SAR-lib

User Manual

Version 1.1

6/1/03

Table of Content

[Introduction 2](#_Toc29578762)

[Library Structure 3](#_Toc29578763)

[Documentation Standard 4](#_Toc29578764)

[Documentation Tools: 4](#_Toc29578765)

[DIA 4](#_Toc29578766)

[dOxygen 4](#_Toc29578767)

[Source Code Commenting 5](#_Toc29578768)

[File Header Block 5](#_Toc29578769)

[Class Header Block 6](#_Toc29578770)

[Property / Method description 6](#_Toc29578771)

[Other useful dOxygen Commands: 7](#_Toc29578772)

[Maintenance comment 8](#_Toc29578773)

[Coding Standard 8](#_Toc29578774)

[Configuration Management 10](#_Toc29578775)

[Policy 10](#_Toc29578776)

[CVS set up 10](#_Toc29578777)

[Creating a password: 10](#_Toc29578778)

[The first login / check out: 10](#_Toc29578779)

[Using CVS 11](#_Toc29578780)

[Editing the code: 11](#_Toc29578781)

[Updating the repository: 11](#_Toc29578782)

[Updating your local source: 11](#_Toc29578783)

[SORAL Specific Documentation 12](#_Toc29578784)

[Why you may want to create a new allocation object 12](#_Toc29578785)

[How to create a new allocation object 12](#_Toc29578786)

[Friend Function 13](#_Toc29578787)

[Using the containers 13](#_Toc29578788)

# Introduction

This manual is part of the SarBayes project and contains documentation related to sar-lib part of the project. Sar-lib is a collection of libraries for use by search and rescue programs. At present SORAL is the only library being actively worked on, although a number of other libraries are planned.

This manual is aimed at programs working on further development of the library. Documentation of the code is also available in various formats.

The SARBayes project is based at Monash University (Australia), the web site is at:

http://sarbayes.org/index.html

# Library Structure

SORAL – SarBayes Optimal Resource Allocation Library

This library will take information on your resources, areas and initial search probabilities and create allocations of resource to areas. The allocation are based on search theory and a number of allocation types are available (with more to be added in the future).

Effectiveness Library

This library will assist the user in calculating an effectiveness table. At this point it is only an idea and effectiveness must be calculated / guessed manually and entered into SORAL.

Moving Target Library

This library will update the probability of all areas on the map based on their previous values and the assumption that the target is moving (based on missing person data).

- this may not be developed as a library

New Ideas?

Please let us know any further libraries or library functions that could be useful for search and rescue software. If you build any additional libraries, it would be appreciated if you let the SarBayes team know.

Current contact details can be found on the SarBayes website at: http://sarbayes.org/index.html

# Documentation Standard

SAR-lib contains both internal and external documentation. All diagrams are created using DIA or are auto generated via dOxygen. Documentation will always be available in PDF format on request, with source files created in a combination of text, Latex, HTML and MS-Word.

## Documentation Tools:

### DIA

<http://www.lysator.liu.se/~alla/dia/>

“Dia is a gtk+ based diagram creation program released under the GPL license”

### dOxygen

<http://www.stack.nl/~dimitri/doxygen/>

“Doxygen is a documentation system for C++, C, Java, IDL (Corba, Microsoft, and KDE-DCOP flavours) and to some extent PHP and C#.”

The dOxygen program functions similar to JavaDoc creating file and class based documentation. Output from dOxygen is in the following formats: HTML, LaTeX, MAN pages, PDF, RTF and others.

dOxygen is released under GNU General Public License.

Note on settings:

To work correctly with the style used for this project [**JAVADOC\_AUTOBRIEF**](http://www.stack.nl/~dimitri/doxygen/config.html#cfg_javadoc_autobrief) should be set to no (the default).

## Source Code Commenting

The source code is commented in four ways. The first three use dOxygen. Although there are many ways of using dOxygen, the style outlined below is closest to the original (pre dOxygen) commenting style of this project.

