuration ‘=’ Duration ‘,’
                      ‘MinGap’ ‘=’ Duration ‘,’ ‘MaxGap’ ‘=’ Duration

UseAutopilotParams := ‘Destination’ ‘=’ Position
```

```
UseModemParam      := ‘ModemName’ ‘=’ MPFLString ‘,’ ‘Message’ ‘=’ MPFLString

UseSonarParams     := ‘SonarName’ ‘=’ MPFLString ‘,’ ‘PingRate’ ‘=’ Frequency

(** Constraint Declarations **)
ConstraintDeclarations := ConstraintDeclaration+

ConstraintDeclaration  := ‘TimeConstraint’ <string> ‘(’ TimeConstraintParams ‘)’
                        | ‘PowerConstraint’ <string> ‘(’ PowerConstraintParams ‘)’

TimeConstraintParams   := Time ‘<=’ ‘StartTime’ ‘<=’ Time ‘,’
                          Time ‘<=’ ‘EndTime’ ‘<=’ Time

PowerConstraintParams  := ‘MaxLoad’ ‘=’ Power ‘,’ ‘MaxEnergy’ ‘=’ Energy

(** Do Expression Declaration **)

DoExpDeclaration : ‘Do’ ‘(’ PlanExp ‘)’

PlanExp := PlanExp ‘>’ PlanTerm
         | PlanExp ‘||’ PlanTerm
         | PlanExp ‘&’ PlanTerm
         | PlanExp ‘^’ PlanTerm
         | PlanExp ‘with’ PlanTerm
         | ‘if’ ‘(’ MPFLBool ‘)’ ‘then’ ‘(’ PlanExp ‘)’
           ‘else’ ‘(’ PlanExp ‘)’
         | PlanTerm

PlanTerm := ‘(’ PlanExp ‘)’ | <string>

(** Composite Primitive Types **)
Angle := ‘Degrees’ ‘(’ MPFLFloat ‘)’ | ‘Radians’ ‘(’ MPFLFloat ‘)’

Duration := ‘Seconds’ ‘(’ MPFLFloat ‘)’
          | ‘Minutes’ ‘(’ MPFLFloat ‘)’
          | ‘Hours’ ‘(’ MPFLFloat ‘)’

Length := ‘Meters’ ‘(’ MPFLFloat ‘)’ | ‘Feet’ ‘(’ MPFLFloat ‘)’
        | ‘Yards’ ‘(’ MPFLFloat ‘)’

Frequency := ‘Hertz’ ‘(’ MPFLFloat ‘)’

Power := ‘Watts’ ‘(’ MPFLFloat ‘)’
       | ‘Horsepower’ ‘(’ MPFLFloat ‘)’
```

```
Energy := ‘Joules’ ‘(’ MPFLFloat ‘)’
        | ‘KilowattHours’ ‘(’ MPFLFloat ‘)’
;

(** Positional Types **)
Positions         := Positions ‘-’ Position | Position
Position          := AbsolutePosition | RelativePosition

AbsolutePosition  := ‘GeoPosition’ ‘(’ ‘Lat’ ‘=’ Angle ‘,’
                     ‘Lon’ ‘=’ Angle ‘,’ ‘Depth’ ‘=’ Length ‘)’

CartesianPosition := ‘CartesianPosition’ ‘(’ ‘X’ ‘=’ Length ‘,’ ‘Y’
                     ‘=’ Length ‘,’ ‘Z’ ‘=’ Length ‘)’

RelativePosition  := ‘RelativePosition’ ‘(’ ‘Center’ ‘=’ AbsolutePosition
                     ‘,’ ‘Offset’ ‘=’ CartesianPosition ‘)’


(** Area Types **)
Area            := RectangularArea | CircularArea

RectangularArea := ‘RectangularArea’ ‘(’ TopLeftToken ‘=’
                   Position ‘,’ BottomRightToken ‘=’ Position ‘)’

CircularArea    := ‘CircularArea’ ‘(’ ‘Center’ ‘=’
                   Position ‘,’ RadiusToken ‘=’ Length ‘)’

(** Time Types **)
Time      := ClockTime | UnixTime
ClockTime := ‘ClockTime’ ‘(’ ‘Days’ ‘=’ MPFLInteger ‘,’
             ‘Time’ ‘=’ MPFLInteger ‘::’ MPFLInteger ‘::’ MPFLInteger ‘)’

UnixTime  := ‘UnixTime’ ‘(’ ‘UTCSeconds’ ‘=’ MPFLInteger ‘)’
```

5.4.2 Syntactic Enforcement of Types

One interesting aspect of the design of the MSL grammar was encoding a big chunk of the type system as production rules within the grammar itself. Most of the typed constructs from the last chapter are encoded as production rules in the grammar. This means that when one utilizes a value of the wrong type for a typed parameter, it results often in a *syntax error* rather than a *semantic error*. If we look at the classical compiler pipeline for a programming language, parsing is encompassed within the lexical analysis and syntactic analysis stages whereas the majority of type checking is encompassed within the subsequent semantic analysis stage. The reason MPFL can get away with this is because the syntax of the MSL is so limited compared to that of a general-purpose language: the order of parameters in constructors is fixed, plan instances must be declared before constraints, constraints must be declared before the Do Expression, the Do Expression must be declared before the handlers. This strict ordering is easy to implement in a context-free grammar. As a parser generator was used, it was easier to enforce many of the type rules during parsing rather than in the semantic analysis stage, hence the design. Regardless, the result is the same to the end user.

5.5 Plan Instance Tree (PIT) Builder

The output of the parser is a simple abstract tree representation of the mission as specified in the MSL. Rather than being encoded now as mere text, the specification is stored in the form of a traversable data structure that can be used by the next stage of MPFL compiler/runtime: the ***Plan Instance Tree (PIT) Builder***.

The PIT Builder has two responsibilities:

- Verify and enforce static semantics that parsing stage could not handle.
- Build an intermediate representation in the form of a *plan instance tree* (the same concept as from Chapter 4) used for subsequent engine stages.

5.5.1 Static Semantics

As mentioned earlier, many of the type rules of the language are enforced syntactically via the parser. However, there are other static semantics of the language that are not handled when parsing. This is because an LALR(1) grammar is not sufficiently powerful enough due to the context sensitivity of the remaining static semantics:

- *Reference scoping rules* - In the MSL, references are used for values in the language, such as the names of plan instances and constraints within a Do Expression, the plan instance chains in infeasibility and conflict handler cases, and the names of user-defined plans referenced in the constructors of `ExecutePlan` plan instances. The PIT builder checks the validity of references by makes sure they refer to values actually declared within the scope of the user-defined plan where they are used.
- *Literal value ranging checking* - Values which are literal values (i.e. explicitly stated in the code and not `Lookup*` calls) have their range checked by the PIT Builder. For example, when values of type *Angle* are used to specify geographical coordinates (latitude/longitude), when specified using the `Degrees` constructor, the range of the latitude must be between -90.0 and 90.0 and the longitude must be between -180.0 and 180.0. Likewise if `Radians` are used, latitude must be between $-\pi/2$ and $\pi/2$ and longitude must be between $-\pi$ and π.
- *Lookup key checking* - The keys used within `Lookup*` calls are defined through the plug-ins passed by the client application to MPFL during initialization. Before compiling, the PIT Builder making sure keys used in `Lookup*` calls in the mission specification exist by performing test lookups on all those key values.

- *Additional type rules* - Certain static type rules could not be enforced in the grammar so the PIT Builder contains additional logic to enforce the remainder.

5.5.2 Intermediate Representation - Plan Instance Tree

The static semantics are enforced as the PIT Builder builds an intermediate representation (a different abstract syntax) to be used for subsequent MPFL engine stages. This intermediate representation comes in the form of a *plan instance tree.* Recall in the last chapter, the plan instances specified in the mission specification form a tree structure called the plan instance tree. The root of this tree is called the *sortie plan instance* with identifier *sortie.* The sortie instance is an instance of the last user-defined plan in the mission specification.

To reason about the static and runtime semantics of the MSL and how MPFL works, we define the abstract syntax in the form of a grammar. This grammar is described as a set of variant type definitions in the OCaml language which was used as the metalanguage to implement MPFL. This notation not only specifies the abstract syntax but it also encodes aspects of the type system. The syntax of variant types in OCaml is quite simple to understand and closely resembles the form of BNF production rules used earlier to specify the MSL grammar. The OCaml syntax for a variant type is as follows:

```
type <identifier> = valueConstructor1 | ... | valueConstructorN
```

Each possible value the variant type can take is separated by vertical bars and is defined by a constructor. A constructor is simply a string identifier (in OCaml it must start with an uppercase later) followed by a tuple of arguments enclosed in parentheses that encode the value. The constructor does not necessarily have to take arguments in which case it is simply works as an enumeration. Constructors have the following syntax:

```
<constructor identifier> of (type1 * type2 * ... * typeN)
```

A complete example of a variant type is as follows. For example:

```
type point = Point of float*float;;
type shape = Circle of float | Rectangle of point*float*float;;

let myCircle = Circle(4.5);;
let mySquare = Rectangle(Point(0,0), 40.0,40.0);;
```

Example 1 - Defining variant types in OCaml

In OCaml, the variables `myCircle` and `mySquare` would both have type *shape.*

The types used in constructors can be recursive meaning that the type that is being declared on the left-hand side can be used as a constructor argument. The use of asterisks (`*`) refers to ***tuple types*** in OCaml. The arguments to constructors in OCaml constructors are actually a single argument of a tuple type. Tuple values are enclosed within parentheses with arguments separated by commas. In addition to variant types, OCaml allows defining ***record types*** which are similar to a tuple, but each argument has to be named. Values of record types are enclosed within curly braces rather than parentheses like tuples and parameters are separated by semicolons instead of commas.

The abstract syntax specification for MPFL is given with the following OCaml code:

```
(**Basic [atomic] types**)
type mpflString          = String of string | LookupString of mpflString;;
type mpflInteger         = Integer of int | LookupInteger of mpflString
                           | AddInt of mpflInteger * mpflInteger
                           | SubInt of mpflInteger * mpflInteger
                           | MultInt of mpflInteger * mpflInteger
                           | DivInt of mpflInteger * mpflInteger;;
type mpflFloat           = Float of float | LookupFloat of mpflString
                           | AddFloat of mpflFloat * mpflFloat
                           | SubFloat of mpflFloat * mpflFloat
                           | MultFloat of mpflFloat * mpflFloat
                           | DivFloat of mpflFloat * mpflFloat;;
type mpflBool            = Bool of bool | LookupBool of mpflString
                           | StrEqual of mpflString * mpflString
                           | NegateBool of mpflBool
                           | IntGTE of mpflInteger * mpflInteger
                           | IntGT of mpflInteger * mpflInteger
                           | IntEQ of mpflInteger * mpflInteger
                           | IntLT of mpflInteger * mpflInteger
                           | IntLTE of mpflInteger * mpflInteger
                           | FloatGTE of mpflFloat * mpflFloat
                           | FloatGT of mpflFloat * mpflFloat
                           | FloatEQ of mpflFloat * mpflFloat
                           | FloatLT of mpflFloat * mpflFloat
                           | FloatLTE of mpflFloat * mpflFloat;;

type angle               = Degrees of mpflFloat | Radians of mpflFloat;;
type duration            = Seconds of mpflFloat | Minutes of mpflFloat
                           | Hours of mpflFloat;;
type length              = Meters of mpflFloat | Feet of mpflFloat
                           | Yards of mpflFloat;;
type frequency           = Hertz of mpflFloat;;
type power               = Watts of mpflFloat | Horsepower of mpflFloat;;
type energy              = Joules of mpflFloat | KilowattHours of mpflFloat;;

(**Positional types**)
type absolutePosition    = {lat:angle; lon:angle; depth:length};;
type cartesianPosition   = {x:length; y:length; z:length};;
type relativePosition    = {center:absolutePosition; offset:cartesianPosition};;
type position             = AbsolutePosition of absolutePosition
                            | CartesianPosition of cartesianPosition
                            | RelativePosition of relativePosition;;

(**Area types**)
type rectangularArea      = {tl:position; br:position};;
type circularArea         = {centerOfArea: position; radius: length};;
type area                 = RectangularArea of rectangularArea
                           | CircularArea of circularArea;;

(**Time types**)
type clockTime            = {day:mpflInteger; hour:mpflInteger; minute:mpflInteger;
                            second:mpflInteger};;
type unixTime             = {utcSeconds:mpflInteger};;
type time                 = ClockTime of clockTime | UnixTime of unixTime;;
type timeWindow           = {beginTime:time; finishTime:time};;

(** Constraint types**)
type timeConstraint       = {startWindow : timeWindow; endWindow : timeWindow};;
type powerConstraint      = {maxPowerLevel : power; maxEnergyToUse : energy};;
type constraintImp        = TimeConstraint of (string * timeConstraint)
                            | PowerConstraint of (string * powerConstraint);;

(** Different primitive plan instance task types **)
type executeUserProblem    = {userPlanName:string};;
type loiterProblem         = {loiterPosition:position};;
type phoneHomeProblem      = {commDeviceName:string; phoneHomeRate:frequency};;
type searchProblem         = {searchSonarName:string; searchArea:area;
                              laneWidth:length};;
type transitProblem        = {waypoints:position list};;
type useAcousticProblem    = {acousticDeviceName:string; startTime:time;
                              endTime:time; taskDuration:duration;
                              minGap:duration; maxGap:duration};;
type useAutopilotProblem   = {destination:position};;
type useModemProblem       = {modemName:string; modemMessage:mpflString};;
type useSonarProblem       = {sonarName:string; pingRate:frequency};;

type problem               = ExecutePlan of executeUserProblem
                             | Loiter of loiterProblem
                             | PhoneHome of phoneHomeProblem
                             | Search of searchProblem
                             | Transit of transitProblem
                             | UseAcoustic of useAcousticProblem
                             |  UseAutopilot of useAutopilotProblem
                             | UseModem of useModemProblem
                             | UseSonar of useSonarProblem;;

(** Lifetime states **)
type offState             = INIT | DISABLE | SYS_RETRACT | BLOCK;;
type onState              = READY | RUN | FORCE_RUN;;
type endState             = RETRACT | COMPLETE;;
type ltState              = On of onState | Off of offState | End of endState;;

(** Error handlers**)
type planInstChain        = string list;;
type handlerExp           = Disable of planInstChain list
                            | Retract of planInstChain list
                            | HandlerIfThenElse of (mpflBool * handlerExp
                                                    * handlerExp);;
type infeasibleCase       = InfeasibleCase of planInstChain * handlerExp;;
type conflictCase         = ConflictCase of planInstChain list * handlerExp;;
type infeasibleHandler    = InfeasibleHandler of infeasibleCase list;;
type conflictHandler      = ConflictHandler of conflictCase list;;

(** Plan Instance Tree (PIT) (DoExpression and PlanExpression) **)
type opType               = SERIAL | PARALLEL | GROUP | XOR;;
type planExp              = PlanInst of string * ltState * doExp
                                        * problem * constraintImp list
                                        * infeasibleHandler
                                        * conflictHandler
                            | Op of opType * planExp * planExp
                            | IfThenElse of (mpflBool * planExp * planExp)
and doExp                 = NIL | Do of planExp;;
```

The `PlanInst` constructor has type *planExp*. A `PlanInst` constructor represents a plan instance and any children it may have. This constructor is used to define the plan instance tree the PIT Builder generates. The entire plan instance tree is encoded as a single `PlanInst` value representing the *sortie* plan instance. Each `PlanInst` value contains within it the name of the instance, the plan type with plan constructor, constraints bound to the plan instance, a Do Expression, infeasibility handler, conflict handler, and other contextual information. The Do Expression itself is also of type *planExp* and can represent another plan instance, conditional expression, or planning operator. In user-defined plan instances, the Do Expression is where children nodes are encoded which is the means to building the tree. Primitive plan instances are leaves so their Do Expressions are empty (i.e. `NIL`). Plan instances which are instances of user-defined plans (such as the sortie instance) have non-empty infeasibility and conflict handlers if those handlers are defined within their corresponding user-defined plan.

5.5.3 Formal Specification of Type Rules

It was mentioned that the reason the abstract syntax is specified in OCaml is because it implicitly also defines the majority of the MSL ***type system***. All of the variant types used in the metalanguage (OCaml) map almost exactly to the types in the MSL. It is a common practice for language implementors to map the types of the language they are implementing as closely as possible to native types in the metalanguage. This makes it easier for the implementor as they do not have to write the logic to enforce the type rules as the metalanguage's compiler already handles it.

In the study of programming languages, these rules tend to be formally specified using ***inference rules***, which are logical statements denoting antecedent (i.e. premise) and conclusion pairs. Take for example the type *mpflInteger*:

