Mass Debugger
Release Report

Niko Simonson
April 4th, 2012

System Objectives and Scope

The MASS (Multi-Agent Spatial Simulation) Debugger allows a user of the MASS library to track and know the status of distributed array elements in remote computing nodes, through a user interface. The debugger displays the values of each element and shows which elements are in communication with each other. The debugger also allows the user to set break points and define iterations by wrapping MASS functions and being incorporated into the user's code. The user interface also provides controls during program activation for stepping through code and advancing to the next break point.

Background

For users and developers of MASS, who must develop code to handle numerous, distributed computational elements, a debugger with a graphical interface that can display data in an easily accessible manner is a valuable tool for validation and verification. Such a tool has the potential to speed up development of applications that use the MASS library, and aid the MASS project itself by identifying potential issues with in the MASS classes.

Rationale

MASS is meant to solve large, parallelized computational problems. It is a painstaking process to validate the correctness of such algorithms based on results, and to verify that data computation based on numerous, interacting elements is occuring according to design. A graphical debugger collects this diverse computational data and displays it in a form that is easily understandable by a human user.

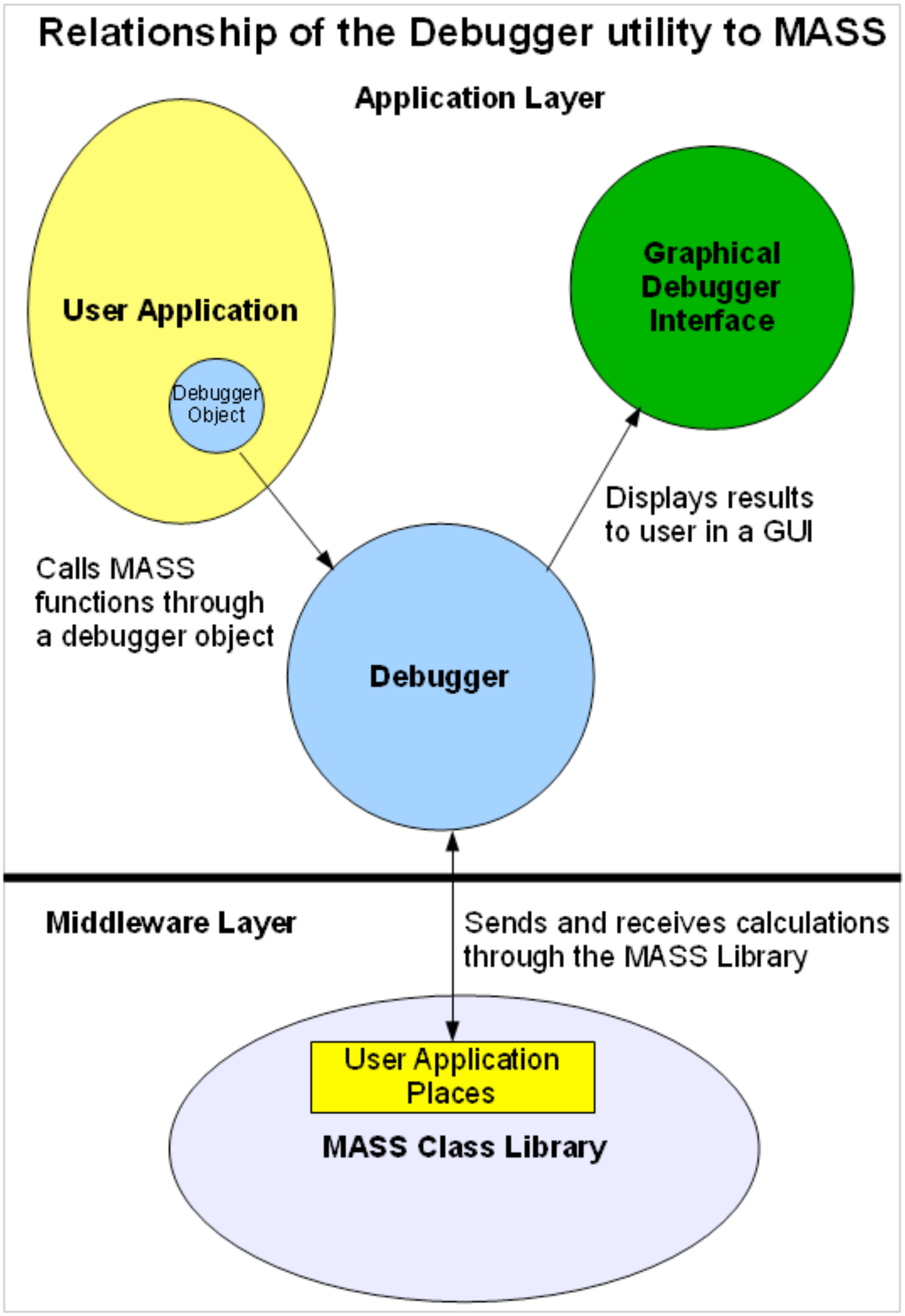
Target Domain

For developers of MASS code, the debugger will provide an additional means to test the library itself.

