UA2

Unmanned Autonomous Architecture

Developer's Guide and API Specification

Mirza A. Shah
Penn State Applied Research Laboratory
Autonomous Control & Intelligent Systems Division

DRAFT COPY – July 7, 2008

Table of Contents

Preface

1. What is in this Document

This document describes an implementation of a modular, autonomy framework for autonomous underwater vehicles (AUVs) called the Unmanned Autonomous Architecture (UA2), funded by the United States Office of Naval Research's (ONR) Unmanned Undersea Vehicle Group (PMS 403). This document was produced as part of the requirements for the UA2 project as mandated by ONR.

This document is the primary manual for an AUV software developer that wishes to utilize UA2 in their vehicle. It describes the role of the framework, how to build and use it, and fundamental concepts behind it.

2. Intended Audience

This document is intended to be read by individuals that are interested in or are planning on using the UA2 framework in an AUV. Chapter 1 is an introduction to the framework, and is intended for both technical and non-technical people. Chapters 2 - 6 are intended to be read by engineers, ideally with a computer science or similar background, that will be using UA2 in their vehicle. Chapters 2, 4, and 5 require a somewhat significant understanding of the C++ programming language, and compilers for that language. Chapter 5 and 6 require an understanding of object-oriented design and software engineering. Chapter 3 requires a rudimentary understanding of programming languages concepts and theory, such as type systems.

Chapter 1
Introduction

The goal of the Unmanned Autonomous Architecture (UA2) project was to create a common autonomous control architecture for autonomous underwater vehicles (AUVs). Across the AUV industry, it is typical for AUV software developers to use their own in-house autonomous control architectures for vehicle control software, customized for their particular vehicle, payload and intended missions. Even here within Penn State ARL, we have different controller architectures running across our vehicles, which are all incompatible with each other. Here in the Autonomous Control & Intelligent Systems division (ACIS), we have attempted to proliferate and encourage the usage of a common controller architecture throughout the Lab called the Prototype Intelligent Controller (PIC), which even though provides a common architecture conceptually, in practice, each controller using this architecture is incompatible with each other due to its nature as a general-purpose system that does not focus on any specific class of problems.

The primary mission of UA2 is to get rid of the redundancies found in AUV autonomy development, and provide a framework that genuinely encourages reuse of software. The requirements of UA2 were to create an architecture which is truly modular, well-defined, and allows software components to be used across different AUVs without having to change modify the internals of those components. For example, if somebody was to create a component for executing a search behavior in some vehicle *x*, that component should have the ability to be reused transparently in vehicle *y*. Another example would be components for controlling an autopilot or a sonar, or some other type of vehicle payload.

Here at Penn State ARL, we have built a prototype for our vision of UA2. This document describes how one is to to use our implementation of UA2 in their particular vehicle, and how to reuse components built using our framework.

1. Common Architecture = Common Language

We approached the UA2 problem in a unique and interesting way: we looked at UA2 as a ***domain-specific programming language***. Domain-specific programming languages, in contrast to general-purpose programming languages (e.g. C, Java, Ada, FORTRAN), are designed to build programs for a particular area (or domain) of problems . The constructs, abstractions, and syntax provided in such a language are tailored to a particular domain.

One of the best examples of a domain-specific programming language is the Standard Query Language (SQL), which is used by virtually any system and website that uses a relational database as its backend. The language allows you to *declaratively* query data based on certain criteria. For example, you can make a query in SQL that at a high-level says:

“Give me the last names for all people in the database with age greater than 43 years old that live within 20 miles of New York City”.

The database query engine parses and interprets this query, and determines a strategy for getting the data as fast as possible. The user has no control over how the engine is doing its search, that responsibility is left to the engine. What makes this so powerful is that the language allows a user to only focus on the domain of “data querying”, and provides only constructs for doing that task. The declarative (versus imperative) nature of the language also means that the user doesn't have to worry about *how* the data is queried, they just care *what* is being queried.

In UA2, we created a domain-specific language where the domain can be best described as “AUV mission planning”. We provide a language where AUV missions can be described in our high-level language, and a ***virtual machine*** (similar in its role to the query engine in a relational database, mentioned in the next section), worries about achieving that task. For example, one can roughly say in the UA2 language:

“I want you to go search these areas, while sending status updates to a base ship in parallel. Then I want you to rendezvous with the base ship, or return to port, I don't care which. Finally, I want you to do the searches before

1400 hrs, using only 50% of your battery, and not to execute your rendezvous mission before 1500 hrs. If you have any problems doing the status updates and search at the same time, forget the status update and just do the search".

The virtual machine will parse and interpret this request, and do its best to accomplish the mission. This example is seemingly complex, but our UA2 architecture can make developing a control system of this sort quite straight forward.

The UA2 language is only a means to controlling the vehicle once all the software is in place. AUV developers have to build plug-in, reusable, software components that are strongly bound to the language, that hook into the virtual machine.

2. What UA2 Does in a Nutshell

UA2 is simply a framework for modeling an autonomous system as a giant ***scheduler***. A scheduler can be thought of as an entity that looks at a collection of problems, and then creates a ***schedule***, a list of sequential actions that are executed at specific times, to solve those problems. For example, when you wake up each morning, you have a set of goals you would like to accomplish. You'll attempt to accomplish these tasks by building your own schedule that, if executed, should allow you to complete the tasks at hand. Let's say on any given weekday, you wake up and decide you need to get ready for work, eat breakfast, go to work, and then come home. You'll think about what tasks need to be done 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 terms ones in the back of your mind. You'll 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 requirement. 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 so that you have to replan, and modify your schedule. For example, your car won't start, so you need to take get an alternate ride, which causes you to be late for work. Being late for work may alter your other plans as well, and affect the schedule you have laid out. You will then again have to replan, and rebuild your schedule.

UA2 works in a freakishly similar way to the example above. UA2 has a virtual machine, called the ***UA2 compiler/runtime engine***, or simply the ***UA2 runtime***, which is responsible for accomplishing goals specified in a ***mission specification*** written in the UA2 language (mentioned briefly in the previous section, see chapter 2 for details). The runtime acts as the scheduler for your AUV, and continuously builds and updates a series of schedules to accomplish the goals specified in the mission specification. The schedules built by the runtime are meant for specific subsystems of your AUV, such as the drive system, sonars, and payload. As the situation around and within the AUV changes, the UA2 runtime replans and builds new schedules to adopt to the scenario. Also, just like a human being, the UA2 runtime attempts to schedule as much as it can into the future, to increase the likelihood of success, and to allocate resources wisely. The UA2 runtime also has a natural way of handling unexpected problems, just as in the example above, breakfast was thrown out the window, some tasks an AUV might want to do may need to be thrown away to achieve more important goals.

The UA2 runtime is divided into two parts, the ***planning kernel***, and reusable plug-in modules (***planners*** and ***knowledge bases***) provided by the user. The kernel is responsible for managing all the different types of tasks at at a high level, and the planners are responsible for building a schedule for a collection of tasks of a specific type. The kernel also verifies the validity of schedules built by these planners, and coordinates passing of vital information back and forth between the two entities. It may be interesting to point out the name "planning kernel" is inspired by operating systems. The UA2 runtime's design is inspired by operating systems, and in a sense, the runtime can be thought of as an operating system for an AUV. The planning kernel is like an OS kernel, where as planners are like drivers that are linked into the kernel, and essentially become a part of it. This analogy of UA2 as an operating system is seen throughout the document, as a lot of concepts are taken from operating systems.

3. Architecture Features

Deliberative Control

In autonomous control, controllers can be either reactive or deliberative, or at some point between the two. What classifies a reactive vs deliberative system is quite subjective, but notionally, a reactive system is one that makes decisions for the immediate situation, and makes decisions for “what to do right now”. Reactive systems tend to avoid holding internal state, and do not reflect much on their past actions. Pure deliberative systems are planning systems, which not only plan what to do right now, but plan into the future “what is to be done”. These systems also typically hold onto state, and evaluate what has been done in the past, what is the current situation, what is the relationship between various subtasks and what steps should be taken in the future. It can be inferred from this that deliberative systems are much more complex and difficult to develop, but also provide much more elaborate autonomy, whereas reactive systems are simpler to implement, easy to reason about, but have a limit to what their autonomy can do.

Our UA2 architecture is essentially a scheduling system, designed for building deliberative systems. The constructs provided in our language and API are meant to help ease the development of deliberative controllers, even though one could simply use it to build reactive controllers.

Graceful Degradation and Error Handling

The UA2 system provides facilities for graceful error handling. Hierarchical error handling is an important part of the computational model for UA2, and was built in from the very start. Even as missions and subsystems of a vehicle fail, facilities are provided to allow graceful degradation of the system, both at a language level, and at the planner API level.

True Plug & Play Capability

As before, one of the main goals of UA2 was creating a truly open, modular system where reusability is not just paid lip service. As long as planner is built correctly, it will function in any system that uses UA2. The API for building planners is very well-defined, type-safe, and side-effect free, making it difficult to write a planner that is not valid, unless one goes out of their way to.

Strong Static and Run-time Verification

The use of domain-specific language allows us to utilize a lot of concepts from programming languages theory. The UA2 compiler can statically verify mistakes that have cost organizations billions of dollars worth of damages and woes, such as having the wrong units for a particular value. Before an AUV even touches the water, a mission written in the UA2 language can be statically verified by the UA2 compiler for things like invalid units, incorrect data ranges, undefined references, and more. During runtime, the UA2 runtime is constantly verifying the missions the AUV is executing and ensuring that planners are not performing illegal operations that violate the objectives/constraints set in the mission specification written in the UA2 language.

Easy Integration

We recognize that the level of participation one wants UA2 to have in their AUV autonomy can vary from very light to very heavy. UA2 alone is not enough to control a particular vehicle, what we provide is presented in the form of a software library that is linked into some host application (which is most likely a controller). The host application can talk to UA2, and looks at it as an advisory module. Whether the host application depends on UA2 to do the brunt of the work, or only do a few specific things, is up to the AUV software developer.

The host application's job is to eventually issue commands to the vehicle subsystems (payload, sonar, autopilot, etc) by translating recommendations from the UA2 runtime. We provide a demo application which depends heavily on UA2, and where the host is extremely simple (about a page of code), issuing commands to the varying subsystems based on what UA2 tells it to do.

This ability to choose UA2's role in building an autonomy solution gives a lot of flexibility to an AUV software developer.

4. Limitations Upfront

We do not want people to have illusions about what this system can or cannot do. Our architecture, just like any other piece of software, has limitations:

Not A Finished Product

The current UA2 implementation is not complete, in the sense some of the constructs of the UA2 language may not realistically model features of an AUV, as this is an experimental framework. However, the design of the language allows us to rapidly change these constructs to more suitably meet the requirements of an AUV developer to be more realistic. As we work more with the community, and our own internal AUV groups here at Penn State, we will eventually end up with a finished product.

Lack of Shared Planning

The UA2 computational model makes it difficult to do what is sometimes called “shared planning”. This will make more sense as one goes through this document, but in essence, in the UA2 model it is difficult to have coordination between tasks executed by the runtime. There are however workarounds for this, that will be discussed later. One of our goals in future work is to address this issue.