```
type mpflInteger          = Integer of int | LookupInteger of mpflString
                            | AddInt of mpflInteger * mpflInteger
                            | SubInt of mpflInteger * mpflInteger
                            | MultInt of mpflInteger * mpflInteger
                            | DivInt of mpflInteger * mpflInteger;;
```

We can encode this using the following set of inference rules:

$$\frac{v : \overline{int}}{Integer(v) : mpflInteger}\ (1) \qquad \frac{e : mpflString}{LookupInteger(e) : mpflInteger}\ (2)$$

$$\frac{e1 : mpflInteger \quad e2 : mpflInteger}{AddInt(e1, e2) : mpflInteger}\ (3) \qquad \frac{e1 : mpflInteger \quad e2 : mpflInteger}{SubInt(e1, e2) : mpflInteger}\ (4)$$

$$\frac{e1 : mpflInteger \quad e2 : mpflInteger}{MultInt(e1, e2) : mpflInteger}\ (5) \qquad \frac{e1 : mpflInteger \quad e2 : mpflInteger}{DivInt(e1, e2) : mpflInteger}\ (6)$$

The rules fully specify the integer type in the language. Each rule states that if the antecedent (the statements on the top of the bar) can be proven true, then the conclusion (the statements on the bottom of the bar) can be considered true. Rule 1 specifies literal integer values, rule 2 specifies the `LookupInteger` function, and rules 3 to 6 specify basic arithmetic expressions. Specifying type rules in this manner can be useful when formally proving properties about the language, particularly its runtime semantics.[1] It is not necessary to specify the remainder of the type rules above as they can easily be derived using the *mpflInteger* example.

---

[1] The type rules do not utilize a *type environment*, commonly denoted as Γ in programming language literature. Type environments are used to resolve the types of free variables in the language. In the MSL, a type environment is not needed as there are no free variables and types can be determined syntactically.

5.5.4 Formal Specification of Run-time Type Rules

The OCaml variant types define the static semantics of the system, but there are aspects of the type constraints that cannot be verified at compile time. This requires runtime checks by the MPFL runtime in order to verify the validity of the mission specification. We can add additional rules and enhance the type rules implied by the OCaml abstract syntax to handle those semantics. However, these rules are different from the previous inference rules as they are runtime rules and require invoking a ***runtime evaluator***. For example, it was mentioned that the latitude and longitude for values representing geographical positions (represented by record type *absolutePosition* in the abstract syntax) have type *angle*. However their value must fall additionally within a particular range depending on whether the value was specified using the `Degrees` constructor or the `Radians` constructor. We can specify these checks as runtime rules using a semantics model called ***operational semantics***. Operational semantics are a means of formally specifying the meaning of a program[2] as it is executed. In other words, the operational semantics describe how an evaluator (i.e. interpreter) for a language operates as it encounters expressions defined in terms of the abstract syntax. The runtime type checking semantics of an *absolutePosition* value are specified by the following rules:

$$\frac{e \underset{Value}{\hookrightarrow} Degrees(Float(v)), -90 \leq v \leq 90}{ValidLatitude(e)} \quad (7)$$

$$\frac{e \underset{Value}{\hookrightarrow} Radians(Float(v)), -\pi/2 \leq v \leq \pi/2}{ValidLatitude(e)} \quad (8)$$

$$\frac{e \underset{Value}{\hookrightarrow} Degrees(Float(v)), -180 \leq v \leq 180}{ValidLongitude(e)} \quad (9)$$

$$\frac{e \underset{Value}{\hookrightarrow} Radians(Float(v)), -\pi \leq v \leq \pi}{ValidLongitude(e)} \quad (10)$$

$$\frac{ValidLatitude(e1) \quad ValidLongitude(e2)}{ValidAbsolutePosition(\{lat = e1; lon = e2; depth = e3\})} \quad (11)$$

In rules 7 to 11, we specify ***properties*** that we would like certain expressions to have (*ValidLatitude, ValidLongitude, ValidAbsolutePosition*). These properties are metaconstructs used by the MPFL compiler/runtime to ensure validity of typed expressions. Whenever these values are used, we can ensure that those values meet their range constraints (or any other constraints) by having one of these *Valid** properties. In other words, the MPFL compiler/runtime ensures that the *Valid** property exists for each

---

[2] Operational semantics are only one type of runtime semantics model. Other popular semantic models include *axiomatic semantics* and *denotational semantics*.

value specified checked right before its usage. Note that expressions that hold the *Valid** property during an MPFL compilation cycle may not hold the property later on (e.g. `Lookup*` call) causing a runtime error. As an example, rule 11 states an expression of type *absolutePosition* is valid when its latitude and longitude subexpressions have the *ValidLatitude* and *ValidLongitude* properties respectively. These in turn are expressed in rules 7 through 10. Each of these rules call an ***evaluator*** (the arrow) that reduces a typed expression to a value representing each angle to its ***reduced form***[3] (i.e. simplest form): a `Degrees` or `Radians` constructor following by a floating-point value. If the value falls within the correct range, the respective property holds. If these properties do not hold, it means a runtime type error causing the MPFL runtime to raise an exception.

Notice that the arrow representing the evaluator has the word *Value* subscripted on it. MPFL has multiple evaluators that are used to express its complete runtime semantics. In this case, *Value* refers to the ***Value Evaluator***. This evaluator can take any typed expression in the language and return a version of it where all primitive typed (boolean, integer, float, and string) expressions have been reduced to their simplest values. For example, `AddInt(Integer(3),AddInt(Integer(4),Integer(2)))` would evaluate to `Integer(9)` with the Value Evaluator. The Value Evaluator is discussed in more detail later in the chapter when the remainder of runtime rules are described.

The remainder of runtime type checks are ignored due to the sheer quantity of rules. However, Rules 7-11 demonstrate how the remaining runtime checks could be described in a more formal manner.

5.5.5 The Next Step

The output of the PIT Builder is a well-formed plan instance tree where all static semantics have been verified. The Parser and PIT Builder are no longer needed after this point and are never called again. In fact, the Parser and PIT Builder are only called when the user initializes the MPFL compiler/runtime within their client application. Initialization will fail if there are syntactic or type errors and return to the user a list of errors with corresponding line numbers. After the plan instance tree is created, the actual compilation of schedules begins. The subsequent MPFL engine stages are responsible for this and utilize the generated plan instance tree to do so.

5.6 Lifetime State Transition (LST) Evaluator

The plan instance tree is more than just a representation of what the user wants the robot to do and the various parameters, constraints, and error handlers they specified in their mission specification. Rather each plan instance in the tree represents a *runtime context* used by MPFL for managing each task the user wants to perform from beginning to end. Figure 5.3 illustrates the kind of information that is contained in a particular plan instance in some arbitrary plan instance tree. Plan instances are analogous to process control blocks (PCBs) for operating system processes, as described in Chapter 2.

---

[3] The type of operational semantics used here are *big-step operational semantics*. The antecedents in each rule always assume that the expression reduces to its simplest possible form in a single ‘big step’.

Just as PCBs hold the runtime context of the respective processes they represent, plan instances are also in-memory structures that are used by MPFL to manage tasks. MPFL manages tasks through plan instances the way an operating system scheduler manages processes through PCBs.

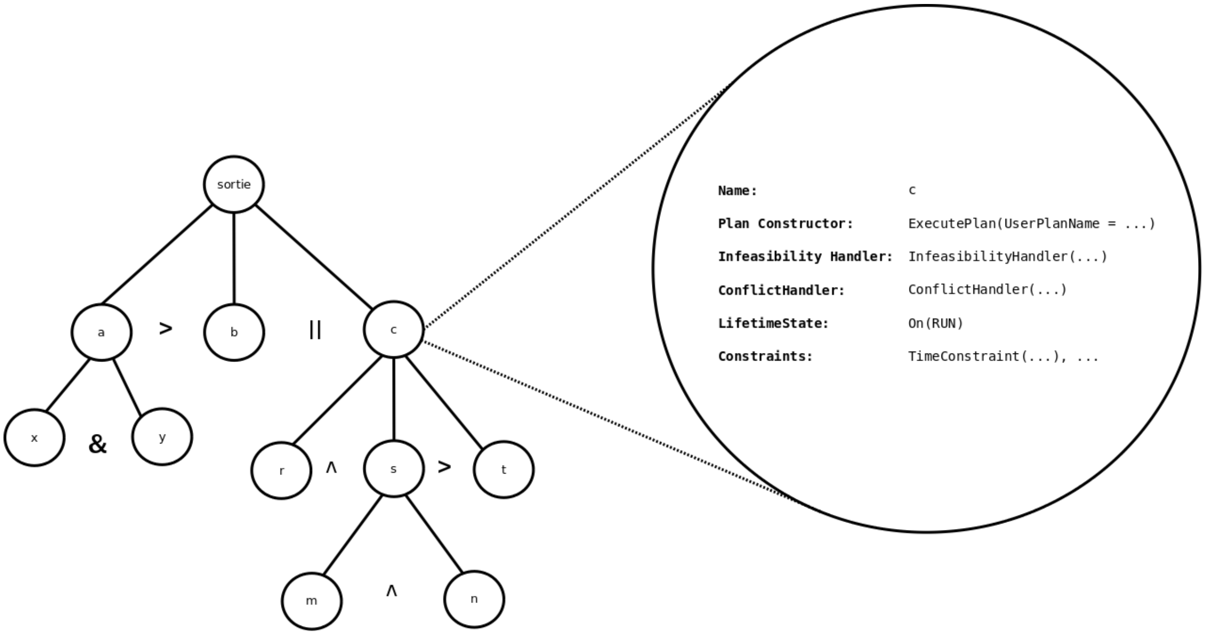

sortie
a
>
b
||
c
x
&
y
r
∧
s
>
t
m
∧
n
Name: c
Plan Constructor: ExecutePlan(UserPlanName = ...)
Infeasibility Handler: InfeasibilityHandler(...)
ConflictHandler: ConflictHandler(...)
LifetimeState: On(RUN)
Constraints: TimeConstraint(...), ...

Fig. 5.3 Each node in the plan instance tree is a context for a task

5.6.1 Lifetime State

The MPFL runtime borrows the idea of *process states* from operating system scheduler implementations to help it manage all the tasks it is to plan. In Chapter 2, it was mentioned that the scheduler of an operating system associates a state for each process (typically contained within the PCB itself) that indicates the status of the process. A process goes through many states in its existence: it can be in a state where it has just been created (INIT), it can be on the ready queue of the OS scheduler waiting to be executed (READY), it can be actually running on a processor (RUNNING), it can be paused as it is waiting for some I/O operation (BLOCKED), and it can be in its end state waiting to have its PCB destroyed and relinquished by the OS (TERMINATE).

Each plan instance also maintains a similar state known as the ***lifetime state***. Each lifetime state is defined by a *macrostate* and a *microstate*. Each plan instance can be in one of three macrostates: *Off*, *On*, or *End*.

- *Off* - The task(s) represented by the plan instance is not currently being executed
- *On* - The task(s) represented by the plan instance is either being executed or is ready to be executed

- *End* - The task(s) represented by the plan instance have been terminated.

Each of these macrostates contains a microstate within that gives more detail about the nature of the macrostate. These are described in Table 5.1. Figure 5.4 shows the transitions that the lifetime state can take.

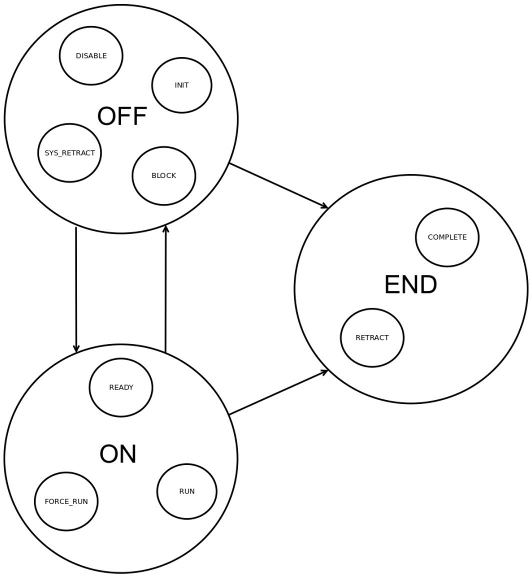


Fig. 5.4 State transition diagram of lifetime state at macrolevel

5.6.2 Usage of Lifetime State and the MPFL Model of Planning

The lifetime state is the key to implementing MPFL's ***model of planning***. By model of planning we mean the logic that MPFL uses to plan and manage tasks in order to achieve goals without violating constraints. When a user writes a mission specification in the MSL, they expect the robot to do exactly what is specified in their program. The MPFL compiler/runtime utilizes the lifetime state to enforce this. The lifetime state is primarily manipulated by the ***Lifetime State Transition (LST) Evaluator*** which is a module that follows the PIT Builder in the MPFL engine. This module manipulates the lifetime states in a way that enforces MPFL's model of planning.

5.6.3 The Lifetime State Transition Evaluator

The LST Evaluator has the responsibility of changing the lifetime state of plan instances in a way that helps the MPFL runtime enforce the semantics of the language, particularly the planning operators, by providing metainformation about the tasks that are to be executed. This metainformation is also used in the next MPFL compiler stage (described later in Section 5.7) in order to perform scheduling.

The LST Evaluator has the ability to take the plan instance tree, or any subtree within it, and attempt to *apply* a lifetime state to it. By *apply* it is meant that the

Macrostate	**Microstate**	**Meaning**
`Off`	`INIT`	The task is in initial state
`Off`	`DISABLE`	The task has been temporarily removed from consideration for planning
`Off`	`SYS_RETRACT`	The task is not considered for planning, but may go into an active state later
`Off`	`BLOCK`	The task is waiting for another task(s) to complete before it can be scheduled
`On`	`READY`	The task is ready to be scheduled
`On`	`RUN`	The task is currently running
`On`	`FORCE_RUN`	The task needs to run immediately
`End`	`COMPLETE`	The task successfully completed
`End`	`RETRACT`	The task was not completed and permanently removed from consideration for planning

Table 5.1 Meaning of lifetime states in MPFL

LST Evaluator attempts to transition as many plan instances in the tree to the specified state. The MPFL runtime always holds on to the plan instance tree in the form of an expression of the aforementioned type *planExp*. The LST Evaluator is simply a function that takes a value of type *planExp* and a lifetime state (defined as OCaml variant type *ltState*) to be applied and returns a new version of the *planExp* expression where the lifetime states of each plan instance has potentially been changed. We can refer to the LST Evaluator function as `LST` with the following type signature[4]:

$$\texttt{LST} : \text{planExp} \rightarrow \text{ltState} \rightarrow \text{planExp}$$

The MPFL runtime invokes the LST Evaluator passing in the *sortie* plan instance in the form of a *planExp* and applies the particular state `On(READY)` meaning a lifetime state where the macrostate is `On` and the microstate is `READY`. The goal of the LST Evaluator is to get as many plan instances into a ready state as possible so that they can be scheduled. However, not all plan instances will transition to the `On(READY)` state because it would violate the semantics of the language.

5.6.4 LST Transition Rules

The LST Evaluator encompasses over 600 rules that define how the lifetime state of the plan instances in a plan instance tree are manipulated. The choice of rule is decided exclusively by what the current lifetime state of each plan instance is and the operators used within each Do Expression. To give a gentle introduction, let us take a hypothetical example with the following MSL code that utilizes a serial operator:

---

[4] All function type notations in this thesis use *curried* notation. There is no technical reason for this use but rather stems from the use of OCaml as the implementation language where all function types are presented to the programmer in curried form.

```
Plan ltDemo
(
   Transit a(...)
   Loiter b(...)
   Do(a > b)
)
```

Example 1 - A simple plan to demonstrate lifetime states

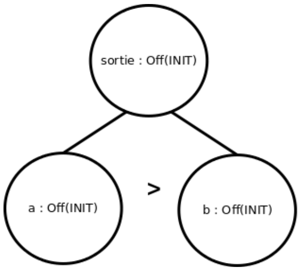


Fig. 5.5 A freshly created plan instance tree representing Example 1

Given this specification in Example 1, the PIT Builder will emit a plan instance tree that is graphically depicted in Figure 5.5. In the figure, each plan instance is denoted by its name and current lifetime state. On startup, each instance is initialized in the `Off(INIT)` state. If we were to pass this tree to the LST Evaluator and apply the state `On(READY)` the LST Evaluator would return a tree as depicted in Figure 5.6.

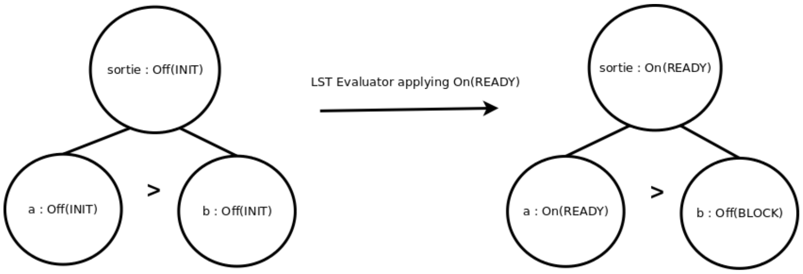


Fig. 5.6 The plan instance tree from Figure 5.5 after the LST Evaluator applies an `On(READY)` state to it

The tree depicted in Figure 5.6 shows that the lifetime state of each plan instance has changed. `sortie` and `sortie`→`a` have gone into the `On(READY)` state and `sortie`→`b` has gone into the `Off(BLOCK)` state. The reason that this happened is because of the meaning of the serial (>) operator; nothing on the right hand side can be attempted until everything on the left hand side has completed. By assigning a blocked state to `sortie`→`b`, MPFL can ensure that the task represented by `sortie`→`b` is not attempted before `sortie`→`a` terminates. Also note that plan instance *sortie* has a lifetime state of `On(READY)`. The lifetime state of a user-defined plan instance represents an *overall lifetime state* of its children, which is also determined by the planning operators used in the instance's Do Expression and the current lifetime state of each child instance.

Every compilation cycle, the MPFL runtime uses the LST Evaluator to apply the `On(READY)` state to the sortie plan instance. The evaluator returns a new copy of the plan instance tree with the lifetime state of the plan instances potentially changed. The runtime throws away the old tree and replaces it with the new one. From an imperative perspective, the LST Evaluator is producing side-effects on the plan instance tree. However, as a functional approach was used this metaphor will not be used. The reason to think of MPFL in a functional sense is not only because the metalanguage (OCaml) is functional, but because it is easier to reason about the semantics due to the difficulty of modeling side-effects in known semantic models.

5.6.5 Formal Semantics of LST Evaluator

There are over 600 rules that describe how the LST Evaluator shifts the lifetime state of the plan instances it processes. The overall evaluator (`LST`) utilizes additional evaluators which in turn utilize each other. Each evaluator can be represented as a typed function described as followed:

- `Value :` $\alpha \rightarrow \alpha$ - This evaluator takes any typed expression in the language and returns the expression where any subexpressions that have a primitive type (i.e. *mpflInteger*, *mpflString*, *mpflBoolean*, *mpflFloat*) are evaluated and reduced to a single value. This evaluator was briefly mentioned earlier in Section 5.5.4.

- `ChangeOnState : planExp` → `ltState` → `planExp` - This evaluator is used when plan instances are being switched to an `On(*)` state. The LST evaluator is not allowed to change certain states to the `On` state, so this evaluator enforces that.

- `CurrentGroup : ltState` → `ltState` → `ltState` - This evaluator is used to determine the overall lifetime state of two expressions bound by a group operator based on their respective lifetime states.

- `CurrentParallel : ltState` → `ltState` → `ltState` - This evaluator is used to determine the overall lifetime state of two expressions bound by a parallel operator based on their respective lifetime states.

- `CurrentSerial : ltState` → `ltState` → `ltState` - This evaluator is used to determine the overall lifetime state of two expressions bound by a serial operator based on their respective lifetime states.

- `CurrentXor : ltState` → `ltState` → `ltState` - This evaluator is used to determine the overall lifetime state of two expressions bound by an xor operator based on their respective lifetime states.

- `Current : planExp` → `ltState` - This gives us the current overall lifetime state of a *planExp*. Not only do plan instances have a current lifetime state, but all expressions of type *planExp* have an overall current lifetime state determined by the current lifetime state of each plan instance contained within the expression along with any planning operators used.

- `NextGroup : planExp` → `planExp` → `ltState` → `(ltState * ltState)` - This evaluator takes two *planExp* bound by the group operator and a lifetime state that is to be applied that expression, and returns a lifetime state to apply to each subexpression based on the semantics of the group operator.

- `NextParallel : planExp` → `planExp` → `ltState` → `(ltState * ltState)` - This evaluator takes two *planExp* expressions bound by the parallel operator and a lifetime state that is to be applied to that expression, and returns a lifetime state to apply to each subexpression based on the semantics of the parallel operator.

- `NextSerial : planExp` → `planExp` → `ltState` → `(ltState * ltState)` - This evaluator takes two *planExp* expressions bound by the serial operator and a lifetime state that is to be applied to that expression, and returns a lifetime state to apply to each subexpression based on the semantics of the serial operator.

- `NextXor : planExp` → `planExp` → `ltState` → `(ltState * ltState)` - This evaluator takes two *planExp* expressions bound by the xor operator and a lifetime state that is to be applied to that expression, and returns a lifetime state to apply to each subexpression based on the semantics of the xor operator.

- `LST : planExp` → `ltState` → `planExp` - This is the evaluator that represents the entire LST Evaluator. It takes a *planExp* expression, a lifetime state to apply to the expression and in turn returns an updated plan expression with new lifetime states.

We can describe the ***runtime semantics*** of the LST Evaluator as a series of inference rules that utilize the other evaluators. The other evaluators can similarly be described using inference rules. Unlike the static semantics in Section 5.5, these rules describe how the abstract syntax (the plan instance tree) is manipulated during runtime. Due to the sheer number of rules, only a small subset of rules will be shown to give the flavor of the concept. Also, we will simplify the abstract syntax to be more minimal as all that is needed by the LST Evaluator is the name, lifetime state, and the Do Expression contained within each plan instance. The ignored parameters can be assumed to be immutable. The reduced abstract syntax is as follows:

```
type planExp               = PlanInst of string * ltState * doExp
                             | Op of opType * planExp * planExp
                             | IfThenElse of (mpflBool * planExp * planExp)
and doExp                  = NIL | Do of planExp;;
```

The following subsections describe the rules that compose the various evaluators described above.[5]

---

[5] Just as type rules can use a *type environment* (typically denoted as Γ), runtime evaluators can also utilize a *runtime environment* (typically denoted as ρ). The purpose of the runtime

5.6.5.1 `Value` Evaluator (Partial Set of Rules)

The `Value` Evaluator's purpose is to take any typed expression and reduce any primitive typed (integer, float, string, and boolean) subexpressions to their simplest form. Not all rules are shown because of the sheer quantity, but a small subset is given to give the idea.

$$\frac{e \underset{Value}{\hookrightarrow} v}{Integer(e) \underset{Value}{\hookrightarrow} Integer(v)} \tag{12}$$

$$\frac{e \underset{Value}{\hookrightarrow} v}{Float(e) \underset{Value}{\hookrightarrow} Float(v)} \tag{13}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} v_1 \quad e_2 \underset{Value}{\hookrightarrow} v_2 \quad v = v_1 + v_2}{AddInt(e_1, e_2) \underset{Value}{\hookrightarrow} v} \tag{14}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} v_1 \quad e_2 \underset{Value}{\hookrightarrow} v_2 \quad v = v_1 / v_2}{DivFloat(e_1, e_2) \underset{Value}{\hookrightarrow} v} \tag{15}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} v_1 \quad e_2 \underset{Value}{\hookrightarrow} v_2 \quad v_1 == v_2}{StrEq(e_1, e_2) \underset{Value}{\hookrightarrow} Bool(true)} \tag{16}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} v1 \quad e_2 \underset{Value}{\hookrightarrow} v_2 \quad v_1 \neq v_2}{StrEq(e_1, e_2) \underset{Value}{\hookrightarrow} Bool(false)} \tag*{((17)}$$

$$\frac{e \underset{Value}{\hookrightarrow} v}{Meters(e) \underset{Value}{\hookrightarrow} Meters(v)} \tag{18}$$

$$\frac{e \underset{Value}{\hookrightarrow} v}{Feet(e) \underset{Value}{\hookrightarrow} Feet(v)} \tag{19}$$

$$\frac{e \underset{Value}{\hookrightarrow} v}{Yards(e) \underset{Value}{\hookrightarrow} Yards(v)} \tag{20}$$

5.6.5.2 `ChangeOnState` Evaluator (Complete Set of Rules)

The `ChangeOnState` Evaluator is utilized when applying the `On` state to an expression. The only purpose of this evaluator is to ensure that plan instances already in the `On(RUN)` state are not demoted to an `On(READY)` state.

$$OnMicroStates = \{READY, RUN, FORCE\_RUN\}$$

$$OnMacroStates = \{On(READY), On(RUN), On(FORCE\_RUN)\}$$

$$\frac{cs \underset{Value}{\hookrightarrow} On(v_1) \quad v_1 = RUN|FORCE\_RUN \quad app \underset{Value}{\hookrightarrow} On(v_2) \quad v_2 \in \{OnMicroStates \backslash RUN\}}{< cs, app > \underset{ChangeOnState}{\hookrightarrow} On(v_1)} \tag{21}$$

---

environment is to lookup the values bound to variables and to substitute them into expressions as the evaluator encounters them. Just as the type environment was not necessary as types can be determined solely by abstract syntax structure, a runtime environment is not necessary as the MSL has no real concept of variables. Though the concept of `Lookup*` calls is similar to a variable, it will later be shown that `Lookup*` expressions are reduced via substitution in the *Value* Evaluator.

$$\frac{cs \underset{Value}{\hookrightarrow} On(RUN) \quad app \underset{Value}{\hookrightarrow} On(s) \quad s \in OnMicroStates}{< cs, app > \underset{ChangeOnState}{\hookrightarrow} On(RUN)} \tag{22}$$

$$\frac{cs \underset{Value}{\hookrightarrow} lst \quad lst \in onMacroStates \quad app \underset{Value}{\hookrightarrow} v \quad v \notin OnMacroStates}{< cs, app > \underset{ChangeOnState}{\hookrightarrow} v} \tag{23}$$

5.6.5.3 `CurrentSerial` Evaluator (Complete Set of Rules)

The `CurrentSerial` Evaluator determines the overall lifetime state of a serial expression based on the lifetime states of each respective expression.

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2)}{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} Off(v_1)} \tag{24}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2)}{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} On(v_1)} \tag{25}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2)}{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} Off(v_2)} \tag{26}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2)}{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} On(v_2)} \tag{27}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2)}{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} End(v_2)} \tag{28}$$

5.6.5.4 `CurrentXor` Evaluator (Complete Set of Rules)

The `CurrentXor` Evaluator determines the overall lifetime state of an xor expression based on the lifetime states of each respective expression.

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2)}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} Off(v_1)} \tag{29}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2)}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} On(v_1)} \tag{30}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad v_1 = RETRACT \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2)}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} Off(v_2)} \tag{31}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad v_1 \neq RETRACT \quad e_2 \underset{Value}{\hookrightarrow} End(v_2)}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} End(v_2)} \tag{32}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad v_1 = RETRACT \quad e_2 \underset{Value}{\hookrightarrow} On(v_2)}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} On(v_2)} \tag{33}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad v_1 = RETRACT \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad v_2 = RETRACT}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} End(RETRACT)} \tag{34}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad v_1 \neq v_2 \quad v_1 = COMPLETE | v_2 = COMPLETE}{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} End(COMPLETE)} \tag{35}$$

5.6.5.5 `Current` Evaluator (Complete Set of Rules)

The `Current` Evaluator determines the overall lifetime state of any *planExp*. Note that this evaluator turns around and calls the various `Current*` evaluators in its operation.

$$\frac{}{PlanInst(n, cs, NIL) \underset{Current}{\hookrightarrow} cs} \tag{36}$$

$$\frac{e \underset{Current}{\hookrightarrow} v}{PlanInst(n, cs, Do(e)) \underset{Current}{\hookrightarrow} v} \tag{37}$$

$$\frac{< e_1, e_2 > \underset{CurrentSerial}{\hookrightarrow} v}{Op(SERIAL, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{38}$$

$$\frac{< e_1, e_2 > \underset{CurrentXor}{\hookrightarrow} v}{Op(XOR, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{39}$$

$$\frac{< e_1, e_2 > \underset{CurrentGroup}{\hookrightarrow} v}{Op(GROUP, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{40}$$

$$\frac{< e_1, e_2 > \underset{CurrentParallel}{\hookrightarrow} v}{Op(PARALLEL, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{41}$$

$$\frac{cond \underset{Value}{\hookrightarrow} Bool(true) \quad e_1 \underset{Current}{\hookrightarrow} v}{IfThenElse(cond, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{42}$$

$$\frac{cond \underset{Value}{\hookrightarrow} Bool(false) \quad e_2 \underset{Current}{\hookrightarrow} v}{IfThenElse(cond, e_1, e_2) \underset{Current}{\hookrightarrow} v} \tag{43}$$

5.6.5.6 `NextSerial` Evaluator (Complete Set of Rules)

The `NextSerial` Evaluator determines the state to apply to each operand in a serial expression during state transitioning based on the lifetime state that is to be applied to the complete expression. The return value is a pair of lifetime states where the first is to be applied to the first operand and the second to the second operand.

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < Off(v_3), Off(v_3) >} \tag{44}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < On(v_3), Off(BLOCK) >} \tag{45}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_3), End(v_3) >} \tag{46}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad < On(v_1), On(v_3) > \underset{ChangeOnState}{\hookrightarrow} v_4}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < v_4, Off(BLOCK) >} \tag{47}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_3), End(v_3) >} \tag{48}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), Off(v_3) >} \tag{49}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), On(v_3) >} \tag{50}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), End(v_3) >} \tag{51}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), Off(v_3) >} \tag{52}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad < On(v_2), On(v_3) > \underset{ChangeOnState}{\hookrightarrow} v_4}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), v_4) >} \tag{53}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), End(v_3) >} \tag{54}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), End(v_2) >} \tag{55}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), End(v_2) >} \tag{56}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextSerial}{\hookrightarrow} < End(v_1), End(v_2) >} \tag{57}$$

5.6.5.7 `NextXor` Evaluator (Complete Set of Rules)

The `NextXor` Evaluator determines the state to apply to each operand in an xor expression during state transitioning based on the lifetime state that is to be applied to the complete expression. The return value is a pair of lifetime states where the first is to be applied to the first operand and the second to the second operand.

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < Off(v_3), Off(v_3) >} \tag{58}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < On(v_3), Off(SYS\_RETRACT) >} \tag{59}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} Off(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_3), End(RETRACT) >} \tag{60}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < Off(v_3), Off(v_3) >} \tag{61}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad < On(v_1), On(v_3) > \underset{ChangeOnState}{\hookrightarrow} v_4}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < v_4, Off(SYS\_RETRACT)) >} \tag{62}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3) \quad v_3 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_3), On(READY) >} \tag{63}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} On(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3) \quad v_3 \neq RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_3), End(RETRACT) >} \tag{64}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), Off(v_3) >} \tag{65}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3) \quad v_1 \neq RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(RETRACT) >} \tag{66}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), On(v_3) >} \tag{67}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad v_1 \neq RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), On(v_3) >} \tag{68}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(v_3) >} \tag{69}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} Off(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3) \quad v_1 \neq RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(RETRACT) >} \tag{70}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), Off(v_3) >} \tag{71}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), On(v_3) >} \tag{72}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} On(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3) \quad v_1 = RETRACT}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(v_3) >} \tag{73}$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} Off(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(v_2) >} \quad (74)$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} On(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(v_2) >} \quad (75)$$

$$\frac{e_1 \underset{Value}{\hookrightarrow} End(v_1) \quad e_2 \underset{Value}{\hookrightarrow} End(v_2) \quad app \underset{Value}{\hookrightarrow} End(v_3)}{< e_1, e_2, app > \underset{NextXor}{\hookrightarrow} < End(v_1), End(v_2) >} \quad (76)$$

5.6.5.8 `LST` Evaluator (Complete Set of Rules)

The `LST` Evaluator is the master evaluator that we are trying to describe. This evaluator takes a *planExp* and applies some desired *ltState* value to it. The end result is copy of the passed *planExp* with lifetime states of some of the plan instances potentially changed.

$$\frac{cs \underset{Value}{\hookrightarrow} ns \quad ns \in OnMacroStates \quad < cs, app > \underset{ChangeOnState}{\hookrightarrow} ns}{< PlanInst(n, cs, NIL), app > \underset{LST}{\hookrightarrow} PlanInst(n, ns, NIL))} \quad (77)$$

$$\frac{cs \underset{Value}{\hookrightarrow} ns \quad ns \notin OnMacroStates}{< PlanInst(n, cs, NIL), app > \underset{LST}{\hookrightarrow} PlanInst(n, app, NIL))} \quad (78)$$

$$\frac{< e, app > \underset{LST}{\hookrightarrow} e' \quad e' \underset{Current}{\hookrightarrow} ns}{< PlanInst(n, cs, Do(e)), app > \underset{LST}{\hookrightarrow} PlanInst(n, ns, Do(e'))} \quad (79)$$

$$\frac{\begin{array}{c} e_2 \underset{Current}{\hookrightarrow} cs_2 \quad e_1 \underset{Current}{\hookrightarrow} cs_1 \\ < cs_1, cs_2, app > \underset{NextSerial}{\hookrightarrow} < ns_1, ns_2 > \\ < e_1, ns_1 > \underset{LST}{\hookrightarrow} v_1 \\ < e_2, ns_2 > \underset{LST}{\hookrightarrow} v_2 \end{array}}{< Serial(e_1, e_2), app > \underset{LST}{\hookrightarrow} Serial(v_1, v_2)} \quad (80)$$

$$\frac{\begin{array}{c} e_2 \underset{Current}{\hookrightarrow} cs_2 \quad e_1 \underset{Current}{\hookrightarrow} cs_1 \\ < cs_1, cs_2, app > \underset{NextGroup}{\hookrightarrow} < ns_1, ns_2 > \\ < e_1, ns_1 > \underset{LST}{\hookrightarrow} v_1 \\ < e_2, ns_2 > \underset{LST}{\hookrightarrow} v_2 \end{array}}{< Group(e_1, e_2), app > \underset{LST}{\hookrightarrow} Group(v_1, v_2)} \quad (81)$$

$$\frac{\begin{array}{c} e_2 \underset{Current}{\hookrightarrow} cs_2 \quad e_1 \underset{Current}{\hookrightarrow} cs_1 \\ < cs_1, cs_2, app > \underset{NextParallel}{\hookrightarrow} < ns_1, ns_2 > \\ < e_1, ns_1 > \underset{LST}{\hookrightarrow} v_1 \\ < e_2, ns_2 > \underset{LST}{\hookrightarrow} v_2 \end{array}}{< Parallel(e_1, e_2), app > \underset{LST}{\hookrightarrow} Parallel(v_1, v_2)} \tag{82}$$

$$\frac{\begin{array}{c} e_2 \underset{Current}{\hookrightarrow} cs_2 \quad e_1 \underset{Current}{\hookrightarrow} cs_1 \\ < cs_1, cs_2, app > \underset{NextXor}{\hookrightarrow} < ns_1, ns_2 > \\ < e_1, ns_1 > \underset{LST}{\hookrightarrow} v_1 \\ < e_2, ns_2 > \underset{LST}{\hookrightarrow} v_2 \end{array}}{< Xor(e_1, e_2), app > \underset{LST}{\hookrightarrow} Xor(v_1, v_2)} \tag{83}$$

$$\frac{cond \underset{Value}{\hookrightarrow} Bool(true) \quad < e_1, app > \underset{LST}{\hookrightarrow} v_1 \quad < e_2, Off(SYS\_RETRACT) > \underset{LST}{\hookrightarrow} v_2}{< IfThenElse(cond, e1, e2), app > \underset{LST}{\hookrightarrow} IfThenElse(cond, v_1, v_2)} \tag{84}$$

$$\frac{cond \underset{Value}{\hookrightarrow} Bool(false) \quad < e_1, Off(SYS\_RETRACT) > \underset{LST}{\hookrightarrow} v_1 \quad < e_2, app > \underset{LST}{\hookrightarrow} v_2}{< IfThenElse(cond, e1, e2), app > \underset{LST}{\hookrightarrow} IfThenElse(cond, v_1, v_2)} \tag{85}$$

5.6.6 Purpose of Formal Specification

In programming language theory, the purpose of defining a language formally serves two purposes. The first is to describe precisely what a program in the language means so that somebody creating a compiler or interpreter for the language can implement it. The second is to be able to prove properties about the language, mainly those that show its *correctness*. What defines correctness depends on the language, but one of the typical properties language designers want to prove is if their language is ***well-typed***. To show that a language is well-typed, we must prove that the evaluation semantics (the operational semantics in this case) exhibit two properties: ***type preservation*** and ***evaluation progress*** (Pierce 2002). Type preservation means that if we take any typed expression in the language and evaluate it, each new expression that is created during the evaluation leading up to the final reduced expression has the same type. Evaluation progress means that for every valid expression in the language, the evaluator will be able to process it and produce another valid expression in the language (which in turn exhibits the progress property and so on). Proving these properties typically requires looking at every possible expression in the language via a proof by structural induction. If a language is shown to be well-typed, it is an extremely powerful statement about the language as it guarantees that a valid program will never end up in an invalid state. Unsafe languages such as C and C++ are not well-typed because they allow users to circumvent the type system by being able to cast any value in the language to any type, even if it is not valid. This is also why use of such languages result in programs that have a much higher tendency for bugs, memory leaks, and security vulnerabilities.

In this thesis, no proof is given to show the correctness of the language as it is beyond the scope of the thesis. It would be wise in the future to create such a proof.

For example, if one looks at the rules for one of the evaluators specified above, they may notice that even for evaluators where all rules are given, rules are not specified for all possible expressions. This is because not all expressions are valid in the language, even if they can be expressed by the abstract syntax (just as with the static semantics). By showing a proof of the language being well-typed, there is a guarantee that none of those invalid expressions can ever be reached. It is also interesting to note that because the MSL is not as powerful as a general-purpose language, one maybe able to prove more interesting properties than just being well-typed, such as *"every MSL program is decidable."*

Setting up such a proof requires a complete set of type and runtime rules that describes the entire compiler as a single evaluator function. For each possible expression in the language, one has to show inductively that both types are preserved during evaluation (*preservation*) and that indeed the evaluation can proceed by a single step to another valid expression (*progress*). Practically this is difficult because not only is the number of rules large, but only a subset of the semantics have been formalized.

The subsequent section discusses the MPFL runtime stage that follows the LST Evaluator. Unlike the LST Evaluator, the runtime semantics will be described informally.

5.7 Planner Invocation (PI) Evaluator

After the LST Evaluator completes its execution, the plan instance tree has been updated with new lifetime states for each plan instance. Up to this point, it is understood that the lifetime state is important in MPFL to enforce the semantics of the language. but how it uses that state has yet to be explained. The lifetime state is used extensively in the next component of the MPFL engine, the ***Planner Invocation (PI) Evaluator***, where the bulk of system computation occurs. The PI Evaluator is the final stage in the MPFL engine and the point where schedules are generated for the robot's actuators and subsystems.

5.7.1 Plugins and Initialization

When a user initializes the MPFL runtime/compiler in their client application, in addition to passing in a mission specification file written in the MSL, the user also must pass in a set of plugins. These plugins are used almost exclusively by the PI Evaluator. One of the plugins is called the ***Knowledge Base*** and the remainder are called ***Planners***. These plugins are implemented through an object-oriented API by subclassing from a set of provided base classes and then passing in instances of those classes (i.e. objects) into the MPFL initialization function. We can now define the type signature of the initialization function, `initialize_mpfl`:

```
initialize_mpfl:  string -> knowledgeBase -> planner list -> unit
```

The function `initialize_mpfl` takes a string representing the path and name of the MSL mission specification file, an object of the class *knowledgeBase* and a list of objects of the class *planner*. The base classes that MPFL provides define an abstract interface that forces implementors of the plugins to implement a set of callback methods needed by the runtime. During initialization, the specification is verified within the Parser and PIT Builder stages. If there are any problems, an exception is raised indicating the system could not be bootstrapped.

5.7.2 Knowledge Base

The purpose of the knowledge base is to provide perception information to MPFL and the various planners that are passed into the system. The knowledge base is what drives the `Lookup*` calls that users can utilize in the MSL. Users define a knowledge base by subclassing from the MPFL-provided base class *knowledgeBase* which requires overriding four abstract methods with the following type signatures:

- `lookup_string : string -> string`
- `lookup_float : string -> float`
- `lookup_integer: string -> int`
- `lookup_bool: string -> bool`

The first argument to each function corresponds to the set of valid keys that one can utilize with the various lookup calls. The implementor of the knowledge base must return a value based on the key passed to the method. If not found, they should call the base class version of the method which will raise an exception likely terminating the MPFL runtime. Users can also add additional methods to the knowledge base which can then be accessed by the various planners in the system.

5.7.2.1 Formal Semantics of Lookup Calls

The operational semantics of the Value Evaluator when dealing with `Lookup*` calls can now be specified formally. Note that the '#' notation below indicates invocation of a method on on object (i.e. ***object#method***) in OCaml.

$$\frac{key \underset{Value}{\hookrightarrow} keyVal \quad \texttt{KnowledgeBase\#lookup\_string keyVal} \underset{OCaml}{\hookrightarrow} v}{LookupString(key) \underset{Value}{\hookrightarrow} v} \tag{86}$$

$$\frac{key \underset{Value}{\hookrightarrow} keyVal \quad \texttt{KnowledgeBase\#lookup\_float keyVal} \underset{OCaml}{\hookrightarrow} v}{LookupFloat(key) \underset{Value}{\hookrightarrow} v} \tag{87}$$

$$\frac{key \underset{Value}{\hookrightarrow} keyVal \quad \texttt{KnowledgeBase\#lookup\_bool keyVal} \underset{OCaml}{\hookrightarrow} v}{LookupBool(key) \underset{Value}{\hookrightarrow} v} \tag{88}$$

$$\frac{key \underset{Value}{\hookrightarrow} keyVal \quad \texttt{KnowledgeBase\#lookup\_int keyVal} \underset{OCaml}{\hookrightarrow} v}{LookupInteger(key) \underset{Value}{\hookrightarrow} v} \qquad (89)$$

5.7.3 Planners

Planners are the modules that actually perform scheduling in the system. Each primitive plan in the MSL has a corresponding planner associated with it. For example, there is a `Search` Planner for the `Search` type, a `UseAutopilot` Planner for the `UseAutopilot` type, and so on. Planners are implemented by subclassing from a set of MPFL-provided base classes, one for each primitive plan type. Each planner is simply a scheduler for the tasks represented by plan instances of a certain primitive plan type. Recall that each plan instance is a representation of a living instance of a plan (i.e. task). The PI Evaluator extracts from the plan instance tree each set of primitive plan instances of a certain primitive plan type and then feeds them to the corresponding planner of the same plan type and requests the planner to schedule the tasks. For example, all the `Loiter` plan instances in the tree, regardless of where they are in the tree, are fed to the Loiter planner. The `Loiter` planner can then build a schedule for all `Loiter` plan instances via a callback method implemented by the designer of that particular `Loiter` planner. The planner can utilize the metadata contained within the plan instance (name, constraints, lifetime state, task parameters, etc) then to build a valid schedule for each one.

5.7.4 Planners and the MSL

One of the interesting things about MPFL's Planner API is that planners have the features of the MSL integrated into the API. Planners have the ability to create additional primitive plan instances that can be attached as children to the plan instances it is scheduling, just as one can in the MSL by creating plan instances of a user-defined plan. For example, performing a search operation requires use of both the vehicle's navigation system as well as its sensors. In this case, an implementation of a `Search` planner could create a `Transit` plan instance for performing the search movement and a `UseSonar` plan instance for utilizing the sensor for each of its `Search` plan instances. These children plan instances can be attached to each Search plan instance and end up extending the plan instance tree. Additionally, the Planner API allows users to describe temporal relations between these new plan instances just as in the MSL using a Do Expression with all available planning operators. In the example of the `Search` planner, the planner implementation not only wants to create a `Transit` and `UseSonar` plan instance for each `Search` plan instance, but also to describe the relationship through a parallel (`||`) operator. The API also allows the user to create additional constraints (i.e. time, power) as in the MSL and bind them to the newly created children.

5.7.5 Planner Graph

The ability for planners to create new problems (i.e. plan instances) is implemented by arranging planners in a hierarchy, specifically a directed acyclic graph called the ***planner graph***. In the example with the `Search` planner, the `Transit` and `UseSonar` planners would be children of the `Search` planner as the `Search` planner can create subproblems for the `Transit` and `UseSonar` problems. When a planner designer creates a planner, they must pass which types of children plan instances they intend to create in the constructor specifying the ***planner dependencies***. Dependencies of planners cannot be circular, hence why the planner graph is a directed acyclic graph. Having circular dependencies would make the PI Evaluator algorithm undecidable, hence the restriction. When `initialize_mpfl` is called to bootstrap the system, MPFL creates a topographical sorting of the planners to guarantee this property. If a topographical sorting does not exist, `initialize_mpfl` will raise an exception.

5.7.6 Planner Isolation and Component Reusability

MPFL was designed to make it easy to reuse components. Those components refer primarily to planners where most of the application-specific code sits. MPFL is able to achieve this by making planners independent entities which can be built without having any understanding of implementation details of any other planners or the MPFL compiler/runtime itself. Though planners can create additional plan instances for their children planners, they do not have nor should have an understanding of how those children planners operate. This makes it easy to swap out one planner with another of the same type transparently and potentially even allowing for hot swapping. The caveat is that 1) the swapped-in planner's dependencies do not create cycles in the planner graph and 2) the swapped-in planner has all the information it needs within the existing knowledge base. When a person implements a specific planner, they focus exclusively on that problem and do not worry about any of the other planners in the system. This makes it easier for autonomy developers as they can focus completely on building schedulers for a particular type of task without worrying about how other planners are implemented. This also makes designing the system easier as it breaks the entire planning autonomy system into piecemeal chunks which can be reasoned about individually. It is also not necessary for users to provide a planner for each type of primitive plan in the language but rather only the subset that the user uses in their mission specification. For example, if one does not create `Loiter` plan instances in their mission specification, it is not necessary to create a `Loiter` planner. However if one were to declare `Loiter` plan instances and no `Loiter` planner was provided, it would result in an exception during initialization.

5.7.7 The Planner API

Each kind of primitive plan in the language has a corresponding base class that users must subclass in order to implement that planner. Each of the bases class has a common base class of type *planner*. Each class defines five methods that the user must override in the same manner as they would with the *knowledgeBase*. The methods require an additional set of OCaml types not defined in the abstract syntax that are exclusively part of the MPFL API.

```
type userPlanExp   =  PlanInst of (string * problem)
                      | Op of opType * userPlanExp * userPlanExp
                      | IfThenElse of (mpflBool * userPlanExp * userPlanExp)
                      | With of (constraintImp);;
type errorReason    = string;;
type planInstName   = string;;
type planInstChain  = string;;
type scheduleRecord = ScheduleRecord of (time * time * planInstChain * string);;
type schedule       = Schedule of scheduleRecord list
                      | ScheduleInfeasible of (errorReason * planInstChain) list
                      | ScheduleConflict of (errorReason * (planInstChain list)) list
                      | ScheduleNoUpdate
                      | ScheduleAutobuild of string;;
```

The type *userPlanExp* is a simplified version of the type *planExp* which is utilized by planners to build children plan instances. The type *scheduleRecord* encodes a single row of a schedule where the first parameter refers to the time at which the command encoded within the row should be issued, the second indicates the time when the task associated with the command is expected to complete, the third parameter indicates the plan instance the row is associated with as a string in chain notation (e.g. a → b → c), and the last in a string encoding the actual command that should be issued. The schedule itself is encoded with type *schedule* using the constructor `Schedule` which takes a list of *scheduleRecords* as a parameter. The *schedule* type is a variant type, in the event a schedule cannot be formed it also can take the value of an infeasibility error or conflict error utilizing the `ScheduleInfeasible` and `ScheduleConflict` constructors respectively. For infeasibilities, the constructor takes a list of pairs, each one containing a reason for the infeasibility and the name of the plan instance (again in chain notation) that is infeasible. For conflicts, the constructor is similar, but takes a list of plan instance names as the second argument in the constructor indicating conflicting instances. The constructor `ScheduleNoUpdate` is utilized when no updates are made to last schedule created. Finally, the `ScheduleAutobuild` constructor is a convenient way for planners to automatically build their schedule based on the way their children plan instances are scheduled in child planners. For each plan instance the planner is scheduling for, MPFL will determine the earliest start time and latest end time amongst all its children instances according to their corresponding schedules. A row is automatically built in the schedule with the corresponding minimum start time and maximum end times for each plan instance in the automatically built schedule. The string constructor argument is used to populate the command field for each row.

5.7.7.1 The Class `planInstance`

In addition to the new abstract types, we must also define class *planInstance* before defining the class *planner*. Plan instances are fed to each planner in the form of objects of this class. Each type of primitive plan in addition to having its own *planner* base class has a representative subclass of *planInstance* (e.g. *loiterPlanInstance* is a subclass of *planInstance* and represents a `Loiter` plan instance). The use of objects instead of simple variant type values is useful for plan instances because it provides a means of extracting information about the plan instance via methods. Though the same could be achieved with functions and closures, methods take functions further as they are a means of *encapsulating* those functions within the realm of the object and its internal state. For example, it is typical in modern software engineering to utilize an ***interactive developer environment (IDE)*** to help manage and ease development (e.g. *Eclipse*, *Microsoft Visual Studio*, *NetBeans*). IDEs for object-oriented languages like OCaml pop up a list of methods applicable to an object based on its class type when one tries to invoke a method on that object. Internally MPFL uses a functional style, but externally it utilizes an object-oriented approach to make it easier for developers to reason about and discover the API. The base class *planInstance* has the following methods:

- `get_lifetime_state : unit -> ltState` - Gets the current lifetime state of the plan instance
- `get_constraints : unit -> constraintImp list` - Gets all the constraints bound to the plan instance
- `get_all_time_constraints : unit -> timeConstraint list` - Gets all time constraints bound to the plan instance
- `get_all_power_constraints : unit -> powerConstraint list` - Gets all power constraints bound to the plan instance
- `get_overall_time_constraint : unit -> timeConstraint` - Gets the set intersection of all time constraints as a single time constraint
- `get_overall_power_constraint : unit -> powerConstraint` - Gets the set interesection of power constraints as a single power constraint
- `get_problem : unit -> problem` - Gets the underlying problem based on the specific primitive plan type the plan instance represents (e.g. `Loiter, Search, UseModem, etc`)

There is a subclass of *planInstance* for each specific primitive plan. These subclasses provide additional methods that allow API users to access information about the specific problem the plan instance represents. For example, the class *searchPlanInstance* provides a set of methods to access information about the search area, lane width, sensor to use, etc. Each subclass provides methods to access information contained within the plan instance constructor for the respective type of primitive plan it represents. The MPFL API also provides an additional module which allows users to easily convert values returned by these methods into different units and representations. For example, when one gets the lane width for the search problem, they can get it back in any unit of length they want; it does not matter how it was specified. Additionally, useful calculations applicable to specific types are encoded as functions in this module. For example, if one has two values of type *position*, the API provides a method to get the distance between the two even though the underlying representation of one position maybe as a geographical (i.e. lat/lon) position and the other a Cartesian position relative to some fixed geographical point. API users do not worry about how the user specified the units in the MSL — in fact they have no idea. All they can do is ask for the value in the units that are most convenient to them.

5.7.7.2 The Class `planner`

Using these types, we can now define the class *planner*. The class *planner* is a type polymorphic class, meaning that it requires a type parameter α. α is constrained to be a subclass of another class *planInstance* (i.e. $\alpha <: planInstance$). The class methods are defined as follows:

- `on_ready_to_running:` α `list -> (planInstChain * bool) list` - Callback method that is invoked when asking the planner to specify which plan instances to switch from a `On(READY)` state to an `On(RUN)` state.
- `on_forcerun_to_running:` α `list -> (planInstChain * bool) list` - Callback method that is invoked when asking the planner to specify which plan instances to switch from a `On(FORCE_RUN)` state to an `On(RUN)` state.
- `on_running_to_complete:` α `list -> (planInstChain * bool) list` - Callback method that is invoked when asking the planner to specify which plan instances to switch from a `On(RUN)` state to an `End(COMPLETE)` state.
- `on_ask_for_subproblems:` α `list -> (planInstChain * userPlanExp) list` - Callback method that is invoked when asking the planner to create subproblems (i.e. child plan instances) in the form of *userPlanExp*.
- `build_schedule:` α `list -> schedule` - Callback method that is invoked when asking the planner to build its final schedule .

The MPFL API provides a subclass of *planner* for each primitive plan type. These subclasses are all virtually identical. The only thing that differentiates them is the value of α which is the corresponding *planInstance* subclass. For example, the class *useModemPlanner* is simply an alias for a *useModemPlanInstance planner*, meaning the type variable α has been instantiated with type *useModemPlanInstance*.

5.7.8 The PI Evaluator Algorithm

Now that the class *planner* has been defined, the PI Evaluator's interaction with planners can be examined. The methods defined in *planner* are callback methods that are fired by the PI Evaluator at different stages throughout its execution. The PI Evaluator performs two traversals of the planner graph in order to do this: the first being a top-down, breadth-first traversal and the second being a bottom-up traversal in the reverse order of the first traversal as depicted in Figure 5.7.

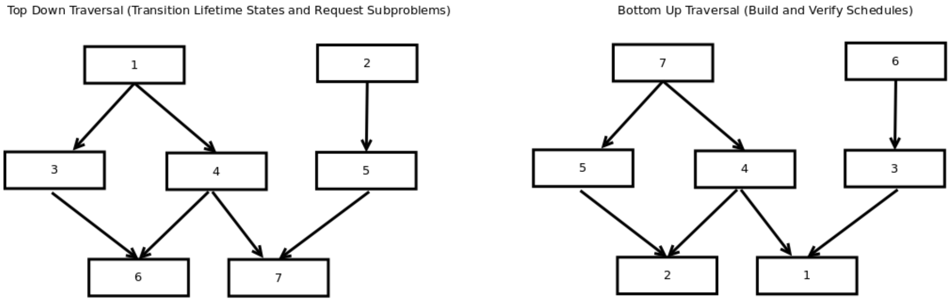

Top Down Traversal (Transition Lifetime States and Request Subproblems)
1
2
3
4
5
6
7
Bottom Up Traversal (Build and Verify Schedules)
7
6
5
4
3
2
1

Fig. 5.7 The two traversals performed by PI Evaluator on the planning graph

5.7.8.1 Top-Down Traversal

In the top-down traversal, the callback methods `on_ready_to_running`, `on_forcerun_to_running`, `on_running_to_complete`, and `on_ask_for_subproblems` are fired for each planner in the order listed. For the first three in the list, the PI Evaluator is asking each planner to make a decision about transitioning a set of plan instances that are in some state into another state. In the previous MPFL stage, the LST Evaluator attempted to transition as many plan instances as it could into the `On(READY)` state. Planners now must decide if they want to move those instances into a running (`On(RUN)`) state so that they can plan for them via the `on_ready_to_running` callback. The method is passed a list of *planInstance* objects as a callback parameter which are plan instances for that specific planner in the `On(READY)` state. To indicate to the runtime that they want to switch the state, the planner returns a list of `(planInstChain * bool)` pairs. The first entry in the pair refers to the plan instance (as a string in plan instance chain notation) to potentially be switched and the second a boolean set to true if the planner implementor wants to switch states. If not, the parameter is set to false and the plan instance remains in an `On(READY)` state. The same applies to `on_forcerun_to_running` and `on_running_to_complete` but from `On(FORCE_RUN)` to `On(RUN)` and `On(RUN)` to `End(COMPLETE)` respectively. For the forced-running to running callback, if the user indicates that the plan instance cannot go into a running state, the plan instance will be disabled as will any plan instances which caused the plan instance to go into the forced-to-run state. The running to complete callback is used to indicate to the MPFL runtime that a plan instance is complete means that the goal represented by the plan instance has been achieved. The final callback method `on_ask_for_subproblems` is invoked to give the planner an opportunity to create subproblems for its plan instances by forming an expression similar to the Do Expression in the MSL for each plan instance. The callback is passed all plan instances for the planner and must return a list of `(planInstChain * userPlanExp)` pairs. Each pair indicates the plan instance (in chain notation) to which to attach children and the second represents the children plan instances along with any planning operators, constraints, and conditional expressions encoded within a *userPlanExp*. If the planner returns an empty list, it indicates that the user either has no children plan instances or does not want to make a change to children attached in a previous compiler cycle. If previous children exist and the user creates new children, the old ones are detached from the plan instance tree, destroyed, and replaced by the new children. All return values of the callback methods are verified using the following rules:

- All plan instances referenced in the return values of each method are verified immediately and will raise an exception if not valid.
- The callback method can only refer to plan instances passed to it during invocation.
- For the first three callbacks that perform lifetime state changes, the planner must make a decision for each passed plan instance otherwise it will result in an exception being raised.

5.7.8.2 Bottom-Up Traversal

After the top-down traversal, the PI Evaluator invokes the final callback method for each planner in the exact reverse order of the top-down traversal (i.e. bottom-up). The method `build_schedule` asks the planner to build a schedule for all the plan instances it has in a running state. The planning algorithm is decided by the implementor of the planner and is the most difficult aspect of building a planner. There is no restriction to the algorithm, however it must have the following properties:

- It must be able to schedule tasks accounting for start time and end time.
- The start time and end time are described as a window of time rather than an exact time. The algorithm must make sure to obey these rules.
- The algorithm must be able to provide feedback when it fails giving indication of why something is not possible in terms of infeasibilities and/or conflicts.
- The algorithm must be decidable and take a limited amount of time, otherwise it will cause the MPFL engine to stall. Currently the compiler is implemented in a single thread of execution so a planner could potentially jam the system if any of its callbacks block.
- The algorithm only needs to return feasible schedules, but it is useful if the algorithm gives back optimal or suboptimal solutions that minimize resource usage. Algorithms that are designed to improve robot performance with each `build_schedule` invocation are ideal.

The planners are meant to build a schedule for the tasks in an running (i.e. `On(RUN)`) state. The planner can and should account for other plan instances, such as those in blocked (`Off(BLOCK)`) state, but does not necessarily have to as the bare minimum requirement is just to look at those in `On(RUN)`. Planners which are more *deliberative* versus *reactive* will take into account plan instances that will eventually be unblocked in the future or may be reactivated such as those in an `Off(DISABLE)` or `Off(SYS_RETRACT)` state. Planners have the ability to look at schedules for child planners in order to build their own schedules to take into account how children plan instances have been scheduled. Planners form their schedules by creating a list of *scheduleRow* values and enclosing them within a `Schedule` constructor. In the event there is an infeasibility or conflict, the constructors `ScheduleInfeasible` and `ScheduleConflict` can be used respectively passing error information in the constructor as mentioned earlier. If the user does not want to update the schedule in the case when nothing has changed, they can use the special `ScheduleNoUpdate` constructor which takes no parameters.

5.7.9 Schedules and Verification

The schedule generated by each planner consists of four entries: a time to issue the command, a time the command is expected to finish, the name of the plan instance associated with the command, and the actual command itself. The PI Evaluator verifies that each row in the schedule has the following properties:

- Each plan instance in each schedule row actually exists and is of the same primitive plan type as the planner.
- Each row has a start and end time that is within the overall time constraint of the plan instance.
- Each row refers to a plan instance that is in an `On(RUN)` state.

Failure to not meet these criteria results in raising an exception that will terminate MPFL. The last entry in each row represents a command which is simply a character string. MPFL does not dictate nor enforce any particular encoding of the command; the only requirement is that it is a string of characters. The client application must have an understanding of how the command is encoded and how it should be parsed. This was done to give planner implementors flexibility in their overall autonomy solution at the cost of not having any verification from the MPFL runtime.

5.8 Putting it All Together - The Basic MPFL Compiler engine

After the schedules are verified, they are passed back to the client application where the user can process them in order to issue commands to subsystems at the appropriate times. At this point, a single compilation cycle of the basic MPFL engine has completed. Now that the basic pieces of the MPFL compiler engine have been described, it can be summarized by the following algorithm.

1. Write a mission specification in the MSL as a text file.
2. Initialize the MPFL framework with a set of planners and the name of the text file.
3. The framework feeds the mission specification to the parser.
4. The Parser builds an abstract representation of the mission specification and hands it to the PIT Builder.
5. The PIT Builder validates the specification and generates the initial plan instance tree. The plan instance tree is then handed off to the LST Evaluator.

6. The LST Evaluator transitions the lifetime state of all the plan instances in the tree based on MPFL's internal transitioning rules. The plan instance tree is then passed off to the PI Evaluator.
7. (a) The PI Evaluator does a breadth-first, top-down traversal of the planners. For each planner:
        i. The planner is asked to transition `On(READY)` instances to `On(RUN)`, `On(FORCE_RUN)` instances to `On(RUN)`, and `On(RUN)` instances to `On(COMPLETE)`
        ii. The planner is asked for subproblems for its children. The subproblems must be of the same primitive plan type as the planner's dependency planners.
    (b) The PI Evaluator traverses all the planners in the reverse order and asks them to build a schedule and verifies the correctness of the schedules.
8. The schedules are passed back to the client application where they can be used to control the robot.
9. Goto 6.

5.9 The Complete MPFL Compiler/Runtime Engine

The previous section described the core stages of the MPFL compiler. However the issue of exception handling was not discussed which adds slightly more complexity. Figure 5.8 depicts the complete engine and an additional evaluator module called the *Exception Handler (EH) Evaluator*.

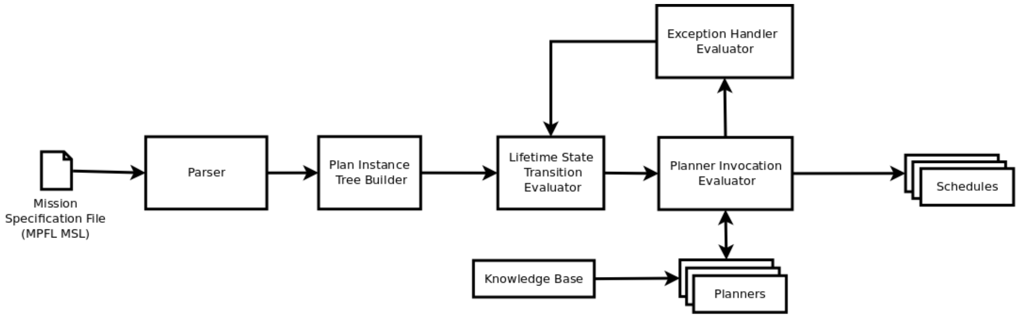

Exception Handler Evaluator
Mission Specification File (MPFL MSL)
Parser
Plan Instance Tree Builder
Lifetime State Transition Evaluator
Planner Invocation Evaluator
Schedules
Knowledge Base
Planners

Fig. 5.8 Complete MPFL Engine

5.9.1 Planners Raise Exceptions

Recall in the MSL there are exception handlers for infeasibilities (`OnInfeasible`) and conflicts (`OnConflict`). How these handlers are invoked by the MPFL compiler has not yet been discussed. The presentation begins with the *schedule* OCaml variant type:

```
type schedule = Schedule of scheduleRecord list
              | ScheduleInfeasible of (errorReason * planInstChain) list
              | ScheduleConflict of (errorReason * (planInstChain list)) list
              | ScheduleNoUpdate
              | ScheduleAutobuild;;
```

Infeasibilites and conflicts are determined by individual planners. If planners are given a set of plan instances to schedule for, and the planner cannot create a schedule, it can inform the MPFL compiler via the return value of the `build_schedule` callback method. If the problem is caused by a task(s) being infeasible (i.e. the task(s) are impossible to perform) then the planner informs the MPFL compiler of which plan instances are causing infeasibility. Sometimes the problem is not an infeasibility, but rather a scheduling conflict. For example, if a UseSonar planner has two plan instances that want to use the sonar at the same time and the two requests cannot be interlaced or separated, then the UseSonar planner can inform the compiler of the conflict via an API call. The planner is not limited to just two plan instances conflicting, if many instances are conflicting, the planner can inform the compiler of all of them. For the callback method `build_schedule`, the planner can indicate an infeasibility or conflict by using the `ScheduleInfeasible` and `ScheduleConflict` constructors respectively.

When building infeasibility or conflict errors with these constructors, planners have the ability to provide ***trace information*** by indicating not only the plan instances that caused the problem, but also why the error occurred in a natural language such as English. This enables one to have a high-level explanation of why problems occured when reviewing the robot's mission logs to see why it was not able to accomplish certain tasks. For example, a `UseAutopilot` planner may deem a particular `UseAutopilot` plan instance infeasible because it cannot make it to the destination within its time constraints. The planner implementor, when informing the compiler of the infeasibility, can also pass a string that says something along the lines of *"Infeasible because planner cannot make it to destination within specified time constraint"*. This is extremely useful in debugging and also gives insight into how the robot *thinks*.

5.9.2 Handling Errors via the Exception Handler (EH) Evaluator

When the `build_schedule` function returns a value of `ScheduleInfeasible` or `ScheduleConflict`, the PI evaluator is aborted and a special evaluator, the ***Exception Handler (EH) Evaluator*** is invoked with the error as a parameter. The handler does the following:

- The plan instances returned in the error must refer to valid plan instances and are validated by the EH Handler.
- The EH Handler checks for the nearest handler within scope. The handlers are encoded within each `ExecutePlan` plan instance. Each handler's case signature is matched against the plan instance name based on the rules from Chapter 4 traversing from the erroneous plan instance up to the root sortie plan instance. If a match occurs, the handler code is extracted. With conflict handlers, handlers are

searched for each conflicting plan instance. The handlers are searched in breadth-first fashion with the first matching handler returned.

- The EH Handler evaluates the handler code, which is encoded as a *handlerExp* mentioned earlier in the chapter. The result of the evaluation is the application of a lifetime state of `Off(DISABLE)` or `Off(RETRACT)` to a subtree within the plan instance tree via an invocation of the LST Evaluator. The subtree root is the plan instance containing the matched handler. Control is then returned to the PI Evaluator which reattempts the bottom-up traversal starting at the lowest planner in the planner graph which has had a change in one of its plan instances. It may take several invocations of the EH Handler to create a schedule, but eventually the PI Evaluator will finish. In the worst case, all plan instances may end up being either disabled or retracted (i.e. nothing can be scheduled) before the PI Evaluator can finish.

5.10 Replanning As Recompilation

The MPFL engine is intended to be constantly reinvoked via the `build_schedule` call which results in a new set of schedules. The lifetime state is all the dynamic information the runtime requires to manage plan instances. This simple state allows the compiler to mark tasks throughout their lifetimes and allows planners to have metainformation about the tasks for its scheduling algorithms. One analogy is to think of airline departure and arrival schedules at an airport. Periodically the schedules refresh with updated information. Planes that were once in an *On Time* state can quickly switch to a *Delayed* status and then back again.

5.11 Segue: Building an Autonomous Robot with MPFL

This chapter described the inner-workings of the MPFL compiler/runtime while simultaneously showing how the API is utilized. One can now see that the core MPFL compiler/runtime really does not perform the actual planning, but rather provides guide rails for implementing a distributed hierarchical scheduling system with lots of verification, a powerful language to express problems, and metainformation to help schedule. Though the internals of MPFL may be a bit confusing, the ease of use of MPFL is made much clearer with an example. The next chapter details an example system that implements both a knowledge base and a set of planners in order to provide a complete example of an autonomous system robot MPFL.

Chapter 6

A Complete Demonstration System

In order to demonstrate how to use MPFL and to test the prototype framework, a demonstration system was created using MPFL. A demonstration set of planners and a knowledge base were created. A simple MPFL client application was created which controls a simulated robot. One can see what the robot is doing in real-time on a map display.

6.1 Demonstration System Architecture

The system consists of 3 components:

- *Demo Client* - The application that uses MPFL to autonomously control a robot.
- *AUV Simulator* - A simple, real-time simulator for an autonomous underwater robot.
- *Map Display* - A geographical map display showing the autonomous vehicle position and orientation as well as other mission data.

Figure 6.1 shows a system data flow diagram between these components. The following sections describe the implementations of these components.

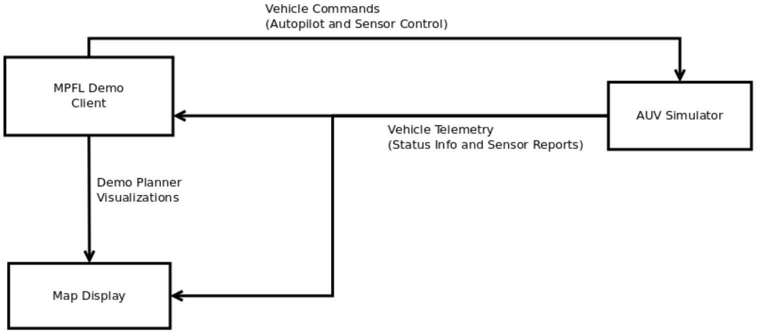

Vehicle Commands
(Autopilot and Sensor Control)
MPFL Demo Client
AUV Simulator
Vehicle Telemetry
(Status Info and Sensor Reports)
Demo Planner Visualizations
Map Display

Fig. 6.1 Data flow block diagram of MPFL demo

6.2 Demo Client Application

MPFL requires a client application to link against and call the MPFL compiler. The client application bootstraps the system with a set of planners and a knowledge base. Figure 6.2 depicts the internals of the demo client as a system block diagram. The planners and knowledge base are subclassed from the API provided base classes. The demo client is very simple and described the following algorithm:

1. Initialize MPFL with the `initialize_mpfl` API call passing in the MSL file, knowledge base, and planners.
2. Invoke the `build_schedules` API call.
3. Parse returned schedules and issue commands as dictated by the schedule.
4. Goto 2.

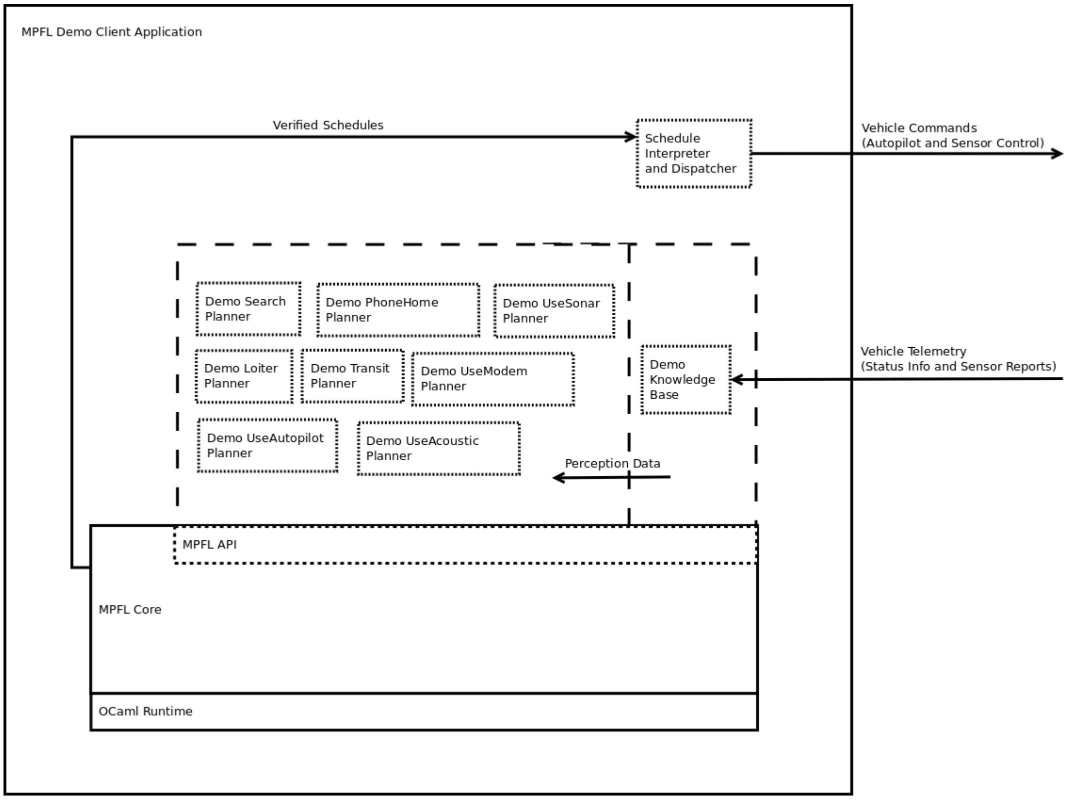


Fig. 6.2 Internals of MPFL demo client

6.3 AUV Simulator and Map Display

To demonstrate MPFL, a simple simulator of an AUV was created encompassing basic kinematics, sensing, and power usage. A map display is used to show what the AUV is doing as well as portraying additional information about what the AUV is *thinking* based on its current set of schedules.

6.4 The Planner Hierarchy

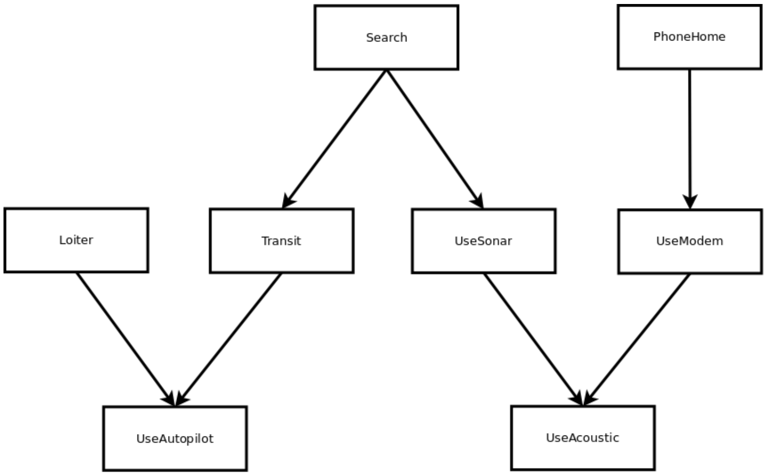


Fig. 6.3 Example planner hierarchy

Figure 6.3 depicts the planner graph for the demonstration system. Recall that the purpose of the hierarchy is to allow planners to break its own problems (encoded as plan instances) into smaller problems for its children planners. The subsequent sections will describe the implementation of each demonstration planner and define the following planner aspects:

- Callback `on_ready_to_running`
- Callback `on_forcerun_to_running`
- Callback `on_running_to_complete`
- Callback `on_ask_for_subproblems`
- Schedule Encoding
- Callback `build_schedule`

6.5 Demo `UseAutopilot` Planner

The `UseAutopilot` planner poses a difficult problem where each plan instance represents a single waypoint. The waypoint must be reached within the end window and must not be attempted at any time outside the start window. The scheduling algorithm must choose a sequence of waypoints, calculating when to leave and when to arrive. A valid solution to the problem must not violate these constraints otherwise it should cause an infeasibility or conflict error. This problem is very similar to the *traveling salesman problem*, meaning that the best known algorithm to find the most optimal path is in computational complexity class NP. However we do not require the optimal path, but rather just a feasible path that meets all constraints. It would be ideal if the feasible path was optimal or as close to optimal as possible so as to reduce resource usage. A simple genetic algorithm that runs in worst-case $O(n^2)$ time was employed to search for a feasible path that can meet all constraints. The algorithm iteratively improves the best known solution, even if currently infeasible. This is ideal as the planner is able to schedule once a feasible solution is found, but can continue to improve the schedule quality as the robot's mission progresses.

6.5.1 Callback `on_ready_to_running`

The example planner has all the ready plan instances go into a running state. The scheduler attempts to schedule all problems immediately so it makes sense to do this.

6.5.2 Callback `on_forcerun_to_running`

All plan instances that were forced to run (likely due to a parallel (||) operator in the mission specification) are also switched to a running state. If the time constraints for each of these instances is within their start window, they are marked to go into a running state. Those that violate the constraint are not changed and will be disabled by the framework.

6.5.3 Callback `on_running_to_complete`

As each plan instance represents a waypoint, the first waypoint in the last generated schedule is the one that is currently active as visiting waypoints is a serial operation (i.e. you must visit one waypoint before going to the next). If the current vehicle position (taken from the knowledge base) is within an acceptable threshold distance of the waypoint and the current time is within the end window, it is marked as complete indicating a goal has been achieved. Otherwise the plan instance remains in the running state as do the remainder of plan instances.

6.5.4 Callback `on_ask_for_subproblems`

As the `UseAutopilot` planner is a leaf in the planner hierarchy, cannot produce any subproblems, so the return value of this call is always an empty list.

6.5.5 Schedule Encoding

The schedule is encoded as follows:

- **Start Time** - The time to start heading for the waypoint.
- **End Time** - The time the robot is expected to arrive at the waypoint.
- **Name** - The name of the plan instance associated with the waypoint.

- **Command** - A simple string indicating a latitude, longitude, and depth (e.g. `"Lat = 32.0, Lon = -122.0, DepthMeters = 10.0"`).

6.5.6 Callback `build_schedule`

This callback implements the genetic algorithm that performs scheduling. The goal of the algorithm is to create an ordering of all waypoints specified by each plan instance in the `On(RUN)` state. The algorithm must calculate not only an order, but determine the time at which the robot leaves the waypoint and goes to the next while staying within its time constraint. In order to do this, it must also predict when it expects to leave each waypoint and go to the next.

6.5.6.1 Representing and Rating a Solution

Each potential solution to the problem is an order of waypoints $< w_1, w_2, ..., w_n >$. We also introduce a special waypoint w_0 that represents the current position. In addition to the ordering, the solution must encode what time to leave from each waypoint to arrive at the next and at which speed. A simple way to reason about the problem is to build a table like the one depicted by Table 6.1. Each row represents a waypoint. The table is built row by row, each row depending on the previous. The first row refers to the current position, so the arrival time is whatever the current time is. The departure time is then calculated based on the start time constraint of the subsequent waypoint. If the start window for the next waypoint has not yet arrived, the AUV waits at its current position and heads for the waypoint at the start time of that window. If the start window is happening now, the departure time is set to the current time. If the window was in the past, the departure time is set to the current time as well (i.e. leave immediately even though we are running late). The arrival time for the next row is represented by the following function:

$$arrivalTime_i = departureTime_{i-1} + distanceToNextPoint_{i-1}/maxSpeedOfRobot$$

Waypoint	**Arrival Time**	**Distance to Next Point**	**Departure Time**	**Departure Speed**	**Slack/Tardy**
w_0	10:00 PM	1000.0	10:00 PM	0.5	0
w_1	10:30 PM	500.0	10:45 PM	0.5	60
w_2	11:00 PM	1500.0	11:10 PM	0.5	100
w_3	11:55 PM	250.0	1:00 AM	0.5	-50
w_4	1:08 AM	0.0	∞	0	30

Table 6.1 Determining Solution Fitness

The table also contains fields for departure time and departure speed. The algorithm always attempts to have the robot depart as soon as possible: the maximum of the current time and the beginning of the start window of the next waypoint. The robot then heads at maximum speed towards the next waypoint. If the robot arrives within the end window of the next waypoint, the slack/tardy value in the table is filled with the time remaining in the window indicating how early the robot is (i.e. the slack). If the robot is late and missed the end of the end window, it is assigned a negative value with the amount of time it late by (i.e. tardiness). If the robot arrives before the beginning of the next time window, the speed is reduced so that the AUV arrives at the beginning of the end window of the target waypoint.

The table can then be used to calculate the ***fitness of the solution***. In other words, the table tells us a lot about the ordering of waypoints. Anywhere there is a negative value in the slack/tardy box, it means the solution is infeasible. If all are greater than or equal to zero, it means the solution is feasible.

6.5.6.2 Comparing Solutions

However, some feasible solutions are better than others. This also holds for infeasible solutions, some are worse than others. The slack/tardy value can be utilized to give a goodness of the solution by calculating the *total slack* and *total tardy values* as defined by the following equations where i refers to the slack/tardy value:

$$TotalSlack = \sum i, \forall i > 0$$
$$TotalTardy = \sum i, \forall i < 0$$

A solution which has a total tardy less than 0 means it is infeasible, whereas all the remaining are feasible. For feasible solutions, the ones with more slack can be considered *better* than the others, whereas with infeasible solutions, the lower number (i.e. less negative) solutions are *not as bad* as the ones lower than it.

6.5.6.3 Finding the Solution - Genetic Algorithm

There are $n!$ ways to arrange the waypoints, so in the worst case any algorithm would be $O(n!)$. However this is very bad and would not work very well beyond a dozen or so waypoints. What is worse is that if all solutions are infeasible, we might need to remove waypoints, giving a total complexity of $O(^nC_n +^n C_{n-1} + ... +^n C_0)$ which is $\leq O(3n!) = O(n!)$. A simple genetic algorithm to perform a heuristic base search can be employed to get around this.

A genetic algorithm works by representing each solution as a ***member*** of a ***population*** of solutions. The algorithm is bootstrapped by creating a population of some size n where the solutions are randomly generated, in this case random orderings of waypoints. The fitness of each solution is based on the algorithm mentioned earlier. All solutions have their fitness rated. The fittest members of the population are chosen to live and *reproduce* whereas the others are *killed* off. The percentage that gets to live is an empirically determined number (10-15% is a good starting point). The remainder of the population is then regenerated in a process called ***breeding*** by randomly picking members of the surviving population and *mutating* a copy of them to form a new ***offspring***. The mutation in this algorithm simply swaps two random waypoints. The breeding continues until the population is restored to its original size. Typical genetic algorithms also employ a *crossover* operation which have breeding solutions exchange parts of themselves to create members of the new population, but that is not utilized as that is difficult to employ with this problem and not really necessary.

6.5.6.4 Performance

The genetic algorithm runs in $O(n^2)$ time. After generating a population, for each solution the fitness information can be determined in linear time ($O(n)$). The solutions then have to be sorted by fitness scores which we know in the worst case can be ($O(n^2)$). Regenerating the next population from the remainder is also a $O(n)$ operation resulting in an overall complexity of $O(n^2)$. Genetic algorithms can quickly converge to a feasible solution if the feasible space is large. The smaller it gets, the more time it will likely take to find the solution, eventually to the point where it is as poor as brute force. However, in practice if the plan specification in the MSL is reasonable, the algorithm does quite well.

6.5.6.5 Failure to Converge

As mentioned in the previous chapter, the callback functions should not block the MPFL engine. The algorithm should run either in a thread or run for some number of generations and then stop. In the actual implementation, the latter is chosen. The algorithm runs for a set number of generations and if it does not converge, it means there's an infeasibility somewhere. One waypoint is thrown away from the set of waypoints based on the tardy information (i.e. the waypoint with the worst tardy value is thrown away) and the algorithm runs again. The algorithm keeps removing waypoints till a solution is found or all waypoints are removed. Once a solution is found, the `build_schedule` call returns a schedule infeasibility passing the names of the instances that were removed to get a feasible solution. This will invoke the error handling potentially disabling or retracting plan instances.

6.5.6.6 Accounting for Blocked Instances

The genetic algorithm is performed not only on instances in the `On(RUN)` state, but also those in the `Off(BLOCK)` state. When a schedule is formed, all the `Off(BLOCK)` states are removed from the schedule and the `On(RUN)` instances are bumped up the list. The idea is that if any of the instances become unblocked, there is a good chance that it will be able to achieve the goal as the detour for the newly unblocked instances will be minimal. This may not necessarily hold true, but in practice it was found this heuristic works quite well. The ability to account for blocked instances means that the planning is more deliberative and the planner can handle scheduling operations well, especially for heavy uses of the serial (>) operator which causes blocking.

6.5.6.7 Improvement over Time and Solution Caching

As genetic algorithms retain the best solutions, the schedule chosen can only improve. The best solution is chosen for each potential length of the waypoint list of sizes n down to those of size 1 and are included as members of the initial population for the next `build_schedule` in addition to the randomized solutions. Even if waypoints are added and removed, the algorithm will continue to work.

6.6 Demo `Transit` Planner

The `Transit` planner differs from the `UseAutopilot` planner as each of its plan instances represents a list of waypoints that are visited sequentially rather than a single waypoint. This can easily be implemented by utilizing the `UseAutopilot` planner defined before which is a child planner in the planner graph.

6.6.1 Callback `on_ready_to_running`

This callback leaves plan instances as `On(READY)`. The `UseAutopilot` planner will handle switching lifetime state.

6.6.2 Callback `on_forcerun_to_running`

In this callback, the planner opts to set the plan instances to an `On(RUNNING)` state. If there is any infeasibility it will occur in the child planner.

6.6.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseAutopilot` children instances complete.

6.6.4 Callback `on_ask_for_subproblems`

One can create a set of `UseAutopilot` plan instances to represent each waypoint in the list of waypoints. The *userPlanExp* for each `Transit` plan instance with waypoints $< w_1, ..., w_n >$ can be formed by using the serial operator on each `UseAutopilot` plan instance as follows:

$$w_1 > w_2 > ... > w_n$$

Below is the equivalent code in OCaml code using the MPFL API. The `Destination` value is omitted but should contain the respective position in practice. One can keep nesting `Op` constructors with the `SERIAL` value as the operator type.