For users of MASS code, the debugger will allow easy and understandable access to distributed computational elements, aiding in the testing of that code.

The scope of the project is currently limited to the single-process, multi-threaded version of the MASS library, until the unification with the multi-process version is achieved.

Design Strategy


Relationship of the Debugger utility to MASS
Application Layer
User Application
Debugger Object
Graphical Debugger Interface
Displays results to user in a GUI
Calls MASS functions through a debugger object
Debugger
Middleware Layer
Sends and receives calculations through the MASS Library
User Application Places
MASS Class Library

(Figure 1. This diagram shows how the debugger provides a graphical interface and links a user application to the MASS library.)

The development of the debugger relied on an iterative approach that focused

on design, planning, and conceptualization. Features were ranked by priority, and development was driven by features from the user's perspective. The basic goal was to implement the features as simply as possible; just enough to illustrate their utility.

Early in the process it became apparent that there were two challenges: accessing MASS and creating the graphical interface. Comparing the two, using MASS was simpler, but had not yet been thoroughly explored. Initially, Java reflection was used to attempt to read user information from the driver classes. However, this approach did not gain much information due to the way functions are numerically named in a MASS-friendly application. Instead, the MASS class itself was wrapped by the debugger.

The debugger is the first application that divides the utilization of MASS between itself and the driver program, demonstrating that multiple concurrent programs can work together utilizing different aspects of MASS. Though it is focused on showing users MASS data, it is shows how a debugger with a graphical interface can be made very early in the development cycle of MASS: much earlier than equivalent graphical debuggers have occurred in other highly parallelized applications.

Implementation

The debugger program exists separately from the MASS library. Therefore, it is not affected by changes in the MASS implementation that don't affect MASS function signatures. In effect, it serves as a middle layer that wraps MASS functions.

To use the debugger, a prospective MASS user instantiates a debugger object in their code, and use its similarly-named functions in place of the MASS library. The wrapped functions take the same parameters as the MASS functions, and produce the same return values. The debugger provides additional functions mostly related to the setting of breakpoints and defining of execution iterations. The user can define these iterations arbitrarily so that one execution need not correspond to a program loop.All visual controls are supported by public debugger functions that can be called from code, if the developer prefers. The graphics for the debugger rely on Java's awt classes.

System Design
The Debugger is a collection of Java classes that are separate from but include the MASS class library. To use the debugger, a developer who includes the MASS library makes calls to debugger class methods in the code. The debugger itself functions by making calls to the rest of the MASS API.

Execution Model
Instantiating a debugger in code will bring up the debugger GUI when the code is run. The user can interact with the GUI as described in the Use Case scenarios. Breakpoints are set manually in code.

Programming Style
The debugger is written in Java, using object-oriented programming techniques, and adhering to the accepted style of MASS library programming.

Development Resources
The Eclipse software development platform for Java was used to the develop the debugger. The Open Office productivity suite generated the documentation related to the debugger.

Instructions for Use
1.0a Compilation – General
Files required:
- Debugger.java
- MASS-Thread.jar
- a Java application that uses the debugger (examples include Wave2DDriver.java and DebuggerDriver.java)

1.1a Compilation – Unix
Compile in the command line prompt of the local directory:
```
javac -cp MASS-Thread.jar:. Debugger.java
javac -cp MASS-Thread.jar:. Driver.java
```

1.1b Compilation – Windows
Compile in chosen development environment, with MASS-Thread.jar added as an external jar. Otherwise, compile in the command line prompt of the local directory:
```
javac -cp MASS-Thread.jar; Debugger.java
javac -cp MASS-Thread.jar; Debugger.java
```

2.0 Implementation – General

The Debugger does not run on its own. A driver program is required to instantiate a debugger object and call its methods. The driver program must be written in a way that utilizes MASS:

- extends Place
- calls the super class in the constructor
- implements callMethod

The driver then instantiates a Debugger object, and uses Debugger methods.

There are three sorts of debugger methods:

- Methods that wrap MASS functions. These methods have identical function signatures to the MASS functions they wrap.
- Methods specific to the debugger, such as setting breakpoints.
- Hybrid methods that combine MASS functions with debugging functions, such as IterateCallAll, which combines the CallAll MASS function with the Iterate function of the debugger.

The Debugger initializes and finalizes MASS, and runs MASS methods.