Chapter 2
Getting Started

1. System Requirements

UA2 is written in ANSI C++, and will compile on any system that has a modern C++ compiler. UA2 requires the compiler to have decent support for templates, run-time type information (RTTI), and the Standard Template Library (STL).

2. Building UA2

UA2 is provided as a single C++ library, that can be either statically or dynamically linked into a host application. We provide both a Makefile and a Microsoft Visual Studio 2005 project. Besides the C and C++ standard libraries, there are no external dependencies.

Building Instructions - Make

TODO

Building Instructions - Microsoft Visual Studio

TODO

Chapter 3
The UA2 Language

Before we get into using UA2 in an application, we have to understand its primary interface, the domain-specific language which we refer to as the UA2 mission specification language, or simply the UA2 language. Users of the AUV are meant to specify the missions they want their AUV to attempt in this language, and then feed it via some means to the AUV autonomy software, which will eventually pass it onto the UA2 compiler/runtime. We begin by looking at the primary constructs in the language, ***Plans***.

1. Plans and Plan Instances

The primary type in UA2 is called a ***plan***. Plans are meant to represent a description of a task to be done. The UA2 language defines a set of primitive plans of the following types:

- **Search** – searches a specified area using an onboard sensor
- **UseSonar** – activates a sonar device onboard
- **UseModem** – activates an onboard communication device, such as an acoustic modem
- **Transit** – requests the vehicle to move through a set of waypoints
- **PhoneHome** – sends status reports back to a base 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

Users describe the task they want to do by declaring instances of these plans, called ***plan instances***. Plan instances can be thought of as tasks or jobs or problems to be solved, where the task description is based on its type. For example, if one wants a AUV to move to some waypoint, they can achieve this with the following code:

```
Transit goToSomePosition(Destination = GeoPosition(Lat=Degrees(40.0), Lon=Degrees(-122.0), Depth=Feet(10)))
```

In this example, we create a plan instance of type **Transit**, and with the name *goToSomePosition.* The red highlighting indicates keywords recognized by the language. Following the name is a list of parameters enclosed in parentheses. This is referred to as the ***plan instance constructor.*** This constructor describes the parameters for the plan instance. In the case of the example above, there is one parameter, the "Destination". Hence, the instance *goToSomePosition* represents transiting to a geographical position of 40° N, 122° W, and a depth of 10 feet.

Plan Instance Constructors

It can be inferred from this that every type of primitive plan in UA2 has a constructor associated with it. Each constructor is different for each type of primitive plan. For example, the **Search** plan takes parameters such as 'the area to be searched' and 'the sensor to search with'. Each type of plan may have more than one constructor, allowing several ways to define the problem parameters. We do not go into detail of what each constructor looks like here, as that can be found in Appendix I of this document.

Strong Type System and Explicit Parameter Naming

By now, you have probably noticed the convoluted syntax in the constructor above. The UA2 language is designed to ensure there is no ambiguity in what the user wants to do, and uses a very strong type system to accomplish this. The constructor requires each parameter to be named (in the above case only one parameter called Destination is required for a Transit constructor) followed by an '=', followed by the value. The ***kind*** of value expected for the parameter Destination is a ***Position***. A kind is simply a "type of type". A *Position* can be defined in several ways, or ***types***, one is a **GeoPosition**, which refers to an absolute geographical position, which is what is used above.

GeoPosition, just like a plan instance, has a constructor, as do all other types in the language. **GeoPosition** itself 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 kind ***Angle*** which can take a value of the **Degrees** type. Depth requires a value of kind ***Length**,* which can take **Feet** as a type. Notice the constructors for **Degrees** and **Feet** simply take a number, indicating the termination of nested constructors.

The notion of a kind and types is found in many languages, and is essentially equivalent to what is called a ***tagged union*** (also known as a ***variant type, discriminated union, algebraic data type*** or ***disjoint set***). In a tagged union, the union represents a type that can take on values of several different types. We use the concept of a *kind* to represent a indicate a set of *types* that can be used wherever a value of that *kind* is expected.

The following kinds and types are used within plan constructors as well as device declarations (which we will see in section 7).

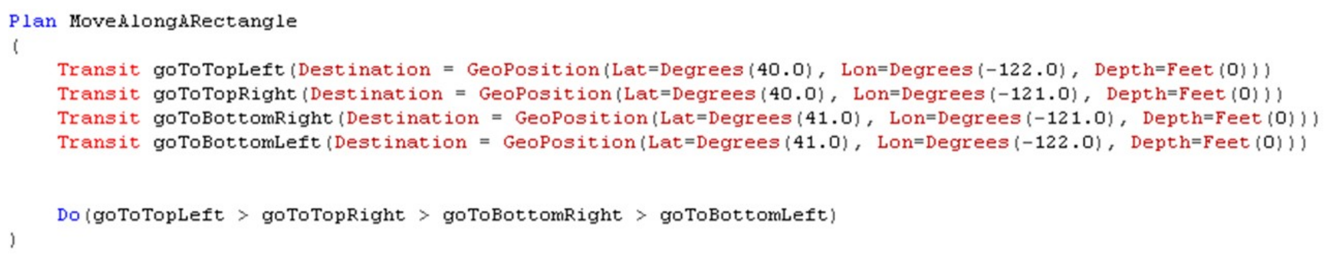
Kind	**Types**
Angle	**Degrees**, **Radians**
Area	**RectangularArea**, **CircularArea**, **PolygonalArea**
Duration	**Seconds**, **Minutes**, **Hours**
Energy	**Joules**, **KilowattHours**
Frequency	**Hertz**, **Kilohertz**
Length	**Feet**, **Meters**, **Yards**
Position	**GeoPosition**, **RelativePosition**
Power	**Watts**, **Kilowatts**, **Horsepower**
Time	**UnixTime**, **DHMSMTime**
*	Boolean
*	Integer (32-bit signed)
*	Float (64-bit double precision)
*	String (ASCII)

Note the last 4 types have no kind, they are the most basic, primitive types in the language, and compose the other types. The type constructor definitions are described in Appendix I, along with the constructor definitions for plan instance declarations.

2. User-defined Plan Types and the SortiePlan

Plan instances have to be declared within a ***user-defined plan,*** which defines a user created plan type. User-defined plan types are similar to primitive plan types, in the sense that they represent some sort of task description, but they can be composed of one or more plan instances.

An example of a user-defined plan is as follows:



```
Plan MoveAlongARectangle
(
    Transit goToTopLeft(Destination = GeoPosition(Lat=Degrees(40.0), Lon=Degrees(-122.0), Depth=Feet(0)))
    Transit goToTopRight(Destination = GeoPosition(Lat=Degrees(40.0), Lon=Degrees(-121.0), Depth=Feet(0)))
    Transit goToBottomRight(Destination = GeoPosition(Lat=Degrees(41.0), Lon=Degrees(-121.0), Depth=Feet(0)))
    Transit goToBottomLeft(Destination = GeoPosition(Lat=Degrees(41.0), Lon=Degrees(-122.0), Depth=Feet(0)))


    Do(goToTopLeft > goToTopRight > goToBottomRight > goToBottomLeft)
)
```

The plan consists of the following:

- The keyword **Plan** followed by a name for the plan, in this case the name is *MoveAlongARectangle*.
- Following this is the ***plan body*** enclosed within parentheses. The body begins with the plan instance declarations that constitute that plan. In this case, we have 4 plan instances of the plan type Transit.
- Finally, there is an expression known as the ***Do Expression*** which we will see in section 3, ignore this for now.

Creating Plan Instances from User-Defined Plans

The types of plan instances we have created so far are primitive plan types. We can create instances of user-defined plan types by using the special keyword: **ExecutePlan.**

Here is an example of another plan that uses the plan declared above.

```
Plan MoveAroundThenGoHome
(
    ExecutePlan moveAround(MoveAlongRectangle)
    Transit     goHome(Destination = GeoPosition(Lat=Degrees(41.7), Lon=Degrees(-124.5), Depth=Feet(0)))

    Do(moveAround > goHome)
)
```

The **ExecutePlan** keyword creates an instance of the type passed into the instance constructor. In the example above, we create a plan instance with plan type *MoveAlongRectangle* (which we declared earlier), called *moveAround.* The plan instance *moveAround* acts just like any primitive plan instance.

The astute reader will probably infer that user-defined plans are the primary means of abstraction in the UA2 language, and allow a user to organize their missions in a logical way.

Children Plan Instances

The sub-plan instances that compose an instance of user-defined plan are known as that instance's ***children***. This term will become more important in later sections. In the example above, the instance *moveAround* has the children *goToTopLeft, goToTopRight, goToBottomRight,* and *goToBottomLeft*.

Multiple Plan Instances

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.

Circular Dependencies

The UA2 language requires that in order to create an instance of a user-defined plan, the user-defined plan must already be declared. This means it is not possible to create a circular dependency.

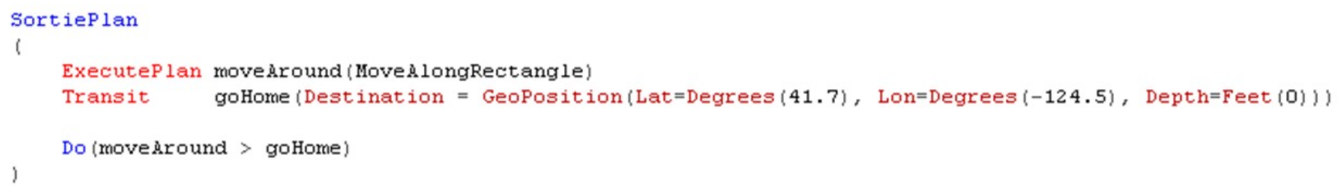
The Sortie Plan

The UA2 compiler/runtime requires at least one user-defined plan to be declared, called the ***sortie plan.*** This can be thought of as the master task description for what the user wants the AUV to do. The word *sortie* comes from military aviation, and refers to the period where a vehicle/vessel is away from its base ship/port performing a mission (TODO: check if this definition is correct).

The sortie plan is declared by using the keyword, **SortiePlan**. This plan is no different from any other user-defined plan except that it:

1. Does not take a name (the name is implicitly “SortiePlan”),
2. No other user-defined plans can create an instance of it (which is obvious as no circular dependencies are allowed)
3. The UA2 runtime implicitly creates one, and only one plan instance (also known as a ***singleton***) of this type. This instance is known as the the ***sortie plan instance***, or simply the ***sortie instance***.

We can make our plan above, *MoveAroundThenGoHome* into the sortie plan by simply replacing the text “Plan MoveAroundThenGoHome” with the text “SortiePlan”, as follows:



```
SortiePlan
(
    ExecutePlan moveAround(MoveAlongRectangle)
    Transit     goHome(Destination = GeoPosition(Lat=Degrees(41.7), Lon=Degrees(-124.5), Depth=Feet(0)))

    Do(moveAround > goHome)
)
```

3. The Do Expression and Planning Operators

As we saw in earlier examples, every user-defined plan declaration contains within it an expression that starts with the keyword **Do**. This expression is known as the ***Do Expression.*** The Do Expression tells the UA2 runtime how we want to organize our plan instances (or problems/tasks/jobs if you will) in respect to the time domain. In this example from the previous section, we have the do expression:

```
Do(moveAround > goHome)
```

The “>” sign between the names of the two plan instances *moveAround* and *goHome*, is one of the many types of ***planning operators,*** in this case, it is the ***serial planning operator.*** What the serial planning operator tells the UA2 runtime is:

“I want you to do the task *moveAround* before you do the task *goHome*”.