More information on dOxygen commenting styles can be found at: <http://www.stack.nl/~dimitri/doxygen/docblocks.html>

### File Header Block

Each file contains a head block with the project name, the file name, a description of the functionality in the file (classes should not be randomly lumped together, but a source code file may have more than one class in it) and finally a version history of the file. The version history contains who changed the file, when, what version it is now, and what was changed.

Format:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SARBayes OPTIMAL RESOURCE ALLOCATION LIBRARY 2001-03 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* \file containr.h

\* \brief containr.h contains AreaAssignment and ResourceAssignment

\*

\* These objects are used to return area and resource information to

\* the user. The information is wrappered in this class so that the

\* user can not do harmful operations on the allocation list.

\*

\* <b><u>Version History</u></b>

\*

\* \verbatim

\*-----+----------+-----+--------------------------------------------

\* Who | When | Ver | What

\*-----+----------+-----+--------------------------------------------

\* ME | 05/12/01 | 1 | Created.

\*-------------------------------------------------------------------

\* GT | 25/02/02 | 2 | Modifications. AreaAssignment now

\* | | | encapsulates a resource number rather than

\* | | | an area number, whilst ResourceAssignment

\* | | | now encapsulates an area number rather than

\* | | | a resource number.

\*-------------------------------------------------------------------

\* ASO | 10/12/02 | 3 | Modified. Removed base class "container"

\* | | | and updated "child" classes as needed.

\*-------------------------------------------------------------------

\* \endverbatim

\*/

### Class Header Block

Each class is preceded with a header block briefly describing it and its purpose. The description should contain sufficient information to decide if new functionality falls within the classes purpose (and should be part of the class) out outside of it. Following the description is the original authors name. Except when a class has been completely redone (i.e. nothing but the name remains) no further version history is contained at the class level.

Format:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// A brief (1 line) description of class magic

/\*\*

\* A more elaborate class description.

\* This description may span multiple lines.

\*

\* Author: The white Rabbit

\*/

class magic

{

NB: note that the first row is not for dOxygen but rather for when the code is read in an editor.

### Property / Method description

This is a short description of the property of method.

Property: Does not contain its type or name, just what it is for. For any public function, this should be filled in even if it is obvious. Private function may or may not be filled in.

Format:

/// Short description

Int magicBeans; /\*\*< Longer additional description \*/

Or:

/// Brief (1 line) description of magic beans

/\*\*

\* Extended description of

\* of magic beans

\*/

Int magicBeans;

Function: Does not need contain it’s return type, name or parameters unless these need explaining. Unless it is a private function, it should always explain what it does even if it is obvious. If appropriate it should explain why the function is there.

Format:

/// Brief (1 line) description of grow()

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Extended description of

\* of the grow() method

\*/

void grow(int amount);

### Other useful dOxygen Commands:

#### Lists

Bullets are created using – for unnumbered lists or -# for numbered lists. The – symbol must in either case be in the same column (character position) for it to be regarded as the same list.

Example:

/\*\*

\* A list:

\* - Bullet Item 1

\* -# Numbered Item 1

\* -# Numbered Item 2 \n

\* More info about Numbered Item 2

\* -# Numbered Item 3

\* - Bullet Item 2

\* -# Numbered Item 1

\* -# Numbered Item 2

\*

\* Some other text.

\*/

#### Bold

<b> This in bold </b>

#### Underline

<u> This in underline </u>

More at: <http://www.stack.nl/~dimitri/doxygen/commands.html>

NOTE: /page /section etc allow for dOxygen to handle regular documentation, this manual should eventually be converted to this format, allowing fast updating and creation of rtf, html and latex copies.

### Maintenance comment

These comments are not for dOxygen and external readers.

They include:

Commented out code 1:

// Programmers-Initials Date Reason – code

Commented out code 2:

// Programmers-Initials Date Reason

/\*

code

\*/

A note on commented out code:

If there is any possibility a change will need to be undone, the existing code should be commented out but not deleted. The same goes for unneeded function. Testing code should always be commented out unless it is in use.