3.0a Execution – Unix

Execute from the command line prompt of the local directory:

```
java –cp MASS-Thread.jar:. Debugger
```

3.0b Execution – Windows

Execute from chosen programming environment. Otherwise, execute from the command line prompt of the local directory:

```
java –cp MASS-Thread.jar; Debugger
```

4.0 Assumptions

- It is assumed that the programmer is already familiar with how to utilize functions in the MASS Class library.
- Furthermore, the programmer is going to use the Debugger's graphical interface instead of displaying any graphics for their own application.
- The programmer will be importing the Debugger classes in addition to the MASS class library.

5.0 Implementation Specifics - Incorporating the Debugger Into Code

5.1 Setup

- import MASS
- The programmer's class must extend Place
- The class's constructor must call super();
- The class must be programmed in the form of a MASS-using class with respect to numbered functions
- Instantiate a debugger object
- Call the debugger's debugInit function and pass the following
  - int[] size – the spatial dimensions of the computational elements
  - String appName – the name of the driver application
  - int threads – the number of threads to be used in the MASS simulation
  - double minValue – the anticipated minimum value of the results range
  - double maxValue – the anticipated maximum value of the results range
- Call the debugger's finish() function before the application exits

5.2 What Not to Include

- Direct calls to MASS – The debugger will call MASS functions on the class's behalf.
- Graphics – The debugger's purpose is to display values in a graphical interface. Additional graphics generated by the driver program may produce unintended results.

3.3 Using Graphics

It is important that the debugger's finish() function be called to properly dispose of graphics.

The following MASS functions can be called through the debugger, using the same parameters. This includes all overloaded forms of the function.

- CallAll() is called through debugCallAll()
- CallSome() is called through debugCallSome()
- ExchangeAll() is called through debugExchangeAll()
- ExchangeSome() is called through debugExchangeSome()

3.5 Setting Breakpoints and Iterations

Call the setBreakPoint() function and pass it an integer corresponding to the iteration of the breakpoint.

The programmer sets the maximum number of iterations using setTotalIterations().

Call the iterate() function in order to advance an iteration each time the program performs one set of calculations.

4.0 Running the Debugger

The debugger will display values of the array color-coded from green (low) to red (high).

1. Display Buttons – clicking on any of the displayed values will display its value in a new window. Further values will also be displayed if the application is using more than two dimensions.
2. Next Break – clicking the Next button at the bottom of the display will iterate through to the next break point
3. Continue to End – clicking the Conitnue button at the bottom of the display will remove all further breakpoints and iterate through to the end of the application
4. Step – clicking the Step Through button will iterate one single time.
5. Save – Saves the current state of the computational nodes and exchanges.
6. Restore – Restores the state of the computational nodes and returns them to the driver program.
7. Change Perspective – Changes between a 2D flat perspective and a 3D hawk's-eye perspective.
8. Show Exchange View / Show Value View – Swaps between showing values exchanged by a cell and showing a cell's value.
9. Exiting the debugger – The debugger may be exited by using the X button at the upper righthand corner of the main window.

Function Descriptions

dbgInit

Calls the init function of the MASS class library and starts graphics.

`Parameters: int[] size, String appName, int threads, double minValue, double maxValue`

int[] size – the spatial dimensions of the computational elements
String appName – the name of the driver application
int threads – the number of threads to be used in the MASS simulation
double minValue – the anticipated minimum value of the results range
double maxValue – the anticipated maximum value of the results range

`Returns: void`

dbgFinish

Finishes and disposes of MASS and graphics.

`Parameters: none`
`Returns: void`

Debugger Accessors

isFinished

Returns true if iterations > maximum number of iterations or if manually set to finished.

`Parameters: none`
`Returns: Boolean`

getIteration

Returns the number of the current iteration.

`Parameters: none`
`Returns: int`

getSizes

Returns an array with the simulation's length in each of its dimensions.

`Parameters: none`
`Returns: int[]`

showingExchanges

Returns true if the UI is set to display data exchanges, false if set to display values.

`Parameters: none`
`Returns: Boolean`

Debugger Mutators

dbgIterate

Increments the number of iterations by one. This is only performed if not on a break point or when stepping through.