The UA2 runtime will ensure that this happens, and that the task *goHome* will only occur once *moveAround* is complete.

Types of Planning Operators

The UA2 language has several planning operators that can be used in the Do Expression. These operators are left associative, but can be grouped using parentheses as well.

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

Here are some examples of more complex Do Expressions using these operators:

```
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 do either do the tasks *a* followed by *b*, OR do the tasks *c* and *d* in parallel (only one or the other). While it's doing 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, do task *d*.

We can see from these examples that one can create significantly complex relationships between the tasks needed to be done in the AUV.

A Caveat of the Do Expression

An important note about the Do Expression is that one is not allowed to reference a plan instance in the expression more than once, as we have not been able to create a semantic model for this yet. This feature, however, is not completely forgotten, and is on our back-burner of things to do.

4. Run-time Value Lookups

It is obvious that the parameters to plan instance constructors and device declarations (seen later) may not be known at compile time, or in other words, cannot be statically determined. The UA2 language provides the ability to substitute values in code with a ***lazy-lookup call***. For example, in our SortiePlan example from the last section, we can replace the hard-coded values for latitude and longitude with the following:

```
SortiePlan
(
    ExecutePlan moveAround(MoveAlongRectangle)
    Transit     goHome(Destination = GeoPosition(Lat=Degrees(LookupFloat("HomeLatitude")),
                                                 Lon=Degrees(LookupFloat("HomeLongitude")),
                                                 Depth=Feet(0)))

    Do(moveAround > goHome)
)
```

The constructor for Degrees expects a floating point value. The **LookupFloat** keyword allows a user to lookup a double-precision (64-bit) floating point value in a lazy fashion. Whenever the UA2 runtime requires the values for latitude and longitude, it will do a lookup on the keys “HomeLatitude” and HomeLongitude”. The compiler ensures that there is type safety as using any other type of lookup call will cause a type mismatch error.

There are currently four different Lookup*() calls, but our plan is to add more for every type mentioned in section 1.

Type	Lookup Callname
Float (64-bit double precision)	**LookupFloat**
Integer (32-bit signed)	**LookupInteger**
Boolean	**LookupBoolean**
String (ASCII)	**LookupString**

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. Therefore, it cannot be verified until there is a lookup. The UA2 compiler's static verifier ensures that literal values are within range. For example, a latitude of 800 does not make sense, it has to be between 90 and -90. This can cause the UA2 runtime to throw an exception.

- 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 AUV maybe more difficult to predict.

- If the key is not defined in the ***knowledge base*** (see chapter 4 and 5), it will cause the UA2 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.

5. Conditional Expressions

Besides the planning operators, users can use ***conditional*** **(if-then-else-endif)** ***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 UA2 runtime make the decisions. The following example illustrates the concept:

```
SortiePlan
(
    ExecutePlan moveAround(MoveAlongRectangle)
    Transit     goHomeToBase1(Destination = GeoPosition(Lat=Degrees(LookupFloat("Base1Latitude")),
                                                     Lon=Degrees(LookupFloat("Base1Longitude")),
                                                     Depth=Feet(0)))
    Transit     goHomeToBase2(Destination = GeoPosition(Lat=Degrees(LookupFloat("Base2Latitude")),
                                                     Lon=Degrees(LookupFloat("Base2Longitude")),
                                                     Depth=Feet(0)))
    Do(moveAround >
       if(LookupBoolean("DistanceToBase1ShorterThanBase2")) then
           goHomeToBase1
       else
           goHomeToBase2
       endif
      )
)
```

Usage
The conditional expression looks similar to one from any run-of-the-mill programming language. The syntax is:

```
if(<condition>) then (<planExpression>) else (<planExpression>) endif
```

The planExpression in each branch of the conditional expression can be more complex than simply a reference to a plan instance as in the example, but rather can contain the usage of multiple planning operators. For example

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

Meaning
In the example above, the serial operator is used to do the task *moveAround* followed by one of the tasks *goHomeToBase1* or *goHomeToBase2.* The decision on which one is done is based on the lookup value "DistanceToBase1ShorterThanBase2". If this is true, than *goHomeToBase1* is chosen, otherwise, *goHomeToBase2* is chosen.

It is possible that the lookup call can change during run-time, as stated in the previous section. In this case, one task is abandoned, and the other is started. It could be possible that UA2 (and in turn the AUV) could constantly switch between the two tasks, depending on where it is.

Comparison Operators
For values used in the <condition> part of the if expression, it is possible to do a comparison between values. The UA2 language currently only defines comparisons between floating point numbers, integers, and strings.

Operator	**Syntax**
Greater than	>
Greater than or equal to	>=
Equal to	==
Less than	<

Less than or equal to	<=

If (You Can Avoid Them) Then (Do so) Else (Use Them) Endif

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 UA2 runtime into the hands of the user. Why give yourself more responsibility? The goal of the UA2 language and runtime is to allow users not to focus too much on *how* the missions are to be executed, but rather *what* missions 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 your vehicle's mission. For example, in the example above, an xor operator maybe sufficient:

```
SortiePlan
(
    ExecutePlan moveAround(MoveAlongRectangle)
    Transit     goHomeToBase1(Destination = GeoPosition(Lat=Degrees(LookupFloat("Base1Latitude")),
                                            Lon=Degrees(LookupFloat("Base1Longitude")),
                                            Depth=Feet(0)))
    Transit     goHomeToBase2(Destination = GeoPosition(Lat=Degrees(LookupFloat("Base2Latitude")),
                                            Lon=Degrees(LookupFloat("Base2Longitude")),
                                            Depth=Feet(0)))
    Do(moveAround > (goHomeToBase1 ^ goHomeToBase2)
)
```

In this case, the UA2 runtime will handle which base to go home to. The semantics of the xor operator dictate that the runtime will attempt to go to base 1 first, if that's not possible, go to base 2. Of course, this maybe less efficient than using the conditional version above, but will nonetheless accomplish the mission if you really do not care which base it goes the AUV goes to, as long as it gets to one. The runtime may find that going base 1 requires too much time or energy, and will dismiss it immediately. Of course, there is no way to know which one it chooses and how it chooses it until it actually does it. It's up to the user to determine which one works best for her or him.

6. Building and Binding Constraints

If it is not evident by now, the UA2 language is a ***constraint-based language***, where 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 plan instance constructor. The language runtime is responsible for solving the problems based on these constraints.

The UA2 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 in an example:

```
Plan LoiterUntilSubmarineArrives
(
    Loiter         doNothing(LoiterPosition = GeoPosition(Lat=Degrees(31.864),Lon=Degrees(-120.4574), Depth=Feet(32.53)))
    ExecutePlan    moveAround(MoveAlongRectangle)
    TimeConstraint timeLimit(DHMSMTime(Hours = 4, Minutes = 30, Seconds = 0, Milliseconds = 0)
                       <= StartTime <= DHMSMTime(Hours = 5, Minutes = 0, Seconds = 0, Milliseconds = 0),
                       DHMSMTime(Hours = 4, Minutes = 30, Seconds = 0, Milliseconds = 0)
                       <= EndTime <= DHMSMTime(Hours = 6, Minutes = 0, Seconds = 0, Milliseconds = 0),

    Do(moveAround with timeLimit > doNothing)
)
```

In this example, we construct an instance of a **TimeConstraint** named *timeLimit*. We bind the constraint to the plan instance *moveAround* using the special operator, **with**. The **with** operator, like the planning operators, is left associative.

Meaning of a Constraint

When a plan instance is bound to a constraint, it means that the UA2 runtime must execute the task specified by that instance under the limitations of that constraint. The time constraint in the example tells us that the task *moveAround* must start at some time between 0430 hours and 0500 hours, and must end anywhere between 0430 hours and 0600 hours.

Binding to an Entire Subexpression

The **with** operator allows binding a constraint to any subexpression. For example, if we had the following do expression:

```
Do((moveAround > doNothing) with timeLimit)
```

All the plan instances within the subexpression (both *moveAround* and *doNothing*) are bound to the restrictions of the constraint *timeLimit*.

Types of Constraint

There are three types of constraints currently in the UA2 language. As we develop the language, we will add more:

Constraint Type	**Meaning**
TimeConstraint	The tasks bound to this constraint are required to start and end in the time window specified by the constraint.
PowerConstraint	The tasks bound to this constraint are required to use no more energy (i.e. fuel, battery) than specified by this constraint, as well as not to exceed the maximum power load specified by the constraint at any given instance in time.
AreaConstraint	Tasks bound to this constraint are not allowed to enter areas specified by this constraint.

The core UA2 runtime only enforces TimeConstraints currently, the other constraints are expected to be enforced by planners (which will be seen in chapters 4 and 5). We are working on a way to enforce these constraints within the core runtime.

The constructors for these constraints, along with the other types in the language, can be found in Appendix I of this document.

Binding Multiple Constraints

It is possible to bind multiple constraints, of both different types or the same type, to a plan instance by chaining the **with** operator repetitively. The following example illustrates (note that the constructor definitions are ignored for simplicity and replaced with a '-').