```
Op(SERIAL, Op(SERIAL, PlanInst("w_1", UseAutopilot(Destination = ...),
              PlanInst("w_2", UseAutopilot(Destination = ...))),
           ...
           PlanInst("w_n", UseAutopilot(Destination = ...)))
```

The planner should only create these children problems on bootstrap or if during runtime the set of `UseAutopilot` plan instances changes. The latter can be detected easily via an API call.

6.6.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time to start heading to the first waypoint.
- **End Time** - The time the robot arrives at the last waypoint.
- **Name** - The name of the plan instance associated with the waypoint list.
- **Command** - A command is not needed here, the value is just set as *"Perform Transit"*. However encoding the list of waypoints may be useful.

6.6.6 Callback `build_schedule`

As all the work is done by the `UseAutopilot` planner. The `Transit` planner can access the schedule returned by the `UseAutopilot` planner to build its own schedule. For each `Transit` plan instance, one can look at the smallest start time and largest end time of all its children instances in the `UseAutopilot` schedule. For each `Transit` instance a row can be added to the schedule. This can be automated using the `ScheduleAutobuild` return value.

6.6.7 Performance

The performance is based on that of the `UseAutopilot` planner, the complexity of the path, and the tightness of all constraints. Most of the plan instances will be blocked because of the user of the serial (>) operator. However, as the `UseAutopilot` planner accounts for blocked instances, the planner performs reasonably well. In fact it can intertwine two paths if they overlap.

6.7 Demo `Loiter` Planner

The `Loiter` planner also utilizes the `UseAutopilot` planner like the `Transit` planner. A `Loiter` is simply going to a waypoint with a departure constraint.

6.7.1 Callback `on_ready_to_running`

All are activated to be in the `On(RUN)` state

6.7.2 Callback `on_forcerun_to_running`

All are activated to be in the `On(RUN)` state

6.7.3 Callback `on_running_to_complete`

The `UseAutopilot` planner will implicitly complete the loiter once the waypoint is completed.

6.7.4 Callback `on_ask_for_subproblems`

For each `Loiter` plan instance, we need to create a single `UseAutopilot` plan instance. A loiter problem not only has a destination, but a loiter duration. The `Loiter` planner has access to the `UseAutopilot` planner's schedule, so it simply looks at the arrival time for the plan instances it created and creates a time constraint adding the loiter duration time to it.

6.7.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time to start heading to the loiter point.
- **End Time** - The time the robot leaves the loiter point.
- **Name** - The name of the plan instance associated with the loiter task.
- **Command** - A command is not needed here, the value is just set as *"Perform Loiter"*.

6.7.6 Callback `build_schedule`

The schedule for the `Loiter` plan instance is based on the start and end time of each `UseAutopilot` plan instance. One extracts all entries from the child planner's schedule and maps each child instances start and end times to the ones used in each `Loiter` schedule row. Each row uses the parent `Loiter` plan instance name as the identifier for each row. Again, the `ScheduleAutobuild` option can be used to do this automatically.

6.8 Demo `UseAcoustic` Planner

The `UseAcoustic` planner has the responsibility of scheduling use of the acoustic communication medium (i.e. the water) for acoustic devices such as sonars and acoustic modems. Each `UseAcoustic` plan instances defines the acoustic problem as a frequency band, the duration of time required for an acoustic ***pulse*** (i.e. a chunk of acoustic time), the minimum spacing gap between the pulses as a duration, the maximum spacing gap between pulses, and the total number of pulses needed.

6.8.1 Callback `on_ready_to_running`

All instances are set to a running state.

6.8.2 Callback `on_forcerun_to_running`

All force running tasks are set to running as well.

6.8.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseAutopilot` children instances complete.

6.8.4 Callback `on_ask_for_subproblems`

This is a leaf planner so no subproblems are generated.

6.8.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time the first acoustic pulse is issued
- **End Time** - The time the last acoustic pulse is issued
- **Name** - The name of the `UseAcoustic` plan instance associated with the row
- **Command** - The command encodes a frequency band, the start time of the first pulse, the start time of the last pulse, the spacing between pulses, and the duration of each pulse

6.8.6 Callback `build_schedule`

A simple ***linear program*** can be used to build the schedule. A linear program is an optimization technique for an optimization problem with a single objective function and a set of constraints which are defined as a set of inequalities. The constraints must be linear. The program can then be solved utilizing a linear program solver, such as the ***simplex method***. For sake of brevity, the algorithm is not defined. However, the approach is quite straightforward: model each discrete block of acoustic time as a set of variables in the linear program. Each block can be defined by variables indicating start time and end time. Constraints can then be developed around these variables to ensure constraints are met (e.g. minimum and maximum gap constraints).

Additionaly a similar approach to the `UseAutopilot` planner can be used where a genetic algorithm searches for a feasible schedule. Other *evolutionary algorithms* such as *swarm optimization* and *simulated annealing* would also likely work.

6.9 Demo `UseSonar` Planner

The `UseSonar` planner activates an ***active sonar*** device or polls a ***passive sonar*** for sensing. Active sonar determines how far away objects are by emitting a sound pulse (called a ***ping***) and measuring the amount of time it takes for the sound to return in order to determine range. The further an object is, the more time it takes to receive the reflected ping. The ***ping rate*** of a `UseSonar` instance determines how quickly the user wants to ping. As the sonar is competing with other acoustic devices such as additional sonar sensors or an acoustic modem, it must make sure not to interfere with those devices. The `UseAcoustic` planner can be used by acoustic devices to allocate chunks of acoustic bandwidth for such a purpose.

6.9.1 Callback `on_ready_to_running`

All instances are set to a running state.

6.9.2 Callback `on_forcerun_to_running`

All force running tasks are set to running as well.

6.9.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseAcoustic` children instances complete.

6.9.4 Callback `on_ask_for_subproblems`

For each `UseSonar` instance, we want to create a `UseAcoustic` instance where the acoustic band matches that of the sonar, the block duration size is however long the sonar needs to ping and receive (active case) or to poll the sensor (passive case), and the spacing between blocks (min gap and max gap) are set to appropriate values depending on sonar type. The number of pulses is based on how long the search needs to be conducted for.

6.9.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time to issue the first ping or poll
- **End Time** - The time the last ping or poll is finished
- **Name** - The name of the plan instance associated with the `UseSonar` request
- **Command** - The command encodes a frequency band, the start time of the first ping/poll, the start time of the last ping/poll, the spacing between pings/polls, and the duration of each ping/poll

6.9.6 Callback `build_schedule`

Building the schedule is quite easy. As the `UseSonar` planner has access to the `UseAcoustic` schedule, it can inspect that schedule and determine at which times the acoustic channel is allocated to it under what conditions. The command within the `UseAcoustic` schedule is almost identical to that in the `UseSonar` schedule with the only difference being terminology (e.g. calling it a ping/poll instead of a pulse). Again the `ScheduleAutobuild` option may be used to automate this.

6.10 Demo `UseModem` Planner

The `UseModem` planner is used for scheduling the use of an acoustic modem. An acoustic modem is similar to a dial-up modem found commonly in personal computers, rather than modulating electromagnetic frequencies over the voice channel of a telephone line, it creates pulses of sound encoding digital data that travel through water. The `UseModem` planner is very similar in operation to the `UseSonar` planner in that it requires the use of the `UseAcoustic` planner as well so as not to interfere with any other acoustic devices. Each `UseModem` plan instance encodes the name of the modem to use (in the event multiple modems exist) and message which is a simple string of text (8-bit ASCII encoded).

6.10.1 Callback `on_ready_to_running`

All instances are set to a running state.

6.10.2 Callback `on_forcerun_to_running`

All force running tasks are set to running as well.

6.10.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseAcoustic` children instances complete.

6.10.4 Callback `on_ask_for_subproblems`

For each `UseModem` instance, we want to create a `UseAcoustic` instance where the acoustic band matches that of the modem. The block duration size is however long the modem needs to transmit its message and receive acknowledgment. Calculating the one-way transmit time is based on the size of the message in bits divided by the predicted bit rate of the modem[1]. The min gap is set to zero and the max gap is set to whatever tolerance is appropriate for the `UseModem` planner. The number of pulses is equal to the number of retries the user is willing to go for in the event one fails.

6.10.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time of the first message send attempt.
- **End Time** - The time the last message send attempt finishes.
- **Name** - The name of the plan instance associated with the `UseModem` request.

---

[1] The predicted bit rate is a function of several variables including distance to receiver, environmental noise, concentration of impurities in the water, and depth.

- **Command** - The command encodes a frequency band, the start time of the first message transmit attempt, the start time of the last retransmission attempt, the spacing between retransmission attempts, and the duration of time available for each transmission attempt.

6.10.6 Callback `build_schedule`

Building the schedule is quite easy and similar to the way the `UseSonar` schedule is built. As the `UseModem` planner has access to the `UseAcoustic` schedule, it can inspect that schedule and determine at which times the acoustic channel is allocated to it under what conditions. The command within the `UseAcoustic` schedule is almost identical to that in the `UseModem` schedule with the only difference being terminology (e.g. calling it a *transmission attempt* instead of a *pulse*). Again the `ScheduleAutobuild` option can be used to automate this.

6.11 Demo `PhoneHome` Planner

The `PhoneHome` planner is a way to send status and sensory information of the vehicle back to some operating platform (e.g. the launch platform or an onshore command center). In our planner hierarchy, the `PhoneHome` planner utilizes the `UseModem` child planner to achieve this.

6.11.1 Callback `on_ready_to_running`

All instances are set to a running state.

6.11.2 Callback `on_forcerun_to_running`

All force running tasks are set to running as well.

6.11.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseModem` children instances complete.

6.11.4 Callback `on_ask_for_subproblems`

The `PhoneHome` planner creates a child `UseModem` plan instance for each of its own plan instances. Each plan instance encodes the name of the modem to use and the rate at which to report home.

6.11.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time to start phoning home
- **End Time** - The time the phoning home operation ends

- **Name** - The name of the plan instance associated with the `PhoneHome` request
- **Command** - No command is needed, so each entry just says "Performing PhoneHome operation"

6.11.6 Callback `build_schedule`

The `PhoneHome` planner schedule is built from the `UseModem` planner's schedule. The start time and end time correspond to the start time of the first entry referencing the child `UseModem` instance in the `UseModem` schedule. The end time similarly refers to the last schedule entry's end time. Again the `ScheduleAutobuild` option can be used to automate this.

6.12 Demo `Search` Planner

The final example planner in the system is the `Search` planner. What makes this planner more interesting than the others is that it is the only one to use more than one child planner, in this case `Transit` and `UseSonar`. The search planner's duty is to traverse some giving area and look for objects of interests. Objects of interest may include other vessels, mines, surface ships, or the seabed (e.g. bottom mapping). The traversal is done via the use of child `Transit` plan instances and the scanning is done with child `UseSonar` plan instances.

6.12.1 Callback `on_ready_to_running`

All instances are set to a running state.

6.12.2 Callback `on_forcerun_to_running`

All force running tasks are set to running as well.

6.12.3 Callback `on_running_to_complete`

The plan instance will automatically turn to complete when all the `UseSonar` and `Transit` children instances complete.

6.12.4 Callback `on_ask_for_subproblems`

The `Search` problem is specified as an area, a lane width, and sensor. The example planner assumes the area to be a rectangle. If it is not specified as such in the MSL, a rectangular hull is calculated using the MPFL API. The rectangle is broken up into strips. The strips are oriented horizontally if the width of the rectangle is longer than the height, and vertically in the reverse case. If the area is square, the strip orientation can be chosen arbitrarily. A set of waypoints is calculated that represent a ***lawnmower search path***, where each strip is traversed down the center from end to end, then the robot turns, moves into the next strip, and starts moving in the reverse direction (i.e. like a lawnmower) (Figure 6.4). These waypoints are encoded within a single `Transit` plan instance, where each `Search` plan instance has a `Transit` instance encoding the search path. As for sensing, for each `Search` plan instance we also create a `UseSonar` plan instance specifying parameters appropriate to the type of search the planner is built for.

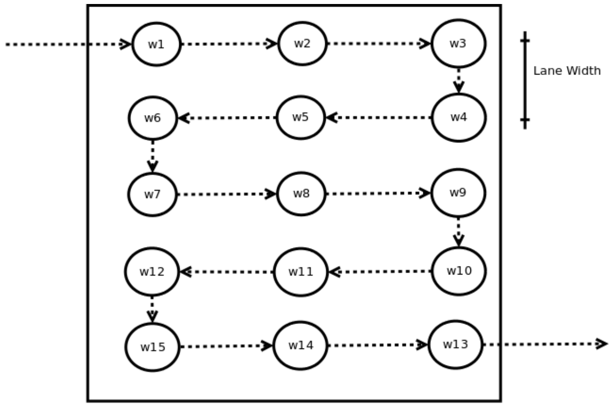


Fig. 6.4 Demo `Search` planner creates waypoints for `Transit` planner

The two tasks depicted by each `Transit` and corresponding `UseSonar` plan instance must happen in parallel. Hence the subproblem for each `Search` plan instance is represented as:

$$t \mid\mid us$$

where t is the name of the `Transit` plan instance and us is the name of the `UseSonar` plan instance.

6.12.5 Schedule Encoding

The schedule encoding is as follows:

- **Start Time** - The time the search commences.
- **End Time** - The time the search ends.
- **Name** - The name of the plan instance associated with the `Search` request.
- **Command** - No command is needed, so each entry just says *"Performing search operation"*.

6.12.6 Callback `build_schedule`

The `Search` schedule row entries map to the rows refereing the children `Transit` instances within the `Transit` schedule. Each entry in the `Search` schedule has a start time/end time corresponding to the start time/end time of the transit operation. Again the `ScheduleAutobuild` option can be used to automate this.

6.13 Segue: Results, Performance, and Related work

This chapter concludes a description of the planners that constitute a demonstration of a total planning autonomy solution for an AUV. Unfortunately due to time, not all of the planners could be implemented fully, though they will be implemented in the future. However, algorithms were given for each planner.

Chapter 7

Results, Related Work, Future Work

This final chapter discusses the outcomes of this research.

7.1 Results

The implementation of MPFL yielded some very interesting results and some fresh perspectives on autonomy software development. During the course of the research, a prototype MPFL compiler was developed in OCaml along with a set of prototype planners and knowledge base as described in the previous chapter.

7.1.1 Mission Performance

This prototype system demonstrated deliberative autonomous control of a simulated AUV via a specification written in the MSL without violation of constraints, even in complex scenarios. In general, the AUV not only did not violate any constraints, but performed optimally (or at least close to it) by minimizing use of resources (time and power) thanks to the heuristics implemented in the planners. In the cases of failures to find feasible schedules, MPFL's exception handling mechanisms gracefully allowed the engine to continue. It was interesting to watch the virtual robot do its best to complete tasks and the decisions it had to make when it could not attempt certain tasks.

7.1.2 Computational Performance

The simplicity of MPFL results in negligible performance overhead from the framework itself. The computational burden is primarily dictated by the performance of each planner's scheduling algorithms. The algorithms used in the prototype example planners all run in worst case polynomial time. The most complex algorithms reside in the `UseAcoustic` and `UseAutopilot` planners where planners iteratively solve their problems, where the number of iterations per compilation cycle are limited.

7.1.3 Ease and Quality of API and Framework

In software engineering, developing good software frameworks and their corresponding APIs can be extremely difficult. Though what makes an API and framework *good* is subjective, generally desirable features are:

- **Simplicity** - Minimizing the set of functions/methods, classes, and types the user has to learn so as not to confuse or overwhelm them.

- **Consistency** - The API should have a consistent feel so that once the user starts learning a part it, they can intuitively figure out the rest as it ‘follows the same formula’.
- **Match Implementation Language Style** - If the API is implemented in a functional language, the API should have functional style versus another style such as imperative.
- **Use Implementation Language Well** - The API should take advantage of the features of its implementation language. For example, if the language is statically typed, the API should use types extensively in API functions and callbacks to provide static verification.
- **Modular** - The API and framwork should be broken into pieces that can be reasoned about in a piecemeal fashion so as to make it easier for the end user to learn and understand. These pieces can come through the abstraction and encapsulation constructs provided by the implementation language such as files, classes, functions, and abstract data types.
- **Documentation** - A good API should be well documented, with descriptions for all types and functions, preconditions, and caveats. The names of functions/methods and types should be simple and self-documenting, meaning that the name implies what the function/method/types does.
- **Verification and Feedback** - All user interactions with the framework should be verified, with static verification preferred over runtime verification. In the event of errors, the user should be informed in the simplest and clearest way possible without sacrificing crucial details.
- **Default Behavior** - Good software frameworks allow users to take off quickly by providing default behavior with good presets. For example, in most widget toolkits, the user does not have to define the behavior of what clicking the close button on a window does — it simply exits the program. However, the user can override this behavior by providing an alternative handler callback function to change that behavior.
- **Isolation of API/Framework and End User Application** - A good software framework hides its implementation details from the user so that they do not have to understand how it works when building their application, nor can they modify the code of the framework and API.

MPFL was designed with all these qualities in mind. The extent to which the prototype implementation bears these qualities is subjective. However, each one is discussed below and the approach taken to incorporate those ideals into MPFL.

7.1.3.1 Simplicity

The end user only has to learn the MSL and two global functions (`initialize_mpfl` and `build_schedules`) assuming that they are not building any of the planners or knowledge base themselves. The MSL itself was designed to be minimal as can be seen from previous chapters. For users that have to write the planners and knowledge base, they additionally have to know three class hierarchies (α `planner`, `knowledgeBase`, `planInstance`, and a little over a dozen variant types used to encode plan instance information such as problem specification and lifetime state.

7.1.3.2 Consistency

The constructs in the API follow a consistent style in naming conventions. The MSL itself uses a consistent style everywhere, most notably the arguments to constructors which follow the <`fieldName`> `=` <`value`> formula. The types and values in the MSL match those found in the API. For example, plan instances in the MSL map one-to-one to the plan instances used in the planner API defined by the `planInstance` class hierarchy.

7.1.3.3 Match Implementation Language Style

The MPFL compiler internals were implemented in a functional style as OCaml is primarily a functional language. The API uses functional style, but also incorporates object-oriented constructs which does not break the style as OCaml is also an object-oriented language. The imperative features of OCaml were rarely used and the majority of the code is written in a pure functional style (i.e. no side-effects) making it easier to reason about semantics and code correctness. The API is purely functional meaning that the user cannot corrupt system state via side-effect unlike languages such as C.

7.1.3.4 Modular