`Parameters: none`
`Returns: void`

setIsFinished
Manually sets whether the debugger is finished or not.
`Parameters:` Boolean finished
`Returns: void`

setBreakPoint
Set how many iterations to perform before reaching a break point.
`Parameters: int breakPt`
`Returns: void`

setStopOnBreakPoint
Set whether the debugger pauses when reaching a break point.
`Parameters: Boolean stopOnBrk`
`Returns: void`

setTotalIterations
Sets the maximum number of iterations for the debugger.
`Parameters: int iterations`
`Returns: void`

changePerspective
Switches the debugger view between 2D and Hawk's Eye.
`Parameters:` none
`Returns: void`

Wrapped MASS Functions
dbgCallAll
Wraps MASS callAll. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID, Object[] args`
int funcID – the number of the function to call in the driver program
Object[] args – parameter values for the computational nodes
`Returns: Object[]`

dbgCallAll (overloaded)
Wraps MASS callAll. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID, Object argument`
int funcID – the number of the function to call in the driver program
Object argument – single value to be passed to each computational node
`Returns: none`

dbgCallAll (overloaded)
Wraps MASS callAll. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID`
int funcID – the number of the function to call in the driver program
`Returns: none`

dbgExchangeAll
Wraps MASS exchangeAll. This is only performed if not on a breakpoint or when stepping

through. The handle is passed so that the function signature is the same as MASS, but the debugger substitutes its own handle when calling MASS.
`Parameters: int callingHandle, int funcID, Vector<int[]> destinations`
int callingHandle – numerical handle of the calling program
int funcID – the number of the function to call in the driver program
Vector<int[]> destinations – a vector of the relative coordinates of the computational nodes where the value exchanges occur
`Returns: void`

dbgCallSome
Wraps MASS callSome. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID, Object[] args, int... index`
int funcID – the number of the function to call in the driver program
Object[] args – parameter values for the computational nodes
int... index – Specifies which dimensional elements in the computational arrays to call.
`Returns: Object[]`

dbgCallSome (overloaded)
Wraps MASS callSome. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID, Object argument, int... index`
int funcID – the number of the function to call in the driver program
Object argument – single value to be passed to each computational node
int... index – Specifies which dimensional elements in the computational arrays to call.
`Returns: none`

dbgCallSome (overloaded)
Wraps MASS callSome. This is only performed if not on a break point or when stepping through.
`Parameters: int funcID, int... index`
int funcID – the number of the function to call in the driver program
int... index – Specifies which dimensional elements in the computational arrays to call.
`Returns: none`

dbgExchangeSome

Wraps MASS exchangeSome. This is only performed if not on a breakpoint or when stepping through. The handle is passed so that the function signature is the same as MASS, but the debugger substitutes its own handle when calling MASS.

`Parameters: int callingHandle, int funcID, Vector<int[]> destinations, int... index`

int callingHandle – numerical handle of the calling program
int funcID – the number of the function to call in the driver program
Vector<int[]> destinations – a vector of the relative coordinates of the computational nodes where the value exchanges occur
int... index – Specifies which dimensional elements in the computational arrays to call.

`Returns: void`

dbgIterateCallAll

Wraps MASS callAll and also calls the debugger's Iterate() function. This is only performed if not on a break point or when stepping through.

`Parameters: int funcID, Object[] args`

int funcID – the number of the function to call in the driver program
Object[] args – parameter values for the computational nodes

`Returns: Object[]`

dbgIterateExchangeAll

Wraps MASS exchangeAll and also calls the debugger's Iterate() function. This is only performed if not on a breakpoint or when stepping through. The handle is passed so that the function signature is the same as MASS, but the debugger substitutes its own handle when calling MASS.

`Parameters: int callingHandle, int funcID, Vector<int[]> destinations`

int callingHandle – numerical handle of the calling program
int funcID – the number of the function to call in the driver program
Vector<int[]> destinations – a vector of the relative coordinates of the computational nodes where the value exchanges occur

`Returns: void`

dbgIterateCallSome

Wraps MASS callSome and also calls the debugger's Iterate() function. This is only performed if not on a break point or when stepping through.

`Parameters: int funcID, Object[] args, int... index`

int funcID – the number of the function to call in the driver program
Object[] args – parameter values for the computational nodes
int... index – Specifies which dimensional elements in the computational arrays to call.