```
Search a(-)
Transit b(-)
TimeConstraint tc1(-)
TimeConstraint tc2(-)
PowerConstraint pc(-)
Do((a > (b with tc1)) with tc2 with pc)
```

What happens in this example is that *b* gets bound *tc1*, *(a > b)* gets bound to *tc2*, and *(a > b)* gets bound to *pc*. This means that *a* is bound to *tc2* and *pc,* and *b* is bound to *tc1*, *tc2*, and *pc*, and *b* . In the case of instance *a*, the UA2 runtime will enforce both *tc2* and *pc* as expected. However, instance *b* is the interesting one, as it has two constraints of the same type.

When UA2 encounters constraints of the same type, it takes the ***set intersection*** of them. In the case of tc1 and tc2, the intersection is the intersection of the time windows specified by those constraints. It is possible the intersection could be empty, in which case accomplishing that task is impossible. When constraints are too tight for the runtime to do anything, it causes an error, discussed in section 8 of this chapter.

Constraints are Hierarchical

When constraints are bound to instances of user-defined plans (i.e. instances of type **ExecutePlan**), those constraints are implicity bound to all it's children instances.

7. Devices

In some of the plan instance constructors (**Search**, **UseSonar**, **UseModem**, **PhoneHome**), one or more of the parameters refer to the name of a device, such as a sonar or a modem. The UA2 runtime requires some of fundamental understanding of these devices in order to appropriately control the vehicle. For example, it is important to know information about the onboard sonar such as the frequency bands in which it operates and how much power it utilizes.

The UA2 language allows declaring two kinds of onboard devices, ***Sonars*** and ***Modems***.

TODO: Finish this section

8. Handling Errors – Infeasibilities and Conflicts

One of the issues that may have crossed your mind reading this document is what does UA2 do, if anything, when it cannot perform the tasks for some reason specified by the user in their mission specification. The UA2 language does indeed have constructs for handling errors. There are two types of errors that can occur during task execution: ***infeasibility errors*** and ***conflict errors***.

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 plan instance of type **Transit** requesting the vehicle move from Chesapeake Bay to Hawaii with a time constraint of an hour, would be an impossible task. This task could be impossible for several reasons, obviously time is one of the issues. Another could be that the vehicle does not have enough power to accomplish the mission.

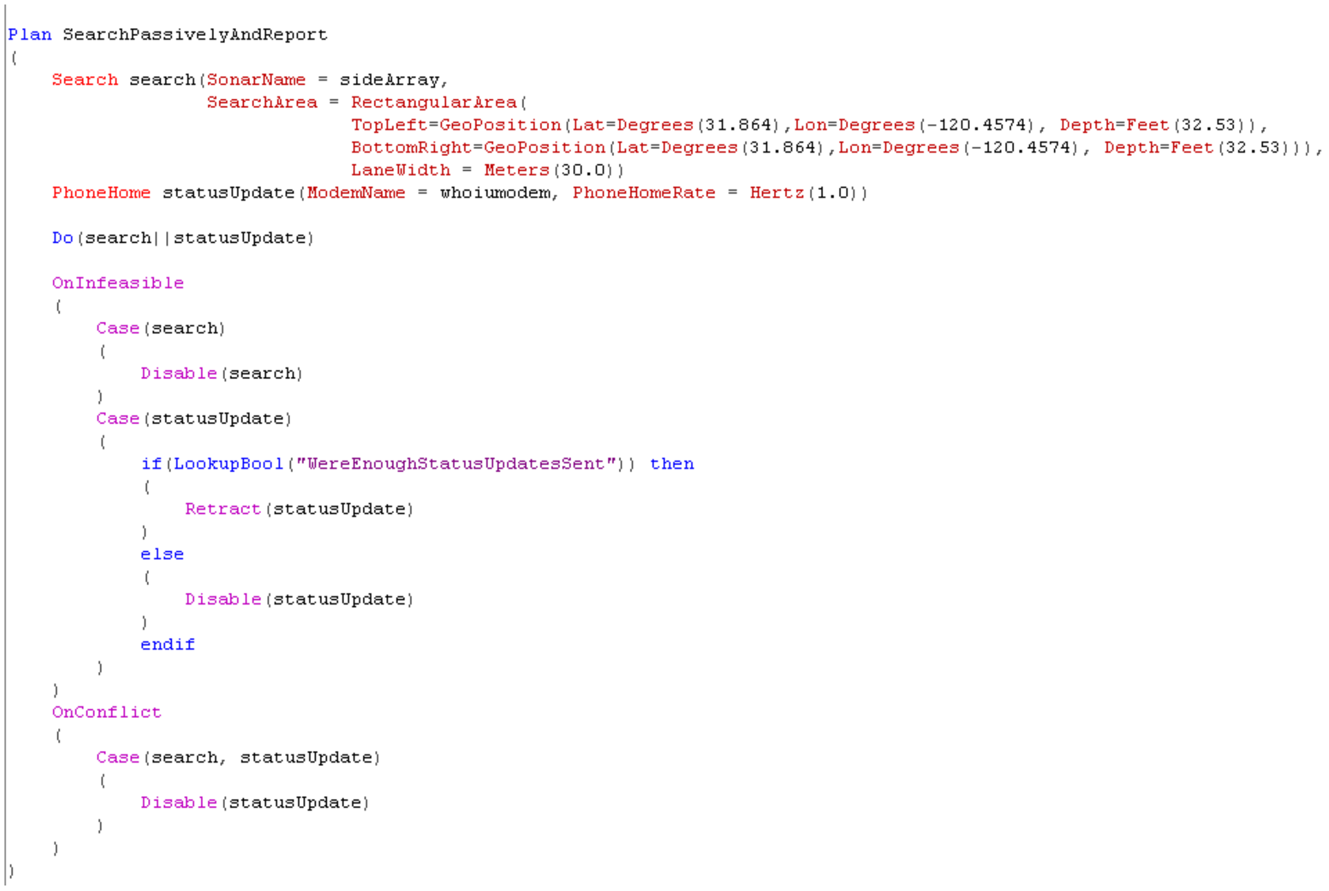
Conflict Errors

Conflict errors, in contrast to infeasibility errors, come into play when the AUV cannot do two or more requested tasks at the same time, and there is a 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.

Exception Handlers

When infeasibilities and conflicts occur, it causes an ***exception*** to be raised which is expected to be caught in the language.

To illustrate, observe the following example:



```
Plan SearchPassivelyAndReport
(
    Search search(SonarName = sideArray,
                  SearchArea = RectangularArea(
                               TopLeft=GeoPosition(Lat=Degrees(31.864),Lon=Degrees(-120.4574), Depth=Feet(32.53)),
                               BottomRight=GeoPosition(Lat=Degrees(31.864),Lon=Degrees(-120.4574), Depth=Feet(32.53))),
                               LaneWidth = Meters(30.0))
    PhoneHome statusUpdate(ModemName = whoiumodem, PhoneHomeRate = Hertz(1.0))

    Do(search||statusUpdate)

    OnInfeasible
    (
        Case(search)
        (
            Disable(search)
        )
        Case(statusUpdate)
        (
            if(LookupBool("WereEnoughStatusUpdatesSent")) then
            (
                Retract(statusUpdate)
            )
            else
            (
                Disable(statusUpdate)
            )
            endif
        )
    )
    OnConflict
    (
        Case(search, statusUpdate)
        (
            Disable(statusUpdate)
        )
    )
)
```

In the example, we have two plan instances, *search* and *statusUpdate*. The Do Expression of this plan wants to run these two tasks in parallel. The UA2 language allows a user to define two handlers in each user-plan declaration using the keywords: **OnInfeasible** and **OnConflict**. These handlers are essentially ***call-back functions*** that are invoked when an infeasibility or conflict occurs respectively. The user is expected to make a decision in each of these handlers to describe what she or he wants to do about the infeasible or conflicting instances. In either case, the user is intended to either temporarily **Disable** or permanently **Retract** one or more the plan instances causing a problem.

Case Signature Matching

Notice that each of the handlers consist of one or more ***case expressions***. Cases are used to match up which handling code to execute when an infeasibility or conflict occurs. In the case of infeasibilities, the argument following each case keyword is known as the ***case signature***, which is the name of the instance causing the infeasibility. In the case of conflicts, the case signature it is a list of the instances that cause the conflict, separated by comments. Note that for this list, the ordering does not matter, the UA2 runtime will figure it out.

Handler Expression

The part of the case expression following the signature enclosed in parentheses is called the ***case body****.* The case body contains within it a ***handler expression***. The handler expression when evaluated is supposed to retract or disable at least one of the arguments in the case signature (there is only one argument in infeasible cases). The UA2 compiler will statically verify that this requirement is met, if it is not, an error will be indicated.

Note that the use of conditional expressions is permitted in the handler expression.

Disable vs Retract

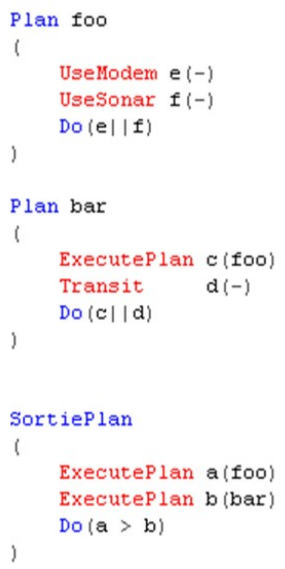
The difference between Disable and Retract is as follows. Disable means the user wants to *temporarily* ignore the task so that UA2 runtime can solve other problems, but will later attempt again to solve the problem. Retract means the user wants to permanently remove the problem from consideration, as if it never existed. In the example above, the plan instance *statusUpdate* is retracted in the **OnInfeasible** handler only when enough “status updates have been sent”, otherwise it will temporarily disable the task.

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 SortiePlan, the exception is classified as an ***unhandled exception,*** and causes the runtime to error (see chapters 4 and 5 for more information about this).

Case Pattern Matching on Plan Instance Chains

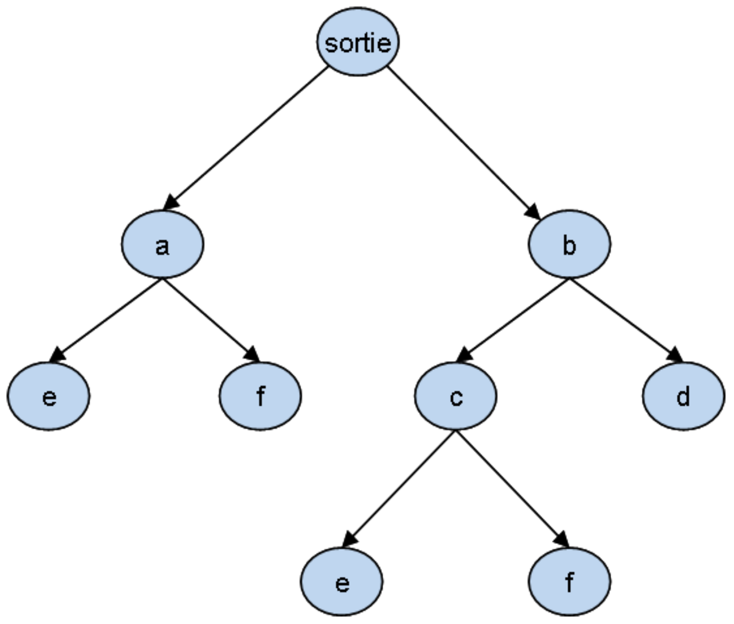
In each case expression, the case signature is actually not matching on the name of a plan instance, but rather it is matching against the ***plan instance chain*** of that instance. Every plan instance in the UA2 runtime can be uniquely identified as a part of a chain that starts from a root instance: the sortie instance (mentioned in section 2). For example, look at the following sample of code (constructors for plan instances are ignored, except for the **ExecutePlan** instances)



```
Plan foo
(
    UseModem e(-)
    UseSonar f(-)
    Do(e||f)
)

Plan bar
(
    ExecutePlan c(foo)
    Transit     d(-)
    Do(c||d)
)