Added code :

// Programmers-Initials Date Reason

code

A note on flow on changes:

Small maintenance changes that results from larger documented changes do not need to be documented themselves. E.g. if the type of a variable is changes, it is commented at the declaration, and not in all the formal parameter lists where it is used.

# Coding Standard

The library code is designed to be readable. The following are some basic rules to assist with this:

Meaningful names should always be used.

Attribute names start with the word “my”.

Attributes should not be public unless there is a specific reason for wanting this.

Each attribute that may be needed for external use should have a get statement, the name of the get function does not include the “my” prefix.

Friend relationship should be limited as much as possible.

Compound property and method names should start with a lowercase letter, each word thereafter should start with a capital letter eg myReallyLongVariableName aReallyLongFunctionName()

Braces are to set out as:

int aFunction(void)

{

// some code

}

Code is to be maintained both error and warning free under both windows and unix, see configuration management for more details.

# Configuration Management

## Policy

Code is in the “clean state” when it compiles without errors or warnings in both windows (Visual c++, version 6) and unix (g++ on bruce, gcc version 3.1.1). Code should be returned to the clean state and checked in each day (when much work is happening) or at least once a week (when less time is being spent on changes).

Class Diagram(s) should be updated and checked in each time the code reaches a clean state. This prevents the work building up and keeps the diagrams recent enough to be useful.

## CVS set up

### Creating a password:

The following is from Peter Moulder ([pmoulder@bowman.csse.monash.edu.au](mailto:pmoulder@bowman.csse.monash.edu.au))

#! /usr/bin/env perl -w

use strict;

# Charles Twardy has requested that you be given access to SARbayes source code

# in the CVS repository on bowman.

# Can you fill in a password in this perl script, run it, and send me the

# output? ~/.cvspass will store your password (in an easily-decrypted form)

# after your initial `cvs login', so it's more important not to use a password

# used anywhere else than to use a memorable password. (E.g. I no longer know

# what my CVS password is.)

# Salt values can be generated with:

# tr -cd A-Za-z0-9./ < /dev/urandom | dd bs=2 count=1 2>/dev/null; echo

my $salt = 'rs'; # 2-char alphanumeric taken from /dev/random.

my $pass = 'open sesame';

print crypt($pass, $salt), "\n";

Once he has received (and added) the encrypted password, you log in and check out the CVS repository as explained below.

### The first login / check out:

export CVSROOT=:pserver: <username> @bowman.csse.monash.edu.au:/home/cvsroot

# or setenv CVSROOT :pserver: <username>@bowman.csse.monash.edu.au:/home/cvsroot

cvs login

# enter the plaintext password you put in the script

cvs checkout SARbayes/src/SORAL

You will now have a directory at ~/SARbayes/src/SORAL

Use:

> Cd /SARbayes/src/SORAL

> ls

To confirm this.

## Using CVS

### Editing the code:

The code in your local directory may now be edited

### Updating the repository:

Go to the directory with your local copy of the code

Type:

> cvs commit

You will enter a vi session (or your default editor). Add a comment about this update below the instructions and save and exit.

You will see the files being updated

Repository is now updated.

### Updating your local source:

Go to the directory with your local copy of the code

Type:

> cvs update

Your local copy will be updated with newer version of files in the repository

# SORAL Specific Documentation

## Why you may want to create a new allocation object

The allocation objects are the core of SORAL, it is in the calcAllocation method that the actual allocations are done. This method should be called from the constructor (but the allocation code should be kept separate in a calcAllocation function for neatness and consistency).

Besides the calcAllocation method, each allocation class is allowed to store it’s allocation in its own fashion. You have complete freedom over the data storage type you use, provided certain functions fro walking through the structure are present. See below for more details.