`Returns: Object[]`

dbgIterateExchangeSome

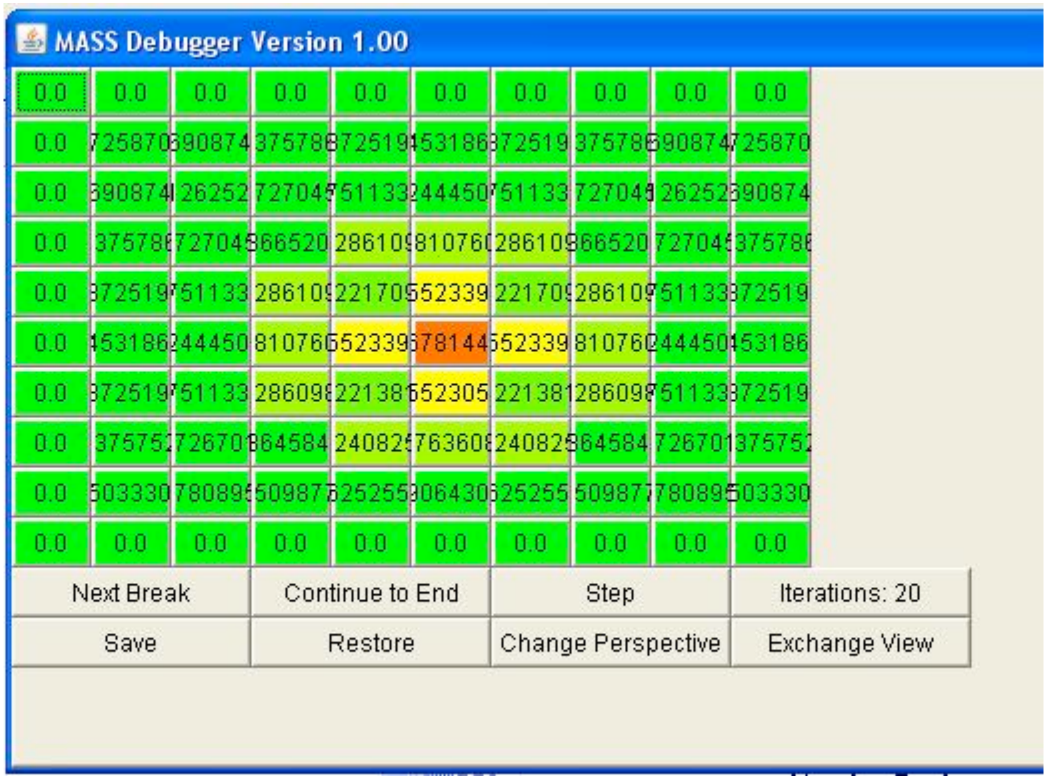
Wraps MASS exchangeSome and also calls the debugger's Iterate() function. This is only performed if not on a breakpoint or when stepping through. The handle is passed so that the function signature is the same as MASS, but the debugger substitutes its own handle when calling MASS.

`Parameters: int callingHandle, int funcID, Vector<int[]> destinations, int... index`

int callingHandle – numerical handle of the calling program
int funcID – the number of the function to call in the driver program
Vector<int[]> destinations – a vector of the relative coordinates of the computational nodes where the value exchanges occur
int... index – Specifies which dimensional elements in the computational arrays to call.

`Returns: void`

Feature Description



MASS Debugger Version 1.00

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	725870	390874	375786	72519	453186	372519	375786	590874	725870
0.0	390874	126252	727045	751133	244450	751133	727045	126252	390874
0.0	375786	727045	366520	286109	810760	286109	366520	727045	375786
0.0	372519	751133	286109	221709	552339	221709	286109	751133	372519
0.0	453186	244450	810766	552339	578144	552339	810760	244450	453186
0.0	372519	751133	286098	221381	552305	221381	286098	751133	372519
0.0	375752	726701	864584	240825	763608	240825	364584	726701	375752
0.0	503330	780895	509877	525255	906430	525255	509877	780895	503330
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

(Figure 2. The flat view of the debugger, showing a small two-dimensional wave simulation.)

The basic function of the debugger is to display the contents of computational nodes in a human-understandable format. The debugger has two types of views: a two-dimensional view and a three dimensional, hawk's eye view. The user can switch between the two types of views during program operation.