SortiePlan
(
    ExecutePlan a(foo)
    ExecutePlan b(bar)
    Do(a > b)
)
```

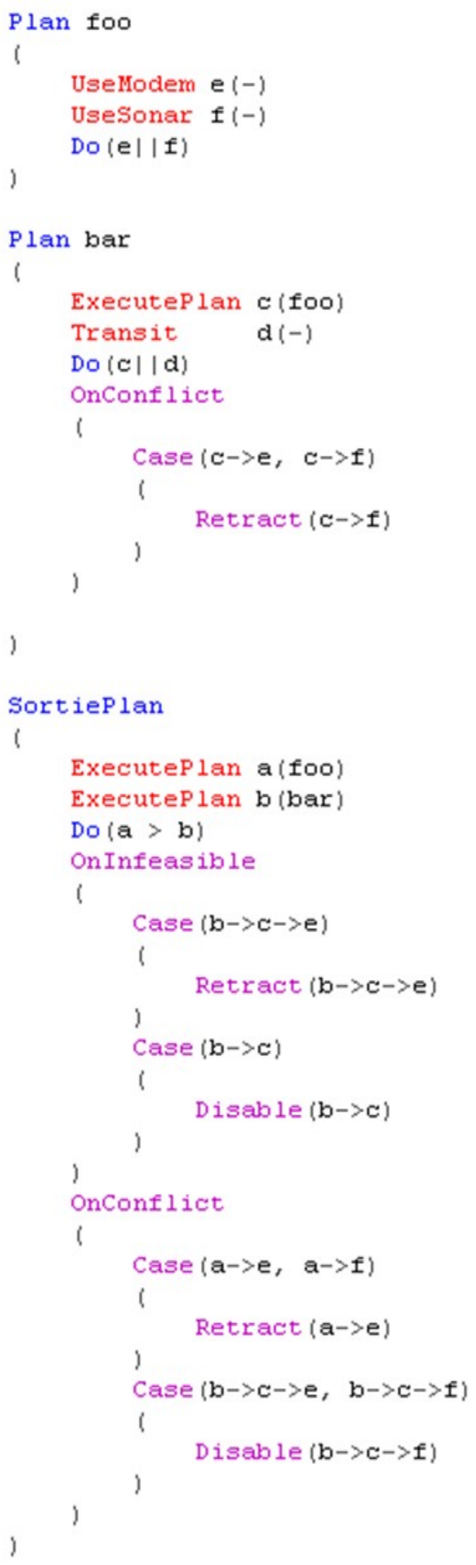


Each plan instance can be uniquely identified as part of a chain, designated by the arrows. Even though instances of the plan type *foo* were created twice, we instance within it can uniquely identify each one using the chain notation:

```
sortie->a->e
sortie->a->f

sortie->b->c->e
sortie->b->c->f
```

In case signatures, one can match on these chains. The reason this feature is provided is because the context of why an infeasibility or conflict occurred in a plan may not be known. In the example above, the instances *e* and *f* within the plan *foo* are part of two unique chains respectively. It may not be suitable to have handlers at the *foo*-level. To illustrate, here is another example extending the one above:



```
Plan foo
(
    UseModem e(-)
    UseSonar f(-)
    Do(e||f)
)

Plan bar
(
    ExecutePlan c(foo)
    Transit     d(-)
    Do(c||d)
    OnConflict
    (
        Case(c->e, c->f)
        (
            Retract(c->f)
        )
    )

)

SortiePlan
(
    ExecutePlan a(foo)
    ExecutePlan b(bar)
    Do(a > b)
    OnInfeasible
    (
        Case(b->c->e)
        (
            Retract(b->c->e)
        )
        Case(b->c)
        (
            Disable(b->c)
        )
    )
    OnConflict
    (
        Case(a->e, a->f)
        (
            Retract(a->e)
        )
        Case(b->c->e, b->c->f)
        (
            Disable(b->c->f)
        )
    )
)
```

The example to left adds in handlers to the example above. Note that in the plan *bar* and the sortie plan, that the case signatures can match against a chain. This gives flexibility to the user as they have a context of which error happened. A user may want to handle an error in the instance sortie->b->c->e differently than the instance sortie->a->e.

Matching Rules

The UA2 runtime matches the *first valid* case signature when running handler code. For example, the OnInfeasible in the SortiePlan handles the cases *b->c->e* and *b->c*. If the instance *b->c->e* were to cause an infeasibility, any of the following signatures would be suitable:

b->c->e
b->c
b

The UA2 runtime though will match them up in the order declared. Users will most likely want to declare their handlers in order from most specific to least specific. The reason UA2 does this rather than automatically match up the most specific is made clear with conflict errors. With conflicts, the same issue applies in the matching rules, however, it is difficult to judge which signature would be "the most specific". Rather than dictate some arbitrary set of rules, we went with letting the user decide.

9. Conclusion

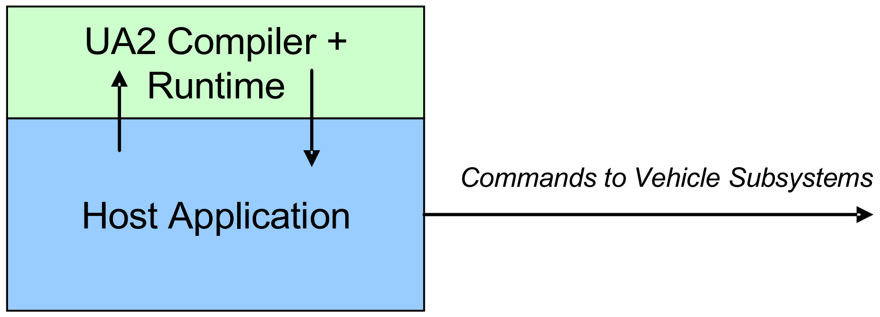
This concludes the chapter on the UA2 mission specification language. It is evident by now how powerful using a programming language can be as a means of control for an AUV. The language allows one to think about the task at hand, planning missions for AUVs, and nothing else. The language also brings with it several concepts from programming languages such as strong typing, a powerful way of handling system failures (exceptions), and a uniform and intuitive way of reasoning about the problem.

Understanding the language is important to understanding how to build components for it. In the next two chapters, we will peek inside of the UA2 runtime, and how it accomplishes the tasks specified in the mission specification language. As before, the language is just a means to controlling the vehicle once all the software is in place, but the UA2 runtime requires special modules, called *planners*, that hook into it to perform primitive operations of the AUV. We will see that these primitive operations coincide with the primitive plan types we saw in this chapter.

Chapter 4
How to Use UA2

1. Using UA2 in Your Host Application

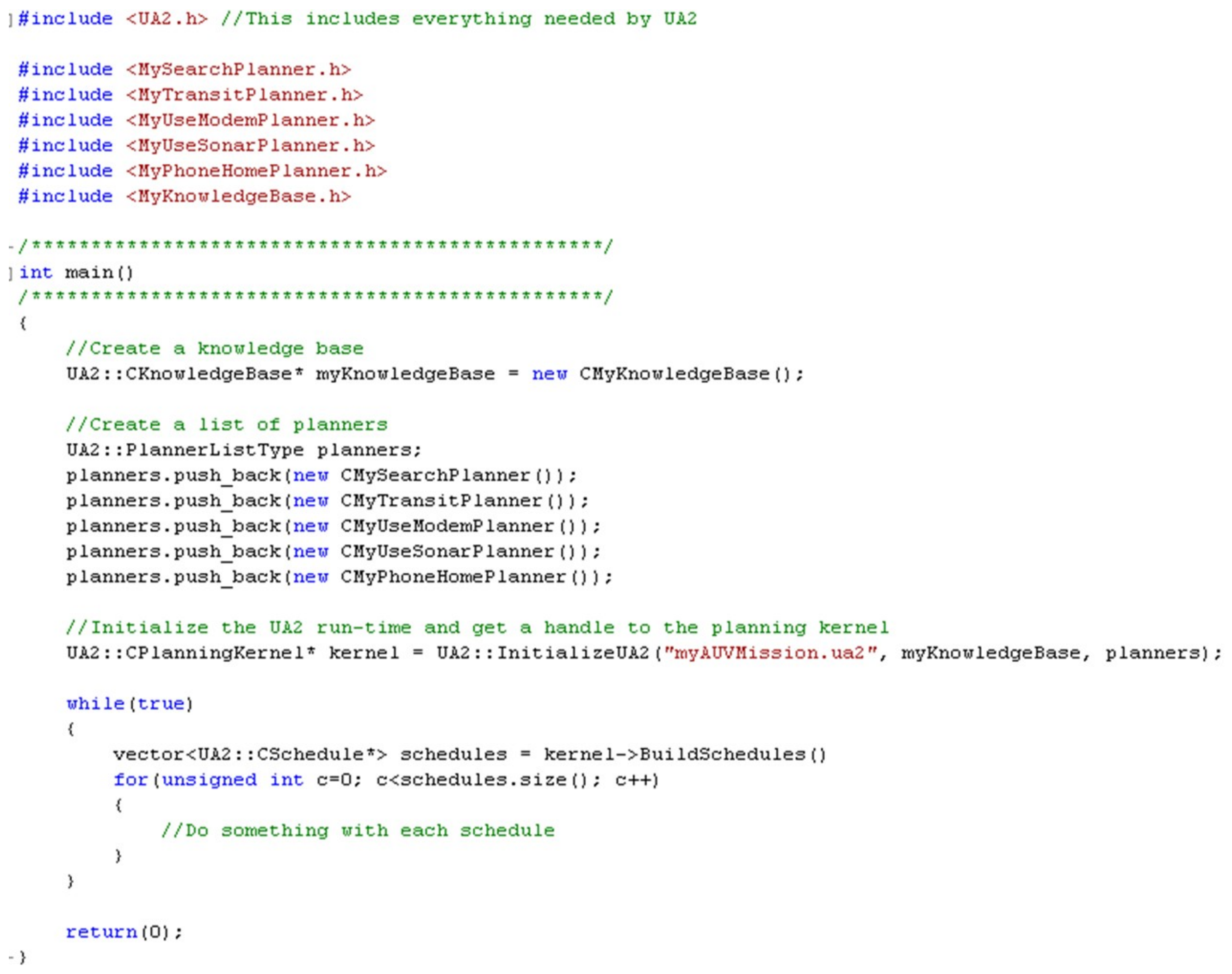
UA2 itself is not a stand-alone application. An AUV developer is expected to link it into a host application that will turn around and call UA2. The host application acts as glue between UA2 and the vehicle effectors.



An AUV developer is expected write a host application that is intended to be the means of talking to all of the vehicle subsystems. The host application is supposed to use UA2 as follows:

1. Initialize the UA2 compiler/runtime by feeding it a mission file written in the UA2 language (from chapter 3), a ***knowledge base***, and a set of ***planners***. (The knowledge base and planners are discussed in subsequent sections.) The initialization of UA2 returns a handle to a C++ object called the ***planning kernel**,* which is the primary means for the host application to talk to the UA2 runtime.
2. Invoke the UA2 runtime by calling the planning kernel's BuildSchedules() method. This returns a list of ***schedules**.* (Schedules are discussed in subsequent sections). Schedules contain timestamped commands for various vehicle subsystems such as the autopilot or onboard sensors.
3. The host application iterates through these schedules, and issues commands to the vehicle subsystems based on what is contained within these schedules.
4. Go to step 2 and repeat. The host is expected to constantly stimulate the UA2 runtime in order to update schedules. Schedules can change with time as the situation changes.

The following C++ example illustrates how UA2 is intended to be used:



```
#include <UA2.h> //This includes everything needed by UA2

#include <MySearchPlanner.h>
#include <MyTransitPlanner.h>
#include <MyUseModemPlanner.h>
#include <MyUseSonarPlanner.h>
#include <MyPhoneHomePlanner.h>
#include <MyKnowledgeBase.h>