## How to create a new allocation object

A new allocation object must inherit from allocation and implement the functions below:

Constructor – make sure to call calcAllocation here. Do not code the allocation calculation directly into the constructor.

Destructor - May sure to empy out any data structures you have created and dynamically assigned memory too.

calcAllocation - The function that does the allocation

And the following movement functions for moving through your data structure:

int firstArea(void)

Returns the area number of the first area in this allocation (i.e. first area with something assigned to it)

AreaAssignment\* firstArea(int resource)

Returns the area number of the first area in this allocation using the resource passed in.

ResourceAssignment\* firstRes(int area)

Given an area, it returns the first resource assigned to that area

int nextArea(int area)

Given an area number, returns the next area that has something assignned to it in this set of allocations.

AreaAssignment \* nextArea(int resource, int area)

Given a resource and an area, returns the next area (and time) that this resource was assigned to.

ResourceAssignment \* nextRes(int Area, int Resource)

Given a resource and an area, it returns the next resource assigned to that area (as a ResourceAssignment).

And finally a storage type (by convention called myAssignments) to store your allocations in.

### Friend Function

You need to declare the following classes friends of your allocation class (copy and paste at the end of your class description):

///So that the AreaIterator can access the first an dnext functions.

class AreaIterator ;

/// So that the ResourceIterator can access the first an dnext functions.

class ResourceIterator ;

/// So that the ActiveAreasIterator can access the first an dnext functions.

class ActiveAreasIterator ;

### Using the containers

There are two types of containers:

* ResourceAssignment
* AreaAssignment

The activeAreaIteraor uses a int instead of a container (as it only needs a single value not a pair).

#### To create a ResourceAssignment

eg for the function :

*ResourceAssignment \* nextRes(int Area, int Resource)*

You must get the correct value for your function’s tempResource variable and then return something like this:

**Syntax:**

return new [ResourceAssignment](http://www.csse.monash.edu.au/~andre/sar/internal/html/classResourceAssignment.html)(tempResource, timeAllocated);

Where tempResources is an int referring to a resource number and timeAllocated is a double referring to how long the resource is used in the area that was passed in.

**Example:**

If your allocation storage object is an array2D called myAssignments, (as in Charnes Cooper and userDef) you would use the following code:

return new [ResourceAssignment](http://www.csse.monash.edu.au/~andre/sar/internal/html/classResourceAssignment.html)(tempResource, [myAssignments](http://www.csse.monash.edu.au/~andre/sar/internal/html/classuserDef.html#o0).[value](http://www.csse.monash.edu.au/~andre/sar/internal/html/classarray2D.html#m2)[Area][tempResource]);

#### To create an areaAssignment

eg for the function :

[*AreaAssignment*](http://www.csse.monash.edu.au/~andre/sar/internal/html/classAreaAssignment.html)*\** [*userDef::nextArea*](http://www.csse.monash.edu.au/~andre/sar/internal/html/classuserDef.html#c3)*(int resource, int area)*

You must get the correct value for your function’s tempArea variable and then return something like this:

**Syntax:**

return new [AreaAssignment](http://www.csse.monash.edu.au/~andre/sar/internal/html/classResourceAssignment.html)(tempArea, timeAllocated);

Where tempArea is an int referring to an area number and timeAllocated is a double referring to how long the area has the resource (that was passed in) for.

#### Example:

If your allocation storage object is an array2D called myAssignments, (as in Charnes Cooper and userDef) you would use the following code:

return new [AreaAssignment](http://www.csse.monash.edu.au/~andre/sar/internal/html/classAreaAssignment.html)(tempArea,[myAssignments](http://www.csse.monash.edu.au/~andre/sar/internal/html/classuserDef.html#o0).[value](http://www.csse.monash.edu.au/~andre/sar/internal/html/classarray2D.html#m2)[tempArea][resource]);

#### Note:

A word of caution, read the descriptions of the functions you must implement carefully. The iterators reply on these functions of yours and will not function as expected if your function do not meet the specification.