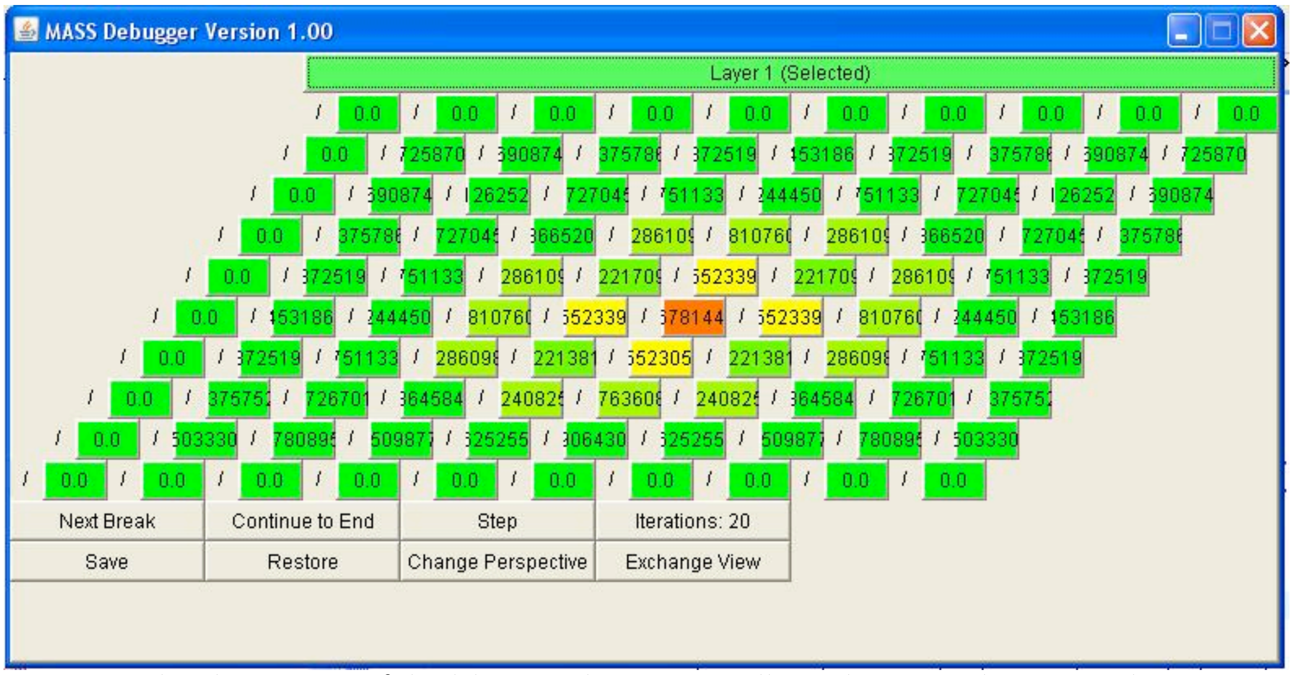
The program can handle multi-dimensionality by showing multiple windows, each containing a two-dimensional view. The top-level window will always display the first value of a sub-array. The user can arbitrarily specify the dimensionality of the debugging display: it can reflect a logical view different from the user's program.
The developer can set break points in program iteration. This is different from traditional debugging code break points, and more akin to setting break points

based on variable values. However, the developer can specify when in the driving code that an iteration occurs. Basic functions based on break points are implemented: the user can step through the program, continue to the next break point, or continue to the end of the program.

The debugger can save its current state to file that serves as a checkpoint. A user can resume operations by feeding the values from the checkpoint file back into the computational places.

Features

Priority 1 – Basic Graphical Debugger



```
MASS Debugger Version 1.00

                                        Layer 1 (Selected)
                         / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0
                       / 0.0 / 725870 / 390874 / 37578 / 372519 / 453186 / 372519 / 37578 / 390874 / 725870
                     / 0.0 / 390874 / 126252 / 72704 / 751133 / 244450 / 751133 / 72704 / 126252 / 390874
                   / 0.0 / 37578 / 72704 / 366520 / 28610 / 81076 / 28610 / 366520 / 72704 / 37578
                 / 0.0 / 372519 / 751133 / 28610 / 22170 / 552339 / 22170 / 28610 / 751133 / 372519
               / 0.0 / 453186 / 244450 / 81076 / 552339 / 578144 / 552339 / 81076 / 244450 / 453186
             / 0.0 / 372519 / 751133 / 28609 / 22138 / 552305 / 22138 / 28609 / 751133 / 372519
           / 0.0 / 37575 / 72670 / 364584 / 24082 / 76360 / 24082 / 364584 / 72670 / 37575
         / 0.0 / 503330 / 78089 / 50987 / 325255 / 306430 / 325255 / 50987 / 78089 / 503330
       / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0 / 0.0
Next Break | Continue to End | Step | Iterations: 20
Save | Restore | Change Perspective | Exchange View
```