/************************************************/
int main()
/************************************************/
{
    //Create a knowledge base
    UA2::CKnowledgeBase* myKnowledgeBase = new CMyKnowledgeBase();

    //Create a list of planners
    UA2::PlannerListType planners;
    planners.push_back(new CMySearchPlanner());
    planners.push_back(new CMyTransitPlanner());
    planners.push_back(new CMyUseModemPlanner());
    planners.push_back(new CMyUseSonarPlanner());
    planners.push_back(new CMyPhoneHomePlanner());

    //Initialize the UA2 run-time and get a handle to the planning kernel
    UA2::CPlanningKernel* kernel = UA2::InitializeUA2("myAUVMission.ua2", myKnowledgeBase, planners);

    while(true)
    {
        vector<UA2::CSchedule*> schedules = kernel->BuildSchedules()
        for(unsigned int c=0; c<schedules.size(); c++)
        {
            //Do something with each schedule
        }
    }

    return(0);
}
```

Include Files and Namespaces
In order to use UA2, one includes the file UA2.h to get all necessary declarations needed by the UA2 core library. The other include files in this example are meant to contain the declarations for the planners/knowledge base built by either a AUV developer or some other 3rd party. All of the UA2 core classes are declared within the namespace "UA2", as can be seen in the example.

The InitializeUA2() Call
The IntializeUA2 call is what starts up the library, and actually invokes the parser and static verifier. The first parameter to this call is the name of a text file that contains code written in the UA2 language from Chapter 3. This describes the mission we want the UA2 runtime to help us achieve. The second is a handle to a knowledge base, which is described in section 3, and finally a list of handles to planners, which are described in section 4. Once this call is complete, it returns a handle to the planning kernel, described in section 2.

The InitializeUA2 call can fail, and can potentially throw a C++ exception. For simplicity this is ignored, but we discuss errors in section 5.

2. The Planning Kernel and Schedules

The InitializeUA2() call returns a handle to a module of type CPlanningKernel. This object is known as the ***planning kernel***, which is the primary means to talk to the UA2 runtime by the host application. Once the handle is returned, UA2 is ready for action.

The BuildSchedules() Call
The goal of the UA2 is to translate the input mission file written in the UA2 language into a set of ***schedules*** which are meant to be followed by the host application. A schedule is simply a collection of timestamped commands. The host application is intended to stimulate the runtime every so often by invoking the *BuildSchedules()* method which returns a std::vector of schedule objects (of type *CSchedule*). Each of the schedules is bound and generated by a particular planner, and represent what that planner wants the host application to do. For example, an autopilot planner can generate a schedule which contains timestamps and waypoints. When the host application sees this schedule, they are meant to issue commands to the autopilot representing those waypoints at the times specified. If the host application does this correctly, the vehicle should do what is specified in the mission specification file. As the situation can change for the AUV and its surroundings, the schedules can change. Reinvoking the BuildSchedules() method causes schedules to be updated.

When we get to the next chapter, it will be more clear how planners and schedules work, and how one can build their own planners and schedules.

3. The Knowledge Base

In the InitializeUA2 call, it is required to pass in an object of type *CKnowledgeBase* for the second parameter. This object is known as the ***knowledge base***, which represents a common repository for all information used in the system. This knowledge can include situational awareness information for the vehicle, configuration parameters for algorithms, or whatever information is required.

The knowledge base serves two roles: 1) wherever a Lookup*() call is made in a UA2 mission file, the UA2 runtime turns around and calls the knowledge base to find the value, and 2) is available to all planners for accessing the same information.

In the next chapter, we will see how one develops a knowledge base. **An important thing to note is that planners are only plug & play between different UA2-based control systems if and only if they share the same knowledge base!** If one AUV developer develops a planner or writes a mission file that requires looking up some value *foo*, and then that planner or mission file is moved to another UA2 based configuration that does not have 'foo' defined in its knowledge base, then that planner or mission file does not have the information required, causing the system to fail.

4. Planners

Planners are the most important modules in the UA2 architecture. A planner is simply a scheduler for a specific type of problem. The goal of a planner is to look at all problems of a specific type in the system, and attempt to build a schedule of commands that solves those problems. You may notice that the names of the planners reflect the primitive plan types from chapter 3 on the UA2 language. We will see in the next chapter that planners are simply schedulers for the tasks represented by a collection of plan instances of a particular plan type.

In the example above, we create several planners and push them onto a list of type PlannerListType, which is just a typedef for std::vector<CPlanner*>. We use the vector's push_back method to add the planners to the list, and then pass it as the third parameter to the IntializeUA2 call.

5. Error Handling

In the example above, we ignored error handling. Both the InitializeUA2 call and BuildSchedules call can throw exceptions. **These exceptions indicate a fatal error that one cannot recovered from**.

There are 4 types of exceptions that can be thrown by UA2, each of which provides methods for determining what the error was and printing it out:

Compiler Exception (UA2::Exceptions::CCompilerException)
A compiler exception can be thrown during the InitializeUA2 call, and is triggered if there is a problem in the source code written in the UA2 language. The UA2 compiler will catch any syntax errors in the code, as well as

any other static errors such as undeclared reference, violation of scoping rules, violation of the type system, values out of range, and more.

Panic Exception (UA2::Exceptions::CPanicException)
A panic exception can be thrown by any call in the UA2 system. The word “panic” comes from the UNIX operating system, where when things went wrong, the OS kernel can panic, and indicate to the user that the system is in a bad state, and needs to be restarted. Panic exceptions are found all throughout the UA2 system, and are used to check the integrity of the system.

To ensure that the implementation is bug-free as possible, our implementation of the UA2 run-time has consistency checks throughout the code. If something isn't consistent, it causes a panic exception to be thrown. If this is to happen, it indicates a bug on our [Penn State ARL's] part, and should be brought to our attention.

Infeasibility Exceptions (UA2::Exceptions::CInfeasibilityException)
In Chapter 3, we saw that we have an exception handling mechanism in the UA2 language for handling infeasibility errors caused by plan instances. If the UA2 runtime cannot find a suitable handler at any level in the mission file, it will eventually come to this level, and inform the host application of the infeasibility.

Conflict Exception (UA2::Exceptions::CConflictException)
Just like the infeasibility exceptions, unhandled conflict exceptions in the mission file will eventually ripple up to the host application.

6. Conclusion
This concludes the chapter on how to hook UA2 into a host application, which one should find quite easy. The hard part is actually building the components: the planners and a knowledge base. The next chapter describes how the UA2 runtime works briefly, and how it utilizes these components.

Chapter 5
Building UA2 Components

1. Understanding the UA2 Computational Model

In order to effectively build components for UA2, one needs a basic understanding of how the UA2 compiler/runtime works. The runtime is complex, but it is not necessary to understanding everything in detail.

What Happens In the InitializeUA2() Call?

When a host application initializes UA2, she or he is causing several events to occur, in the following order:

Stage 1 - Planner Dependency Resolution
We have not discussed planners in detail yet, but in section 3 we will find planners can have dependencies between each other. On initialization, UA2 creates a dependency graph using a topological sort, and ensures there are no circular dependencies. If it is not possible to create a topological ordering, the UA2 runtime will raise a panic exception.

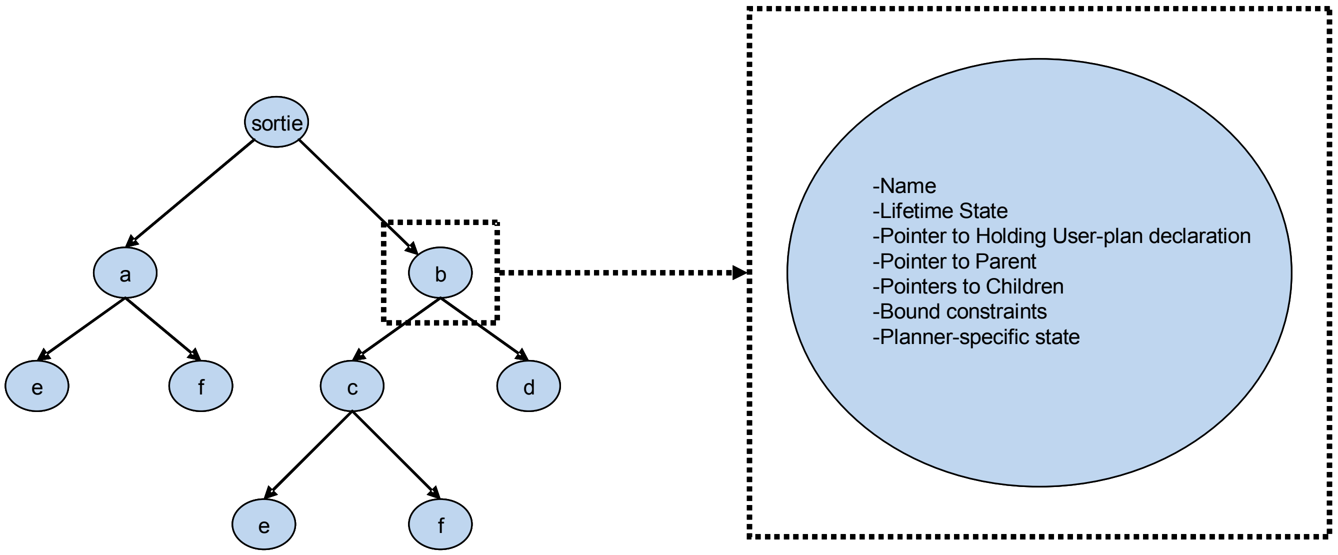
Stage 2 – Parser Invocation
The mission specification file written in the UA2 language is opened, read, and parsed. The result of this is the internal representation of the mission specification that is easy for the computer to work with. If there are any syntax errors in the code, or if the file does not exist, the parser will fire off a compiler exception.

Stage 3 – Static Verification
After the parser has succeeded and created the internal representation, that representation is then verified to ensure everything makes sense. The verifier makes sure typing and scoping rules are enforced, along with any other static rules. If the verifier fails, it will result in a compiler exception.

Stage 4 – Plan Instance Bootstrapping
After the internal representation is verified, it is used as a blueprint to construct what is known as the ***plan instance tree.*** The plan instance tree is a tree of the plan instances declared in the user's code.

Plan Instances and the Plan Instance Tree

From chapter 2, we know that plan instances are essentially the representation of a task to be done. Internally within the UA2 runtime, **plan instances are *C++ objects*** that hold internal state which is used by the runtime to execute the task that that particular plan instance represents. The illustration below shows a potential tree that could be formed after stage 4 of the InitializeUA2 call, and what state is held within a plan instance.



A plan instance contains several items as part of its internal state, but only 3 are important to a UA2 component developer, the rest are used only internally by the core part of the UA2 runtime.

Name
This refers to the name of the plan instance, and is simply a string. The name is whatever is declared in the user code.

Lifetime State
This value is simply a C++ enumeration, but is one of the most significant items in the UA2 runtime. This set of values is as follows:

Lifetime State	**Meaning**
Init	The plan instance has an undefined lifetime state, and either has just been created, or returned from a disabled state.
Ready	The task defined by the plan instance is ready to start.
Running	The task defined by the plan instance is already running.
Blocked	The task defined by the plan instance is waiting on another task for completion before it can go into a ready state.
Disabled	The task defined by the plan instance has been temporarily disabled as it caused an infeasibility or conflict error to occur.
Retracted	The task defined by the plan instance has been retracted, either by the user in an infeasibility/conflict handler, or internally by the UA2 runtime.
System Retracted	The task defined by the plan instance is retracted internally by the UA2 runtime, but maybe brought back onto the table at any point.
Complete	The task defined by the plan instance has been completed.

Some of the values may seem familiar to a person knowledgeable in operating systems theory. This is no coincidence, as UA2's design is inspired by operating systems. Operating systems like Linux and Windows manage a set of processes, which can be in one of three states, ready, running, or blocked. Ready means the process is waiting to be run by the OS scheduler, running means its currently happening, and blocked means its waiting on some event before it can be scheduled.