One of the key goals of MPFL was to make it highly modular with as few dependencies as possible between modules. This allows for reusability of components as well as making it easier to reason about system design as it can be done in a more piecemeal fashion. The components that users build are planners and knowledge bases. These modules are decoupled from each other. Though planners may create subproblems for other planners, the implementation of those planners does not matter. The user is free to couple modules together if their application requires it, though coupling should be minimized.

7.1.3.5 Documentation

It is difficult to use any API without documentation. In MPFL, the implementation and API source code have all functions and types annotated with documentation. The documentation uses OCaml's documentation system enabling interactive help with IDEs that can pop up help information as functions, methods, and classes are used.

7.1.3.6 Verification and Feedback

If there is any philosophy that is at the core of MPFL's design, it is that everything must be verified and the user must be given meaningful feedback when problems occur. Every level of MPFL – the MSL, the compiler/runtime, and the API – verify as much information as possible to ensure correct operation of the robot. Missions described with MSL are guaranteed to be sensible to the robot thanks to the static and runtime checks performed by the compiler. If there are errors in the specification, MPFL will return detailed information with line number containing the violating code. The compiler/runtime is also constantly checking its own internal state to ensure that the implementation is correct and will inform the user if there is a bug in the implementation. Finally, when users build planners and knowledge bases, they have to follow a design contract dictated by the abstract base class definitions and enforced by the OCaml type system.

7.1.3.7 Default Behavior

It is quite easy to immediately start writing code with MPFL as there are default implementations of all callback methods the user has the ability to override. One can start with a set of *empty* planners where callbacks return empty values and slowly build up their components.

7.1.3.8 Isolation of API/Framework and End User Application

It is important for a good API to hide its internals from the user as well as stop the user from corrupting internal state of the compiler/runtime. In MPFL, the user does not have to have an understanding of how the internals of the compiler/runtime (i.e. Parser, LST Evaluator, PI Evaluator, etc) work. All they have to understand is the planner and knowledge base API. The pure functional style of the implementation makes it impossible for the user to corrupt system state as they cannot create a side-effect on the system.

7.2 Issues and Future Work

MPFL is a prototype system of a new software architecture for robotics planning, so it has its issues as well as plenty of potential interesting areas of further study.

7.2.1 Incomplete Features

The current implementation of the MPFL compiler is a complete working system, but to date not all features were implemented due to a lack of time.

7.2.1.1 Power Constraint Enforcement

The astute reader may have noticed that power constraints were not enforced along with the time constraints in the PI Evaluator. Implementing power constraints is straight forward and similar to the way time constraints are enforced. Each schedule has additional columns for power and energy used by each command. Planners have the responsibility of determining an estimate for peak power and total energy. The PI evaluator validates that power and energy usages is within the constraints, if so the schedule is valid, otherwise the runtime raises an exception indicating the planner is faulty.

7.2.1.2 Easily Swappable Planners

One of the intentions of MPFL is to allow users to swap out planners and knowledge bases without the need for recompilation. For example, one could login to the robot's computer running an MPFL client, shut it down, transfer a new planner in the form of a binary blob to the robot, restart the client, and the robot would now be using the new planner. Currently, one can do this by building their individual planners/-knowledge bases as runtime libraries (i.e. shared objects in Unix/Linux, dynamical link libraries in Windows) which can be dynamically linked when the client application is loaded. OCaml supports this feature as does every OS it runs on. The user however needs to be responsible for unloading the library from memory using whatever means the OS provides before running the program again so as to make sure the OS runtime linker chooses the newer version. This can be a bit painful, so in the future it would be helpful if MPFL could provide additional abstractions to make swappable planners easier. Note that there is no intent to ever make *hot swappable* planners as it does not make too much sense in the MPFL thought process and would likely make the system unnecessarily complex.

7.2.2 Problems with Distributed Planning

The modularity and component isolation afforded by planners and knowledge bases comes at a cost. MPFL is a distributed planning system, where each planner independently schedules its own problems. Though there is coordination between parent and children planners through the planner graph, there is no way to finely coordinate the actions created by one planner with the actions created by another. For example, the example `Search` planner from the previous chapter creates problems for the `Transit` and `UseSonar` planners. The only coordination between the two comes through the use of the parallel (`||`) operator. The parallel operator switches either the `Transit` or `UseSonar` plan instances into an `On(FORCE_RUN)` state as soon as the other goes into an `On(RUN)` state. There is no fine-grained coordination between the two tasks, which may be desired. Users can get around this by coupling planners when they implement them. Though this breaks the spirit of the MPFL concept and the independence of planners, it is a reasonable way to approach the problem if needed.

Additionally, a planner in MPFL only creates a single schedule though multiple schedule options may be available (i.e. there is more than one way to accomplish the same set of tasks). When a planner creates subproblems for its children planners, the scheduling algorithms of the children maybe able to generate multiple scheduling solutions. This could mean that the parent planner may have a preference for one schedule over the other, though any schedule that does not violate its constraints will do. This is a slightly more difficult feature to enable as now the planner implementor has to think about the possibility of creating multiple schedules, and also deciding which schedule from its children planners work best. What is more problematic is if a planner has multiple parents, in which case they must arbitrate over which planner is the best. One solution to implement this feature is utilize ***cost functions***, where planners can feed their children a higher-order function that can return a score of *goodness* that the planner can use to choose the best schedule that optimally meets the requirements of all the parents.

7.2.3 The Use of OCaml and Bindings for Other Languages

An early prototype of MPFL was written in the C++ programming language. The reasoning behind that decision was that C++ would provide not only better performance than other high-level languages, but that an object-oriented API in C++ would be more appealing to roboticists as it is a highly used language in the robotics field, is supported on a large number of platforms including many microcontrollers, and can be easily mixed with C code which is also highly used. This approach was abandoned as C++ proved to be a burdensome metalanguage for language development, primarily because of the amount of boilerplate code required, the lack of garbage collection, the lack of pattern matching, and clunky versions of common data structures such as dynamic arrays (vectors), linked lists, and hash tables. The compiler in its completed state along with the API took around 50,000 lines of C++ code when completed.

The prototype proved difficult to debug or alter due to its sheer complexity. It could be blamed on poor design, though a lot of effort went into it. Eventually the compiler was reimplemented in OCaml, where it took fewer than 1500 lines of code and is much easier to read. This is not surprising as the ML family of languages is excellent for prototyping languages[1]. The use of pure functional style allows for code to be flexible as one does not need to worry about side-effects so pieces can be moved around without worry as the code develops. The strong type system, rich standard library, garbage collection, higher order functions/closures, variant types, easily manipulatable linked lists, and pattern matching on variant types and lists also made OCaml the perfect choice. The OCaml compiler also generates good quality code in terms of performance and is considered one of the best in the functional language world. According to OCaml's creator, Xavier Leroy, the OCaml compiler generates code that will typically run no worse than 50% the speed of an equivalent C program utilizing a decent compiler (Schmitt and Leroy 2003). In the end, it turned out that even unoptimized debug code with OCaml's bytecode compiler (rather than its native compiler) ends up providing sufficient performance anyway as the overhead of the MPFL framework is minimal.

Though OCaml was a great choice to implement MPFL, having an API in OCaml is not a good practical choice. This is due to its lack of popularity along with the general lack of experience in the functional programming paradigm in non-academic settings. This is unfortunate as the planners implemented for the demonstration were extremely simple – the most complex being the `UseAutopilot` planner with 300 lines of code. However, we must deal with reality. Therefore we propose to utilize OCaml's C foreign function interface to implement a C binding for the API. The interface is already being utilized to link in some basic C libraries required for the demonstration planners and it is quite straightforward. Once the C binding is written, bindings for other popular languages can be written as virtually every programming language has a way to call

---

[1] ML is an acronym for '**m**eta**l**anguage', implicative of its intended use.

C functions, including Java, Python, Matlab, and C#. C++ can inherently call C functions. By creating a variety of bindings for MPFL, the framework becomes accessible to a much wider audience. The compiler/runtime is still implemented in OCaml and the benefits provided by it remain.

7.2.4 Metalanguage

The original intent of MPFL was to be specific to autonomous underwater vehicles. However, during its development it was realized that it could be applied to any autonomous system. All that ties MPFL specifically to AUVs is the various primitive plans provided by it (e.g. `Loiter`, `Search`, `UseAutopilot`). The primitive plans are not inherently tied to MPFL, in fact the compiler/runtime has no understanding of the problems its managing other than the time and power resources they use and are constrainted to. In other words, MPFL treats all plan instances equivalently, it just uses the type of plan instance to route it to the correct planner when the PI Evaluator is invoked.

The set of primitive plans and their plan instance constructors that currently exist are not completely realistic or finalized. However, rather than attempting to define those primitives, it would make more sense to allow users to add and remove their own primitive plan types with corresponding constructors. The concept is simply a generalization of the current implementation. For example, if one wanted to add a new primitive plan such as `UseWeapon` which allowed the robot to fire a weapon, they would now have the ability to do so along with naming the fields and their corresponding types that go into the new plan's plan instance constructor.

Implementing this feature is quite straightforward and requires implementing a ***metalanguage*** that allows users to define all the primitive plans that their MPFL compiler/runtime understands along with their constructor definition. A compiler for the metalanguage will then generate the OCaml code for all the appropriate class types (i.e. customized planner API) and build a custom version of the MPFL compiler/runtime for the MPFL client to link against.

7.2.5 Graphical Tool

Though the use of a programming language to control the robot (the MSL) is powerful and appealing, it is likely that many end users will not want to control their robot via a programming language. For example, a researcher may develop a robot that uses MPFL in military applications, but a soldier or sailor will not want to write a program when in the field. The intent of the MSL was never to be used by non-technical people, but rather as a middleware layer to communicate with a robot. Instead, what is proposed is creating a graphical tool where users can describe the tasks they want to do as a tree (Figure 7.1). Nodes in the tree represent tasks and children nodes are subtasks. Sibling nodes can be related using a planning operator (e.g. $>$, $||$, ˆ, &) or via a condition. Users can also bind constraints to a branch of the tree. Essentially the user would be visually creating the plan instance tree, which itself can directly be translated into an autogenerated yet easy to read MSL program. This program can then be fed into the MPFL compiler manually or directly through the GUI tool.

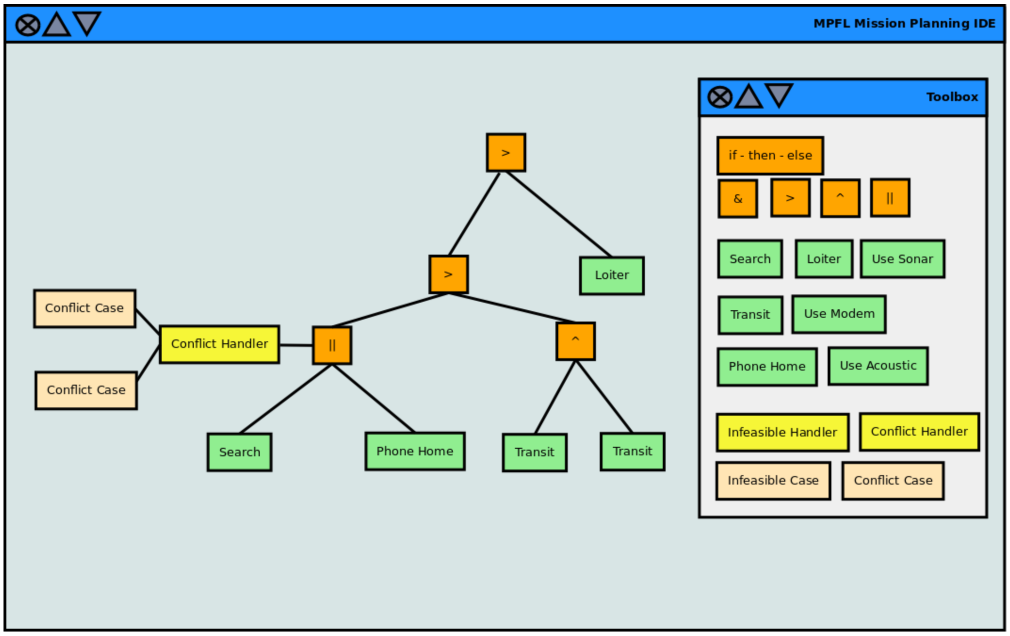


Fig. 7.1 Mockup of a potential MPFL Graphical Tool

7.2.6 Enhancing MSL

There is much to be desired from the MSL in terms of features.

7.2.6.1 Syntax

The syntax of the language is a bit verbose and can make it difficult to read code, so it may make sense to reevalutate it. The heavy use of parentheses can also become frustrating.

7.2.6.2 Primitive Plan Types

As mentioned earlier, a metalanguage should be created to encode different types of primitive plans and their corresponding plan instance constructors. However, until such a feature is added, the current set of primitive plans are not completely desirable for a real AUV. For example, there are many types of searches an AUV can perform and additional parameters that define the search besides an area, a sensor, and a lanewidth. It is quite easy to add in new primitive plan types, however it requires modifying the implementation code which the end user should not touch.

7.2.6.3 Better Lookup Calls and More Built in Functions

The current set of `Lookup*` calls in the MSL only cover the four primitive types: boolean, integer, floating point, and string. In addition to these types, there should be lookup calls for all MSL types, such as *position* and *angle*. This is an easy extension to add and just requires the user to add more lookup calls to their knowledge base implementation. In addition to being able to lookup values, it would be handy to be able to perform some common functions with them. Just as one can perform basic arithmetic with floating points and integers in the MSL and can compare equality of the string, calls for calculating the distance between two *position* values or getting the value of a *duration* value in minutes can be useful in the MSL, particularly with conditional expressions.

7.2.6.4 Variables and Functions

Adding variables and functions to a language, as found in most high-level languages, is quite straightforward. However, this feature was never added into the MSL as the point was to make it as narrow in scope as possible. Adding functions and variables would make the MSL look more like a general-purpose programming language than a domain-specific one, diluting its purpose and making MSL programs unnecessarily complex. However, adding these features may not necessarily be a bad idea; if the MSL becomes powerful enough to be used in general programming, there may come a point where the MPFL client application is no longer needed or at least much smaller in size. Though it is unlikely the MSL will ever add these features, it is interesting to ponder what effects it could have.

7.2.7 Real Usage and System Tuning

Though this thesis puts MPFL in a positive light, the system has only been tested in simulation and to a limited extent. To prove its worthiness, MPFL needs to be used on a variety of robots, by a variety of different people. This real life usage will help provide feedback about its usefulness. If there are shortcomings, they should be addressed as found.

7.3 Related Work

MPFL is one of countless architectures for autonomous planning and not the first to use a domain-specific programming language. The approach however was unique in that the focus was on a very practical and simple software tool where a robotics software engineer could learn the MSL and MPFL API in a few hours and generate highly-verified and deliberative autonomy in contrast to other tools which often can be a bit overwhelming or only applicable to a set of toy problems. However, MPFL was inspired by many of these alternative architectures, and it is prudent to take a look at the other approaches, their advantages, and their shortfalls.

7.3.1 Planners

Those that come from an AI background will likely have a different association with the term *planner* than the way it is used in MPFL. In the AI world, ‘planners’ are similar to the notion of a planner in MPFL in that they are entities that take a set of tasks and a set of constraints and output a schedule of actions. The difference is that in AI, planners usually also take with them a *start state*, a set of *end states*, and a set of valid *actions* similar to those in a finite state machine. In the AI world, planners are algorithms which figure out a schedule of actions to get from the start state to the end (i.e. goal) state. In other words, planners are simply general purpose constraint solvers where the user encodes the problem, the goals, and the valid actions that can be taken by the entity for which the schedule is being generated (e.g. a robot) (Russell and Norvig 2010). MPFL does none of this. Rather it just tells the user to implement a scheduling algorithm for each particular task within the domain. MPFL just provides you with the information needed to schedule, but does not figure out how to schedule the task. Though this may seem like a lacking in MPFL, it was done on purpose because attempting to generalize planning is not a fruitful endeavor in the opinion of the author. Though it is interesting and potentially more useful, in practice general purpose planners have only been able to solve toy problems efficiently. Though more complex problems can be solved, they typically become intractable due to the inefficiency of the solver which is essentially building a proof tree to get from the start state to the goal state. The author thinks it makes more sense to write scheduling algorithms for specific scheduling problems as they occur, which is exuded by the various types of planners that one can create in MPFL. As an example, the way the demo `UseAutopilot` scheduler works with a genetic algorithm is much different than the way the demo `UseAcoustic` planner utilizes a linear program.

7.3.2 Languages for Planning and Robotic Control

Though MPFL takes a different and more limited approach to planning than the generalized planners that are the focus of AI research, there is one thing in common: the use of a domain specific programming language. Many popular planners utilize a language which allows users to describe the tasks they want to perform along with constraints, but also allowing constructs for encoding valid actions, start states, and goal state. The use of languages to encode and reason about plans has been prevalent since the days of PLANNER (Hewitt 1971). Popular languages include STRIPS (Fikes and Nilsson 1971, 1994) and PDDL (Mcdermott 1998). Simmons and Apfelbaum (1998) describes an extension to the C++ language called TDL that enables specification of robotic tasks. Gat (1997) describes ESL which is a planning language for embedded autonomous agents. Likewise, numerous languages specifically for reactive robotic control/planning have also been created such as COLBERT (Konolige 1997) and RPL (Mcdermott 1993). Peterson et al. (1999) describes a declarative language for robotic control which is the most similar concept to MPFL found in the literature. Eberbach et al. (2003) describes a common control language with emphasis on the control of multiple autonomous vehicles.

7.3.3 Robotic Software Architectures

A ***software architecture*** is a “methodology for structuring algorithms” (Russell and Norvig 2010). An architecture encompasses the building blocks for building systems following a common design philosophy. These building blocks can be in the form of software tools, frameworks, APIs, and languages. A ***robotic software architecture*** is a software architecture for the software that runs onboard a robot, what we have referred to as the robot’s autonomy software stack. A robotic software architecture can be used to help build robotics software more rapidly. MPFL can be considered one piece of a robotic software architecture with a focus on mission planning and resource allocation.

7.3.3.1 Design Philospohy: Reactive, Deliberative, Hybrid

Robotic software architectures can be described as being, reactive, deliberative, or a mix of the two (hybrid). The first chapter described the terms *reactive* and *deliberative*. The more reactive a robot is, the less analysis it makes about its next decision. In other words, it thinks less about the long-term consequences of its actions and does not think much about its past experiences. The model is essentially a simple stimulus-response mechanism. The more deliberative a robot is, the more it thinks about its actions based on past information and predictions of the future. Hybrid systems are ones that utilize both philosophies where appropriate for the robot's application (Alami et al. 1998; Gat 1998). Planners (both in the traditional AI sense and MPFL planners) tend to follow a deliberative philosophy. On the other end of the spectrum, one of the best examples of a reactive system is the ***Subsumption Architecture*** created by MIT researcher Rodney Brooks (Brooks 1986, 1991). The premise in Subsumption Architecture is to create an N-level tree of *behaviors* (which are analogous to the planners in MPFL). The behaviors higher up in the tree represent high-level aspects of the robot's intelligence (such as performing a search) whereas the lowest levels represent the most primitive abilities of the robot (e.g. sending power to the servo to move the robot). The higher level behaviors are performed by breaking up the problem into a series of input for the children behaviors (i.e. higher level behaviors *subsume* children behaviors). For example, if we think of the task of driving a car in terms of Subsumption Architecture, the highest level behaviors could pertain to managing the high level goal of getting to a particular destination and the lower level behaviors deal with the actual impulsive movements that must be made to the steering wheel and accelerator to accomplish the higher level goals. Subsumption Architecture and its variants are sometimes referred to as ***behavior-based autonomous control systems*** (Arkin 1998).