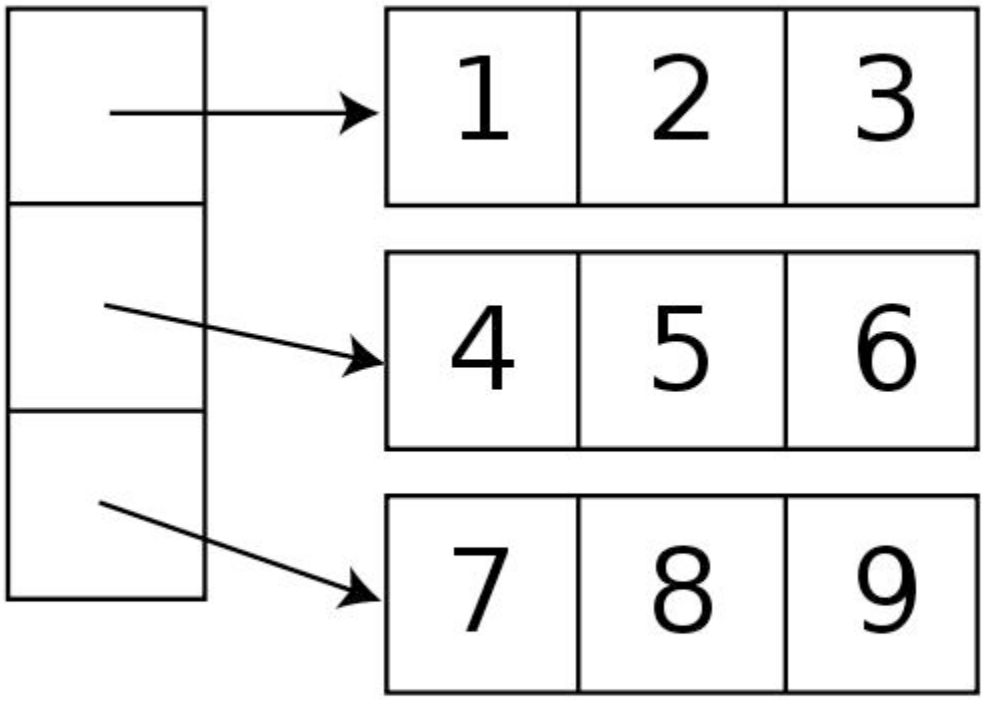
(Figure 3. A hawk's eye view of the debugger, showing a small two-dimensional wave simulation.)

The debugger gathers data from distributed computational elements managed by MASS code and displays such data in graphics objects. Overall display of graphics objects appears as groups of cells.

Individual graphics objects may be directly accessed by mouse click to view their data members, and the data members' contents.

The debugger allows step-by-step execution of MASS code. The debugger also allows the user to set breakpoints in the execution of MASS code.

Priority 2 – Data Exchange View



(Figure 4. Computational elements may be expanded to show their contents. Precise contents not shown. Original image is public domain found on http://commons.wikimedia.org/wiki/File:Array_of_array_storage.svg)

The debugger records which computational elements are exchanging data with other computational elements as well as what data is being exchanged. It displays the above information in graphics objects.

The user is able to access any graphical cell to see an expanded display showing other cells in communication with the selected cell.

Priority 3 Agent View (Future Work)

The debugger shall track the location of MASS agents in distributed computational space. The debugger shall track the data members and their values of MASS agents. It shall display the above information in graphics objects.

The display of agents shall be in the form of square cells representing computational spaces, with an indication of the number of agents inhabiting each space. The user will be able to click on a computational space to see a display of agents, their data members, and those data members' values.

Priority 4 Migration View (Future Work)
The debugger shall track the migration of agents by recording their location over time and their target destinations. It shall display such information in graphics objects.

A user will be able to see agent migration through directional arrows and symbols that represent destinations in the correct cells that represent computational spaces.

Resources Employed
In order to produce this utility, the following resources were utilized:

- Access to the Windows, Linux, and Distributed Systems (DS) computing labs and their contents. Additionally, the DS Lab was used as a meeting location for the MASS team.
- Access to the MASS library documentation and source code.
- Personal desktop PC at home with Windows XP operating system, Firefox web browser, and Open Office productivity suite.
- Network access between personal PC and the University's computing facilities using both the Web and the UWICK SSH Secure File Transfer application.
- MASS team members, advisors, and other stakeholders for advice, assistance, feedback, and evaluation.
- Communication with the Faculty Advisor (Professor Munehiro Fukuda) and other team members and stakeholders using face-to-face meetings, email, and mobile phones.
- Personal vehicle for transportation to and from home, school facilities and MASS meeting sites.
- Parking pass and other parking accomodations for the vehicle.
- Use of standard office-related supplies in the above areas and labs, such as whiteboards.
- Use of standard office supplies such as pens and notebooks.

Results
The project produced deliverables in the form of executable code and documentation, evaluated by the project sponsor, and by peers involved in MASS-related projects.

- Code – the code itself, including comments
- Analysis and Design Documents
- Colloquium Presentation
- Contribution to a formal paper describing MASS utility applications
- Interim and Final Reports

Status
The initial conceptual challenges in the development of the MASS Debugger

were addressed. These were:

- conceptualization – what needs to be written in a debugger
- communication between the debugger, the user's class, and MASS
- development of an interactive graphical interface – how to display graphics, and handle events

The current state of the project:

1. Seamless communication with MASS library.
2. Able to display MASS data with 2D and 3D graphics.
3. Fully functional interactivity.
4. Removal of implemenation of debugging loop: this was replaced by the concept of the user-defined iteration.
5. No means to display exchange of data or agent information. (Priority 3 and 4 items.)
6. Additional feature of saving and restoring data saves the correct data, but does not restore correctly.