The UA2 runtime constantly manipulates the lifetime state, which is used to 1) indicate to planners which problems can be solved and which ones cannot, and 2) verify that planners are solving problems in a valid manner. In section 3, the use of this lifetime state will be more apparent.

Planner-specific State
Planners have the ability to attach objects that hold state specific to them for any plan instance. This state is used by the planner to keep progress of the task the plan instance represents. Section 3 of this chapter will make this clearer.

What Happens During the BuildSchedules Call()?

Everytime the BuildSchedules call is invoked, it runs in two stages:

Stage 1 – Lifetime Transitioning and Constraint Binding Evaluation
When BuildSchedules() is invoked, the first stage involves running a ***lifetime state transition*** **function** for each Do Expression, where each function is passed the respective plan instances in the respective Do Expression,

those instances' lifetime state, and the operators and constraint specified in that do expression. The resulting output of the function is a new state for each plan instance, as well as the plan instances now having constraints bound to them.

The state is important for planners, as it tells them which plan instances (or problems/tasks) to worry about scheduling, which ones should not be planned for currently (in detail anyway), which ones should be ignored, and which ones are to be temporarily ignored.

This part of the evaluation is handled by the planning kernel. It should be evident that the core part of the UA2 runtime, the kernel, has no control over the actual scheduling of any tasks, it is just a proxy for problem distribution and schedule verification.

Stage 2 – Planner Invocation
Once each plan instance has had its constraint bound, and lifetime state updated, the planners can work with them to form a schedule. Each planner has a responsibility at looking at set of primitive-type plan instances of its respective type. For example, plan instances of type **Search** are handled by a ***SearchPlanne***r, a plan instance of type **Transit** by the ***TransitPlanner***, and so on.

UA2 schedulers organize their problems in a top-down fashion. Each planner can break up their own plan instances (i.e. problems/tasks) into series of *more specific* plan instances for planners at a lower level in the dependency graph to solve. For example a, a **Search** planner can break up each search plan instance (i.e. problem) given to it into a **Transit** plan instance and **UseSonar** instance, and then ask the UA2 runtime to run it in parallel via a Do Expression. We will find that the C++ planner API provides the exact same facilities available in he UA2 language, but as a set of library calls instead! This allows one to extend and prune the plan instance graph formed in the InitializeUA2() call dynamically.

AskForSubproblems Substage
Each planner has a C++ method it is expected to override called OnAskForSubproblems(). This method is invoked in a top-down breadth-first traversal in respect to the dependency graph formed during the InitializeUA2() call. At this point, planners are asked to look at the problems given to them, and from those form any subproblems they have for the planners they specified as their dependencies. The planner API gives planners the ability to create primitive-type plan instances, constraints and do expressions dynamically, and then return them as the return value for the OnAskForSubproblems method. The UA2 runtime will verify the expression, prune/append the plan instance tree, and update the lifetime states of the new instances by reinvoking the lifetime transition evaluator.

FinalizeSchedule Substage
Once all planners have told the runtime, what subproblems they want, the runtime reinvokes the planners in the exact opposite order from the AskForSubproblems substage. In this stage, planners are asked to build a final schedule (via the OnFinalizeSchedule() method) representing the result of how they have decided to solve their respective tasks. Each planner has access to the schedules of planners lower than them in the planner dependency graph (as they may have built more specific problems in the previous substage, and need to know how they have been scheduled). Once each schedule is built, it is returned as the return value of the OnFinalizeSchedule method, and verified by the UA2 runtime to ensure it is valid. Once all planners have returned their schedules, and all the schedules have been verified by the UA2 runtime, the BuildSchedules() call is complete, bringing a single planning cycle of the UA2 architecture to a finish.

Subsequent calls to BuildSchedules() will begin again at stage 1, causing updated schedules to be built.

Infeasibilities and Conflicts
It is possible that during the OnAskForSubproblems substage, that a planner can decide the problems given to it are impossible. Special planner API calls allow a planner to tell the UA2 runtime that there is a conflict or Infeasibility, which in turn invokes the exception handling mechanism described in chapter 2.

If the plan instances that cause a problem were created by planners in the AskForSubproblems substage, the planners will be asked first to handle the problem via a handler callback in the form of an overridable virtual function (OnHandleInfeasibility and OnHandleConflict). If these methods are not overriden, or if they do not yield a decision, the UA2 runtime will look for the next handler up, which maybe at the planner level again, or in the mission specification written in the UA2 language.

2. Creating a Knowledge Base

As before, building a knowledge base is important to the UA2 runtime as it is used to bind any Lookup*() calls found in the mission specification to a value. Building a knowledge base is extremely easy. All one has to do is subclass from the class *UA2::CKnowledgeBase*, and override four pure virtual methods, which are intuitively named *LookupString*, *LookupFloat*, *LookupBoolean*, and *LookupInteger*. Each of these methods takes as an argument a *std::string*, which contains the name of the key passed in the lookup call. Knowledge base builders are meant to look at the key, and decide what value to returned based on that key. If the key is unrecognized, special methods are provided for informing the planning kernel that there is an unknown key.

KnowledgeBase Pure Virtual Methods
virtual std::string LookupString(std::string key) = 0
virtual double LookupFloat(std::string key) = 0
virtual int LookupInteger(std::string key) = 0
virtual bool LookupBoolean(std::string key) = 0

All objects in the UA2 namespace have a method for accessing the knowledge base. UAV developers should attempt to keep all their situational awareness data encapsulated within this class, and use it as a proxy for accessing situational data in their planners.

3. Creating a Planner

Building a planner is much more difficult than building a knowledge base. Creating a planner requires an understanding of scheduling algorithms, which is an entire field in itself. This document does not focus on that area, but gives some guidance on approaches to scheduling in chapter 6.

Planner Class Hierarchy

Planners are built in the same way as the knowledge base, by subclassing. All planners have a base class called *CPlanner*, but AUV developers are not intended to directly derive from this class. Rather, each type of primitive Plan has a corresponding base class to derive from. For example, to build a planner for plan instances of type Search, one derives from CSearchPlanner.

Plan Type	**Base Class Name**
Search	UA2::CSearchPlanner
UseSonar	UA2::CUseSonarPlanner
UseModem	UA2::CUseModemPlanner
Transit	UA2::CTransitPlanner
PhoneHome	UA2::CPhoneHomePlanner
UseAutopilot	UA2::CAutopilotPlanner
UseAcoustic	UA2::CAcousticPlanner

The reason specific base classes have been provided for each type is for two reasons that go hand in hand 1) the API is type safe making it more difficult to have bugs and 2) the AUV developer gets a richer set of API calls to help her or him solve problems for the particular type of planner more easily.

Accessing Problems to Solve

Planners have several methods for accessing the plan instances of the respective type for that planner. These methods allow planners to look at plan instances based on their lifetime state (from section 1). The following methods are available for each planner.

Planner Method Name	**Description**
*Problems()	Returns all problems for that planner
Ready*Problems()	Returns all problems in the “Ready” lifetime state.
Running*Problems()	Returns all problems in the “Running” lifetime state.
ForcedRunning*Problems()	Returns all problems in the “Running” lifetime state, that were forced into running via the parallel planning operator.
Disabled*Problems()	Returns all problems in the “Disabled” lifetime state.
Retracted*Problems()	Returns all problems in the “Retracted” lifetime state.
Blocked*Problems()	Returns all problems in the “Blocked” lifetime state.
SystemRetracted*Problems()	Returns all problems in the “System Retracted” lifetime state.
Complete*Problems()	Returns all problems in the “Complete” lifetime state.

Note that where the asterisk appears, one replaces the asterisk with the name of the planner type. For example, to the problems in a Search Planner, one uses the methods SearchProblems(), ReadySearchProblems(), RunningSearchProblems(), and so on.

These methods return a std::vector of pointers to the concrete subclass type of the base class *CPlanInstance*, where the type corresponds with the planner type.

Plan Instance Hierarchy

Plan instances are represented by the class *CPlanInstance*. The subclasses of this class represent the different types of plan instances one can create in UA2.

Plan Type	**Class Name**
Search	UA2::CSearchPlanInstance
UseSonar	UA2::CUseSonarPlanInstance
UseModem	UA2::CUseModemPlanInstance
Transit	UA2::CTransitPlanInstance
PhoneHome	UA2::CPhoneHomePlanInstance
UseAutopilot	UA2::CAutopilotPlanInstance
UseAcoustic	UA2::CAcousticPlanInstance

Planner developers will work with the concrete types. Each concrete class provides methods to access the problem specification defined by that plan instance. For example, the CUseModem class provides methods for accessing which onboard communications device the plan instance specifies, and the data payload to be sent by the modem. The CSearchPlanInstance class provides methods for accessing the sensor to be used, the search area, lane width, and so on.

RequiredDependencies() Method

Each planner is required to override a method called *RequiredDependencies()* to indicate which types of planners that planner depends on. The UA2 runtime will only allow a planner to create problems of the types it specifies as its dependencies. As before, circular dependencies are not allowed.

The return type of this method is a std::vector of UA2::PlanType. PlanType is simply an enumeration indicating a type, e.g. PT_SEARCH, indicates Search, PT_TRANSIT, indicates Transit, and so on.

OnAskForSubproblems() Method

From section 1, we know that the UA2 runtime invokes a method in each planner called OnAskForSubproblems(). This method is expected to return a set of new problems for planners at lower levels to solve. Planners are meant to look at the plan instances for their respective planner via the methods above, and then use the methods in those plan instances to observe the problems presented to them. From this, they are to decide how they want to solve the problems, and if they want to create any subproblems for one or more of their problems.

The return type of this method is a bit confusing, here is the signature for the method as defined in the base class:

```
virtual vector<ParentSubproblemsPairType> OnAskForSubproblems() = 0
```

The type ParentSubproblemsPairType is a typedef for the following:

```
std::pair<UA2::CPrimitivePlanInstance*, UA2::CDoExpression*>
```

The typedef is for a std::pair of two values, the first value is of type *CPrimitivePlanInstance* (this is an intermediate subclass of CPlanInstance), and defines a primitive-type plan instance. This part of the pair indicates the plan instance that the planner wants to attach subproblems to, and is referred to as the *parent*. The second part is an object of type *CDoExpression*. This object contains in it essentially what would be within a Plan declaration in the UA2 language, and represents the Do Expression of a plan.

Planners have several methods to help them build this pair as described in the following table. Notice that the calls allow one to build the same constructs in the UA2 language via API calls (with the exception of the conditional expression).

Return Type	**Method**
ParentSubproblemsPairType	BuildDoExpression(CPrimitivePlanInstance& parent, CPlanExpression* expression);
CWithExpression*	BuildWithExpression(CPlanExpression* targetExpression, CConstraintDeclaration* constraint);
CConstraintDeclaration*	BuildTimeConstraint(string constraintName, CTime startTimeLowerBound, CTime startTimeUpperBound, CTime endTimeLowerBound, CTime endTimeUpperBound);
CConstraintDeclaration*	BuildPowerConstraint(string constraintName, CPower maxPower, CEnergy maxEnergy);
COperatorExpression*	BuildSerialExpression(CPlanExpression* operand1, CPlanExpression* operand2);
COperatorExpression*	BuildXorExpression(CPlanExpression* operand1, CPlanExpression* operand2);
COperatorExpression*	BuildParallelExpression(CPlanExpression* operand1, CPlanExpression* operand2);
COperatorExpression*	BuildGroupExpression(CPlanExpression* operand1, CPlanExpression* operand2);
CPlanInstanceExpression*	BuildPlanInstanceExpression(CPlanInstance* instance);
CPrimitivePlanInstance*	BuildAcousticPlanInstance(string instanceName, string acousticDeviceName, CTime startTime, CTime endTime, CDuration taskDuration, CDuration minGap, CDuration maxGap);
CPrimitivePlanInstance*	BuildAutopilotPlanInstance(string instanceName, CPosition destination);
CPrimitivePlanInstance*	BuildSearchPlanInstance(string instanceName, string sonarName, CArea area, CLength lanewidth);

CPrimitivePlanInstance*	BuildTransitPlanInstance(string instanceName, CPosition transitPosition);
CPrimitivePlanInstance*	BuildPhoneHomePlanInstance(string instanceName, string modemName, CFrequency phoneHomeRate);
CPrimitivePlanInstance*	BuildUseModemPlanInstance(string instanceName, string modemName, string modemMessage);
CPrimitivePlanInstance*	BuildUseSonarPlanInstance(string instanceName, string sonarName, CFrequency pingRate);

Using these methods, the planner can build the pairs it wants. For example, if a Search planner wanted to build a Transit plan instance called *myTransit* and UseSonar instance called *myUseSonar* for one of its Search plan instances, *someSearchInstance*, and have them run in parallel, it could do as follows:

```
CSearchPlanInstance* someSearchInstance = /*get handle to instance here*/;

ParentSubproblemsPairType myPair = BuildDoExpression(someSearchInstance,
BuildParallelExpression(BuildPlanInstanceExpression(BuildTransitPlanInstance(/*-*/),
BuildPlanInstanceExpression(BuildUseSonarPlanInstance(/*-*/))))
```

Where the arguments to BuildTransitPlanInstance and BuildUseSonarPlanInstance are ignored for example's sake. One familiar with functional-style languages may notice building the pair follows a very functional style in this example. Some may find this hard to follow, so here is the same example again in an imperative-style;

```
CPrimitivePlanInstance* transitInstance = BuildTransitPlanInstance(/* */);

CPrimitivePlanInstance* useSonarInstance = BuildUseSonarPlanInstance(/* */);

CPlanInstanceExpression* transitExpression =
BuildPlanInstanceExpression(transitInstance);

CPlanInstanceExpression* useSonarExpression =
BuildPlanInstanceExpression(useSonarInstance);

COperatorExpression*  myParallelExpression =
BuildParallelExpression(transitExpression, useSonarExpression);

ParentSubproblemPairType myPair = BuildDoExpression(someSearchInstance,
myParallelExpression);
```

Once pairs are built, they should be pushed onto a vector, and returned, concluding the OnAskForSubproblems method.

Method Usage
Planners are not intended to create new subproblems every BuildSchedules() invocation. When the OnAskForSubproblems() method returns a list of problems, the ones created in previous cycle are detached from the plan instance tree, and destroyed. When the method returns an empty list, subproblems attached from previous cycles are maintained. Essentially, a planner developer is meant to only return something for this method when she or he has to.

Infeasibilities and Conflicts
Planners are also meant to tell the UA2 runtime if there are an infeasibilities or conflicts between plan instances at this point, using the two following methods:

void InformKernelOfInfeasibility(vector<Infeasible*AndReasonPairType> infeasibleInstancesWithReason)
void InformKernelOfConflict(vector<Conflicting*InstancesAndReasonPairType> conflictInstancesListWithReason)

The asterisk in Infeasible*AndReasonPairType and Conflicting*InstancesAndReasonPairType are meant to be replaced with the specific type of planner one is dealing with (again, this is for type safety). The types are typedefs specified as follows:

```
typedef pair<C*PlanInstance*, string> Infeasible*InstanceAndReasonPairType;
typedef pair<*PlanInstanceListType, string> Conflicting*InstancesAndReasonPairType;
```

Where the asterisks are to be replaced with the appropriate type name. For infeasibilities, each pair consists of the plan instance that is infeasible, and an in-English string explaining why it's infeasible to be printed out to the console. Likewise for conflicts, a planner indicates a list of the plan instances that caused the conflict, and a reason why so it can be printed to screen. Planners can return a collection of these pairs for multiple errors. Planners should only invoke this method during the OnAskForSubproblems phase. Doing so in the OnFinalizeSchedule phase does not make sense, and will cause the runtime to raise a panic exception.

Planner-Specific State
Plan instances are meant to be immutable objects that cannot be changed directly by a planner, nor are they meant to be subclassed. However, it is natural to want to be able to associate some planner specific data with each problem, for example, some sort of state data that keeps progress of the respective task. Primitive plan instances have a method for doing so called *PlannerSpecificState(),* which is both a getter and setter for the state.

```
UA2::CPlannerSpecificState* PlannerSpecificState();
void PlannerSpecificState(UA2::CPlannerSpecificState* state);
```

The intent is for planner developers to subclass CPlannerSpecificState, and add any information they require within that object. The state will be persistent throughout the lifetime of the plan instance. If the instance is destroyed for some reason by the UA2 runtime, the state object will also have its destructor invoked. Providing a base class (CPlannerSpecificState) was done to provide some type safety, but unfortunately, the getter will only get a handle to the base type. The planner developer has to downcast the object to their concrete type in order to use the information within it.

OnFinalizeSchedule() Method
Once all planners have decided what subproblems they want on their behalf, the UA2 runtime asks planners to then finalize their schedule via the OnFinalizeSchedule() method. This invocation is done in the opposite order of the OnAskForSubproblems() invocation, as we want the lower-level, more specific planners to finalize their schedule first, so that higher level planners in the dependency graph can finalize their own schedules based on the information in those lower level schedules.

Building Schedules
Schedules in the UA2 API are simply objects of type *UA2::CSchedule*. CSchedule's are just a container for a collection of *UA2::CScheduleRecord* objects. A CScheduleRecord object represents a single row entry in a schedule, defined by the tuple <Start Time, End Time, Plan Instance, Command Data>.

To build a schedule, one just has to create an object of type UA2::CSchedule on the heap, passing a std::string to the constructor to indicate the name of the schedule. The name can be used to distinguish schedules when they are returned from the Cschedule call. The method Cschedule::AddRecord is used to add records to the schedule. To build a record, one creates an object of type UA2::CScheduleRecord on the heap, passing to the constructor a start time, end time, the plan instance the command corresponds with, and a std::string representing the command for the respective vehicle subsystem.

Return and Verification
The OnFinalizeSchedule() method expects a handle to a UA2::CSchedule* as the return value. Once the schedule is returned, it is verified to ensure a few things 1) All the plan instances in the records are valid and that planner's, and that planner's only, responsibility, 2) All the time constraints are being followed, and 3) The schedule only contains records for plan instances that are currently running. If any of these 3 checks fail, the UA2 runtime rejects the schedule, and panics. This is indicative that the planner is faulty, and has bugs in it. One of our future goals is to account for faulty planners, and allow user to handle them in a similar way to handling infeasibilities and conflicts.

Changing Lifetime State

Planners are the only entities that can decide if a task is causing problems, is complete, or should go into a running mode. We have already addressed the issue of problems via the InformKernelofConflict and InformKernelOfInfeasibility methods. Methods are also provided in the planner API that allow a planner to transition the lifetime state of plan instances to a “Running” state or to a “Complete” state.

At any point during OnAskForSubproblems or OnFinalizeSchedule, planners should tell the UA2 runtime if they want to switch any instances to a running or complete state, via the methods, *SetToRunning()* and *SetToComplete()*.
An important thing to note is that planners should not assume that once they have invoked this method, the state has been changed. For example, a retracted plan instance cannot go into a complete or running state. The return code of these calls indicate if the change was successful. Another way to look at it is by reinvoking the plan instance accessors mentioned earlier in the section, and seeing which category the instance alls under.

OnHandleConflict() and OnHandleInfeasibility() Methods

Planners also have the ability to handle conflicts and infeasibilities, just as in the UA2 language, but through two type-safe API calls instead:

```
virtual UA2::CConflictResolution OnHandleConflict(UA2::*PlanInstanceListType
conflictingInstances);
virtual UA2::CInfeasibilityResolution OnHandleInfeasibility(UA2::C*PlanInstance&
infeasibleInstance);
```

Again, the asterisk is meant to contain the specific type for the respective planner (for type-safety as well as having the planner developer having to avoid downcasting).

These methods are meant to be overriden by a planner developer. The arguments to each method indicate which instance(s) are causing problems. The return value of these methods indicates a decision about which instances to disable or retract, just like in the language. The constructor for CInfeasibilityResolution takes a simple enumerated value of type UA2::InfeasibilityDecision, which indicates whether to retract, disable, or do nothing about the infeasible instance. The constructor for CConflictResolution takes a list of plan instances, as well as a decision for each one of type UA2::ConflictDecision, which indicates whether to retract, disable or do nothing.

If the method does not return a concrete decision (i.e. a “no action” decision), the conflict/infeasibility remains unresolved, and the runtime invokes the handler at the next closets level.

4. Conclusion

This concludes the chapter on building UA2 components. This chapter only described the key concepts, methods and classes for building components, but the API has much more to offer. This chapter is meant to help one get started, but the API specification in the appendix is meant to give more details about the calls available, and give one a better understanding of the types used. The next chapter describes an example that builds UA2 components, along with some source code snippets. In conjunction with this chapter, and the API specification, at the end of chapter 6 you should feel pretty comfortable building your own UA2 configuration.

Chapter 6
Design Philosophies

Appendix I
Type Constructor Definitions

Appendix II

C++ Class Hierarchy

Appendix III

C++ API Specification (Doxygen)

Appendix IV

The Custom Console Interface