What makes the Subsumption Architecture reactive is that the behaviors in the tree are simple functions that take an input from higher level behaviors indicating what needs to be done and in turn causes those behaviors to create further input for its children. The behaviors are stateless and simply reacting to input. The idea is that if one combines enough of these simple behaviors together, a more complex overall ***emergent behavior*** will be exuded by the robot. For example, if you look at the way a flock of birds flies, the exhibit complex behavior as they fly together in perfect formations without explicit communication. Each bird is following a simple set of rules: *stay a certain distance to the bird to your left, to your right, and the one in front of you.* Robots can also follow simple rules like this and exhibit complex control. In fact this is the very approach used in iRobot's *Roomba* vacuum cleaner robots, and it is no surprise that Chief Technology Officer (CTO) of iRobot happens be Rodney Brooks. Brooks' philosophy has been controversial, many agree, many do not. It has been found in practice though when more and more complex behavior is required, simplistic reactive control systems are limited and more complex logic is needed.

In practice, robots tend to use a hybrid approach such as the ***three layer architecture***. In this model, three abstraction layers are used: *reactive*, *executive*, and *deliberative*. The reactive layer sits closest to the hardware and performs the more impulsive tasks, such as orienting a wheel to face a certain direction. This layer needs to be really fast and typically implemented using concepts from the field of *control theory*. The deliberative layer focuses on the high level mission and tasks it composes and sending commands to the reactive layer to perform the mission. The executive layer is simply a glue layer between the two which does the mediation. MPFL is not a complete autonomy solution, but works well as the deliberative layer in a robot using a three-layer robotic architecture.

Though MPFL is a deliberative system, it was ironically inspired by Subsumption Architecture. The planner graph in MPFL is based off the idea of Subsumption Architecture's N-level behavior tree. Planners are analogous to behaviors, but rather than being simple stimulus-response objects, they are deliberative schedulers. However, the notion of breaking down the problem and delegating it to lower level entities remains. This idea is nothing revolutionary, rather it is just another instance of layering as is found throughout all of computer science. MPFL was also not the first planning system to think of this, as many planners and scheduling systems use this. The technical term is ***hierarchical planning system*** as the system is broken into many layers where high level tasks are broken into smaller ones for lower layers. As each planner operates independently of each other, MPFL is also a ***distributed planning system*** or equivalently a ***multi-agent planning system***.

7.3.3.2 Robotic "Operating Systems"

It was implied in previous chapters that MPFL itself can be thought of almost as an operating system for a robot. The idea of thinking about creating an abstraction layer for robots which provides services to robotic software developers the way an operating system provides abstractions to its application developers is nothing new. Virtually every sufficiently complex robotic software architecture can be interpreted as being a robotic operating system. There are however robotic software architectures out there that attempt to be robotic operating systems. These systems act as an intermediary between the application-specific code and the robot, providing APIs for commonly needed tasks in robotics including message passing, navigational calculations, data fusion algorithms, device driver-style hardware abstractions,and data logging and playback. Examples include:

- **Mission Oriented Operating Suite (MOOS)** - MOOS is a set of simple tools written in C++ at Oxford University and MIT for autonomous robot applications (Newman 2003). MOOS allows one to break up their autonomy software stack into several processes, each one able to communicate with each other by publishing and subscribing to data feeds available to all other processes. Each process is synonymous with an agent in a multi-agent system. MOOS comes with a set of prebuilt agents for data logging, navigation, debugging, and talking to various hardware devices. MOOS is free, open-source software with comprehensive documentation. The MPFL demo system utilizes MOOS for interprocess communication.

- **Robot Operating System (ROS)** - ROS, maintained by Willow Garage, is a similar concept to MOOS, but goes beyond it by trying to provide not only the basics, but additional tools used in modern robotics such as computer vision algorithms, robot simulators, and motion planning for robotic arms (Quigley et al. 2009). ROS is free, open-source software and well documented.
- **Microsoft Robotics Studio (MRS)** - MRS is a robotics development tool created in 2006 by Microsoft Corporation. Robotics Studio provides a framework for building individual *services* where each service is an agent in a multi-agent system. For example, a robot can have a service for managing the camera, one for controlling speed, one for managing the mission, and one for a wireless Bluetooth connection. A message passing system with standardized messages allows users to write the glue code in any .NET-based language. In addition to the framework for building the robot's autonomy, MRS comes with a simulator and real-time 3D visualization tool for mapping the robot's movements. Microsoft's goal with Robotics Studio is to standardize robotics software leveraging their influence in the information technology world (Jackson 2007). Though a great idea, Microsoft's development efforts are focused on their Windows operating system platform, as consistent with their philosophy. This is limiting as it creates proprietary lock-in and excludes the efforts of the open source software community. The latter is very problematic as open source software such as Linux-based operating systems and the GNU compiler collection (gcc) are ubiquitous in the robotics world. On the other hand, MRS' development tools are very well-designed and easy to use allowing software developers familiar with Microsoft's .NET framework to quickly take off.
- **Coupled Layer Architecture for Robotic Autonomy (CLARAty)** - CLARAty is a robotics architecture developed through a collaboration between Jet Propulsion Laboratory, NASA Ames Research Center, Carnegie Mellon University, and the University of Minnesota (Volpe et al. 2001). CLARAty also utilizes a multi-agent approach like MOOS, ROS, and MRS. A limited subset of the CLARAty code is available to the public under a limited open source license.

7.3.4 Cognitive Architectures

Cognitive architectures are software architectures that attempt to model the human mind or an equivalent there of. These architectures are not developed with the focus of controlling an autonomous robot, but rather to understand the nature of the human mind and human intelligence. The design of these systems is based on theories of how the mind works from cognitive science. As mentioned in the first chapter, cognitive science is a relatively new field that incorporates many disciplines including computer science, artificial intelligence, biology, neuroscience, and psychology. Though cognitive architectures were not meant to run on robots, many of the problem solving capabilities that have emerged from the development of these architectures have crossed over into the autonomous robotics realm. As mentioned in chapter 2, better autonomy implies more intelligence, so it is not surprising that successful cognitive architectures have been applied to autonomy problems. Vernon et al. (2007) presents a fairly thorough history and comparison of various cognitive architectures. Three of the most well known cognitive architectures are:

- **Soar** - Soar is a cognitive architecture, created by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University and now maintained by John Laird's research group a the University of Michigan. Soar is the successor to Herbert Simon and Allen Newell's *General Problem Solver (GPS)*, and meant to be a software implementation of the cognitive model Newell described in his book *Unified Theories of Cognition* (Newell 1990). Soar utilizes a programming language based approach where users encode both the intelligent agent's knowledge representation and reasoning through the use of production rules similar to those in a logic based language. Soar is similar to the notion of the classical AI planner mentioned earlier, but attempts to avoid problems like rule explosion by introducing metarules that allow the system to cache and stereotype information. (Lehman et al. 1996; Laird 2008; Laird and Congdon 2009)
- **ACT-R (Adaptive Control of Thought - Rational)** - ACT-R is a cognitive architecture created by John Anderson at Carnegie-Mellon University and also uses the rule based approach of Soar. It was inspired heavily by Soar and Newell's book *Unified Theories of Cognition* (Newell 1990). Unlike Soar though, rather than using its own programming language, ACT-R is simply a big collection of Lisp functions that the user utilizes in their client application. (Anderson 1996; Anderson et al. 1997)
- **EPIC (Executive Process - Interactive Control** - EPIC is a cognitive architecture developed by David E. Kieras and David E. Meyer at the University of Michigan. EPIC is different from other cognitive architectures as it has a focus on perceptual and motor capabilities making it particularly useful in *Human-Computer Interaction (HCI)* applications. EPIC is similar to ACT-R in that it is both a rule-based system and is presented a collection of Lisp functions.(Kieras and Meyer 1997)

7.4 Conclusion

In practice, the high-level tools such as classical planners and cognitive architectures have only been able to solve problems in a niche area as they either fail to scale to a more general set of problems or do not have the means of solving problems relevant to autonomous robots. Tools such as the ROS and MOOS fall on the other end of the spectrum and are used quite heavily in practice, but the burden of implementing the high-level intelligence is put on the user of the tools. What makes MPFL unique however is that it tries to find a sweet spot between the practical and the theoretical. MPFL was meant to help robotic software developers move up another level of abstraction from the facilities that ROS and MOOS provide, yet build their planning software in a way that is consistent, correct, fast, easy to reason about, and reusable. Classical approaches have failed because their goals are too lofty: creating a general-purpose planning algorithm. It makes more sense to allow users to build several simple, yet smarter and more efficient domain specific schedulers (i.e. MPFL planners). Though the idea of distributed and hierarchical schedulers is not new, MPFL provides a common framework for building these schedulers which allows for verification at every level along with an extremely powerful and declarative language for controlling the robot. The language is what makes MPFL shine; one can specify the most convoluted and complex mission they can think of, and MPFL will attempt it without violating any constraints. This is remarkable given that from the robotic software developer's perspective they are building simple isolated components, but then are able to get a complex, emergent behavior out of it through the language.

MPFL is not meant to be a silver bullet for solving all problems pertaining to planning in autonomous robots. It does not contain in it anything that the robotics and artificial intelligence communities have not yet seen. However, it tries to take existing concepts and structure them in a particular way using the discipline of software engineering in order to create a tool that is useful in real-life robotics applications to real-life software engineers. At the very least, MPFL creates a fresh perspective on the implementation of robotic autonomy.

Bibliography

R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4), April 1998.

John R. Anderson. Act: A simple theory of complex cognition. *American Psychologist*, 51(4):355–365, April 1996.

John R. Anderson, Michael Matessa, , and Christian Lebiere. Act-r: A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, 1997.

Ronald C. Arkin. *Behavior-Based Robotics*. 1998.

J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335: 131–146, 2005.

Ellen Bialystok. Language acquisition and bilingualism: Consequences for a multilingual society. *Applied Psycholinguistics*, 28:393–397, 2007.

Ellen Bialystok. Global-local and trail-making tasks by monolingual and bilingual children: Beyond inhibition. *Developmental Psychology*, 46(1):93–105, 2010.

J. Boyd. A discourse on winning and losing. In *Air University Library Document No. M-U 43947 (Briefing Slides)*, 1987.

Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

Rodney A. Brooks. New approaches to robotics. *Science*, 253(5025):1227–1232, September 1991.

Lizzie Buchen. Patches for faces. *Scientific American*, December 2008.

Peter Carruthers. The cognitive functions of language. *Behavioral and Brain Sciences*, 25:657–726, 2002.

John Colapinto. The interpreter: Has a remote amazonian tribe upended our understanding of language? *The New Yorker*, April 2007.

Leda Cosmides and John Tooby. Origins of domain specificity: The evolution of functional organization.

Leda Cosmides and John Tooby. Better than rational: Evolutionary psychology and the invisible hand. *AEA Papers and Proceedings*, 84(2), May 1994.

Fergus I.M. Craik, Ellen Bialystok, and Morris Freedman. Delaying the onset of alzheimer disease: Bilingualism as a form of cognitive reserve. *Neurology*, 75(19), November 2010.

Eugene Eberbach, Christiane N. Duarte, Christine Buzzell, and Gerald R. Martel. A portable language for control of multiple autonomous vehicles and distributed problem solving. In *Proceedings of the 2nd International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS'03)*, 2003.

Charles C. Eriksen, T. James Osse, Russell D. Light, Timothy Wen, Thomas W. Lehman, Peter L. Sabin, John W. Ballard, , and Andrew M. Chiodi. Seaglider: A long-range autonomous underwater vehicle for oceanographic research. *IEEE Journal of Oceanic Engineering*, 26(4), October 2001.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189 – 208, 1971. ISSN 0004-3702. doi: DOI: 10.1016/0004-3702(71)90010-5.

Richard E. Fikes and Nils J. Nilsson. *STRIPS, A Retrospective*, pages 227–232. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-52186-5. URL `http://portal.acm.org/citation.cfm?id=190908.190948`.

Erann Gat. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *1997 IEEE Aersospace Conference Proceedings*, volume 1, pages 319 – 324 vol 1, February 1997.

Erann Gat. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. MIT Press, 1998.

Jan Heering and Marjan Mernik. Domain-specific languages for software engineering. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.

Carl Hewitt. Planner: A language for proving theorems in robots. In *International Joint Conference on Artificial Intelligence*, pages 295–301, 1971.

Kenneth E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23:444–465, August 1980. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358896.358899. URL `http://doi.acm.org/10.1145/358896.358899`.

Jared Jackson. Microsoft robotics studio: A technical introduction - standardizing robotic coordination and control. *IEEE Robotics & Automation Magazine*, December 2007.

Paul Kay and Willett Kempton. What is the sapir-whorf hypothesis? *American Anthropologist*, 86(1):65–79, 1984. ISSN 1548-1433. doi: 10.1525/aa.1984.86.1.02a00050. URL `http://dx.doi.org/10.1525/aa.1984.86.1.02a00050`.

David E. Kieras and David E. Meyer. An overview of the epic architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, 12:391 – 438, 1997.

Kurt Konolige. Colbert: A language for reactive control in sapphira. In *Proceedings of the 21st Annual German Conference on Artificial Intelligence: Advances in Artificial*

Intelligence, pages 31–52, London, UK, 1997. Springer-Verlag. ISBN 3-540-63493-2. URL `http://portal.acm.org/citation.cfm?id=647617.731754`.

John E. Laird. Extending the soar cognitive architecture. In *Proceeding of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 224–235, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press. ISBN 978-1-58603-833-5. URL `http://portal.acm.org/citation.cfm?id=1566174.1566195`.

John E. Laird and Clare Bates Congdon. The soar user's manual version 9.1. 2009.

Robin T. Laird. Evolving u.s. department of defense (dod) unmanned systems research, development, test, acquisition & evaluation (rdta&e). In *Unmanned Systems Technology XI, Defense Security Symposium*, volume 7332, April 2009.

Jill Fain Lehman, John Laird, and Paul Rosenbloom. A gentle introduction to soar, an architecture for human cognition. In *In S. Sternberg & D. Scarborough (Eds), Invitation to Cognitive Science*. MIT Press, 1996.

S. Mawu, W.T. Wiersma, and T.A.C. Willemse. Language-driven system design. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, January 2002.

D. Mcdermott. Pddl-the planning domain definition language. In *Machine Intelligence*, 1998.

Drew Mcdermott. A reactive plan language. Technical report, 1993.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

Marvin Lee Minsky. *The Society of Mind*. 1986.

A. Newell, J.C. Shaw, and H.A. Simon. A variety of intelligent learning in a general problem solver. Technical report, July 1959.

Allen Newell. *Unified Theories of Cognition*. 1990.

Allen Newell and Herbert Simon. Computer science as empirical inquiry: Symbols and search (1975 acm turing award lecture). *Communications of the ACM*, 19(3), March 1976.

Michael Newman. Moos - a mission oriented operating suite. Technical report, 2003. Tech. Rep. OE2003-07.

J. Peterson, G.D. Hager, and P. Hudak. A language for declarative robotic programming. In *1999 IEEE International Conference on Robotics and Automation Proceedings.*, volume 2, pages 1144 –1151 vol.2, 1999. doi: 10.1109/ROBOT.1999.772516.

Benjamin C. Pierce. *Types and Programming Languages*. 2002.

Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: An open-source robot operating system. In *Open-source Software Workshop of the Int. Conf. on Robotics and Automation*, 2009.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach: 3rd Edition*. Prentice Hall, Upper Saddle River, NJ, USA, 2010.

Yoshiaki Sakagami, Ryujin Watanabe, Chiaki Aoyama, Shinichi Matsunaga, Nobuo Higaki, and Kikuo Fujimura. The intelligent asimo: System overview and integration. In *Proceedings of the 2002 IEEHRSJ International Conference on Intelligent Rotots and Systems*, October 2002.

Alan Schmitt and Xavier Leroy. Coyote gulch test in caml, January 2003. URL `http://lwn.net/Articles/19378/`.

Ravi Sethi. *Programming Languages: Concepts & Constructs - Second Edition*. Addison-Wesley, 1996.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts - Sixth Edition*. John Wiley & Sons, New York, NY, USA, 2002.

R. Simmons and D. Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931 –1937 vol.3, oct 1998. doi: 10.1109/IROS.1998.724883.

Michael Sipser. *Introduction to the Theory of Computation - Second Edition*. Thomson Course Technology, 2006.

William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing*, 7:255–275, 2008. ISSN 1567-7818. URL `http://dx.doi.org/10.1007/s11047-006-9032-6`. 10.1007/s11047-006-9032-6.

B. Tribelhorn and Z. Dodds. Evaluating the roomba: A low-cost, ubiquitous platform for robotics research and education. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1393 –1399, april 2007. doi: 10.1109/ROBOT.2007.363179.

David Vernon, Giorgio Metta, and Giulio Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agent. *IEEE Transactions on Evolutionary Computation*, 11(2), April 2007.

Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The claraty architecture for robotic autonomy. In *2001 IEEE Aerospace Conference Proceedings*, volume 1, pages 1/121–1/132 vol 1, 2001.

Rafaela von Bredow. Brazil's piraha tribe: Living without numbers or time. *Spiegel Magazine*, May 2006.

Vita

Mirza Akbar Shah

Education

The Pennsylvania State University University Park, Pennsylvania 2006–Present

M.S. in Computer Science & Engineering, expected in July 2011

The Pennsylvania State University University Park, Pennsylvania 2001–2005

B.S. in Computer Science

Awards and Honors

John K. Tsui Scholarship	2004–2005
PSU Board of Trustees Scholarship	2003–2004

Research Experience

Graduate Research The Pennsylvania State University 2007–Present
Thesis Advisor: Prof. John Hannan

This research involved the development of a domain-specific programming language for autonomous underwater vehicles (AUVs). This language allows users of AUVs to specify missions in a very high level language and have a virtual machine process that specification and develop a plan to accomplish it. The virtual machine can dynamically replan the missions as necessary to accomplish the goals specified. The language semantics are customizable through a modular, component development system via a well-defined application programmer interface (API).

Professional Research The Pennsylvania State University Applied Research Lab 2005–Present

Researcher and software engineer involved with development of autonomous systems for next-generation military applications, particularly in the area of undersea warfare.