Current state of the project:

Complete initial project analysis, identifying risks:

- Lifecycle Analysis Document
- Expanded Use Cases

Development – Code Priority 1 features:

- Incorporate debugger into code.
- Set a break point.
- Choose between 2D or 3D visualization.
- Open and Close GUI
- Step through data.
- Continue to next break point
- Finish execution (continue to the end)
- Display contents of a cell
- Hide contents of a selected cell.

Development - Code Priority 2 features:

- Display exchanges between cells.
- Switch between displaying exchanges and displaying values.

Development - Code additional features, as reflected by the following functional requirements:

- Set user-defined iterations.
- Save/Restore – Restore does not function correctly.

Further Documentation:

- Interim Report
- Collaboration for Paper on MASS utilities
- Colloquium Report
- Final System Design Document

Green highlighted items are complete.
Yellow highlighted items are mostly complete or nearing completion but have remaining issues.
Orange highlighted items are partially complete, but significant conceptual issues remain, requiring further analysis
Red highlighted items were not done this quarter.

Difficulties and Resolutions

The MASS project has two major versions, currently: a single process with multiple threads and a singled threaded multi-proceess. The debugger was developed for the single process, multi-threaded version, but the unification of the two versions was occurring at the same time that the debugger is being worked on. Both versions will be developed in Java, and will have a similarity in terms of classes and methods. The debugger relies on common features of the MASS API, and was developed in consultation with the team member responsible for the unification work.

As an individual project in scope, the lack of experience of the executor in the writing of debuggers and developing of GUIs presented some daunting conceptual challenges. To resolve them, the project will be decomposed into simple requirements that were researched and developed in an iterative process, based on priority and risk. Examples of graphics from previously-developed MASS utilities were used as the framework for the development of the debugger.

Debuggers, graphical objects, interaction with threads, and reflection are all well-known instruments in software development. The unknowns and risk involved how well they would be applied to the MASS library, and how long it would take to do so.
This was thought to be dependent upon retrieving information using the MASS API, but other than the results for the callAll() functions, the majority of information was based on the values passed by the user calling the wrapper functions.

The main challenge during the first half of the project was developing a conceptual approach to the implementation of a debugger. For several weeks, the implementation of what was necessary seemed opaque. This was compounded by a certain amount of confusion over the nature of MASS, itself, and anxiety about the weight of the project.

An incremental approach was adopted that emphasized analysis and design, running parallel with the CSS 370 course in Analysis and Design. This helped in the area of anaylisis, alloying for the identification of major issues early in the project.

Design still proved to be problematic. At first, everything was implemented into a single class. Breaking the design up into multiple classes was necessary to prevent excessive unwieldiness.

The big initial development risks were, in order, the conceptualization of debugger functions, the ability of the debugger to gain information from MASS, and the development of a graphical interface. Conceptualization of debugger functions was solved thanks to the regular analysis of the project. The debugger can gain information from MASS by wrapping MASS commands and acting as a broker between a MASS-friendly application and MASS. Development of a graphical interface for display was aided by the example programs already present, especially Wave2D. The interactive graphical interface was dependent on a great deal of personal research, but did not constitute any ground-breaking activity.

Initially this application was thought to include Java Reflection, but for computational element data, a wrapper proved to be sufficient. However, when it cames time to report on data exchanges, the lack of a return value for ExchangeAll and ExchangeSome meant that exchanges had to reported based on stored CallAll results and the vector passed by the user.

A large breakthrough came when the MASS code was divided between the driver and the debugger. At that point, the debugger truly came into being as an independent class that can be instantiated in a MASS-friendly class.

Future Work

The most immediate challenge to still be resolved is a solution for the inability of the restore function to work properly. After that, minor modifications will be undertaken to improve the appearance of the UI. Internal improvement of the code, in terms of organization and efficiency will continue until public release.

Priority 3 and 4 features are the agent view and agent migration view. They will be left to a project successor. The MASS library does not currently have functions in its API to display internal agent data, so a fundamentally different approach than wrapping MASS functions may have to be undertaken.

Conclusion

Developing the MASS Debugger proved to be a challenging and comprehensive project. It required several disciplines in the areas of both design and development, in particular, risk analysis. Decomposition of the problem proved vital to its eventual resolution.

All in all, the design phase was at least equal to the development phase, and it proved necessary for that to be the case. Many changes in implementation were identified in the design that would have caused endless frustration if they had been allowed to go through to development. The major example of the above is the change from reflection to wrapping.

The current debugger program should prove useful in the analysis of non-agent based MASS utilities. Ongoing development of the agent view of the debugger will probably be a challenge equal in scope to the initial debugger development.