Mission Control System for Project Naiad *

Per-Erik Måhl
pml09001@student.mdh.se

ABSTRACT

This report will cover the mission control system for the Naiad AUV platform developed during the fall of 2013 at Mälardalens University. The Naiad AUV is an autonomous underwater vehicle developed as a research platform for the RALF III research project at Mälardalens University. The purpose of the Naiad AUV is to locate toxic waste in the Baltic sea.

The report explains the design decisions made, the overall system design and the implementation of the mission control system. It is concluded by presenting the state of the system at the time this report was written and what still needs to be implemented.

1. INTRODUCTION

This report explains the development of the mission control system for Naiad AUV. Naiad AUV is an autonomous underwater vehicle (AUV), developed during Project Naiad. Project Naiad was a project run by 18 M.Sc. students, studying robotics and embedded systems, during the fall of 2013. The goal of the project was to develop a research platform for an AUV designed to locate toxic waste in the Baltic sea. Part of this goal was to develop a prototype AUV as a proof of concept. The prototype was named Naiad.

*This report was written during the fall of 2013 in an advanced level project course at Mälardalen University, Sweden.

Mälardalens Högskola
School of Innovation, Design and Engineering
Project in Advanced Embedded Systems DVA425

Mikael Ekström, Evaluator
mikael.ekstrom@mdh.se

Lars Asplund, Customer
lars.asplund@mdh.se

1.1 Requirements

During the design of the mission control system, a set of requirements were established.

- Missions for Naiad should be easy to create, preferably without having to write any code.
- Missions should be executed on the mission control system in concurrence with any other tasks performed by the mission control system.
- Missions should be transmitted to the mission control system through the umbilical chord, which is what the physical cable connection into the AUV is called.
- Missions have to be executed on the AUV whether the umbilical chord is connected or not.
- The mission control system has to communicate with other systems in the AUV.

1.2 Limitations

These limitations were specific requirements requested by the client.

- The mission control system had to be developed in Ada [1].
- The system should operate under the restrictions of the Ravenscar profile [5].

2. METHOD

This section contains the system architecture of the mission control system for Naiad. It also covers the idea behind designing missions for the AUV.

2.1 System architecture

The system needed the ability to simultaneously execute missions, receive new missions through the umbilical chord, and communicate with the other systems of the AUV. In order to do this, several tasks were set up.

The new missions needed to be received using Transmission Control Protocol (TCP), so a task was created to handle this. Missions need to communicate with the rest of the AUV through a Controller Area Network (CAN), so two tasks were created to handle the input and output over CAN. Finally, a main task was created to run a virtual machine

(VM) which would execute the actual missions. The workflow between these tasks can be seen in Figure 1 along with the resources they use. The TCP resource allows the software to use the Ethernet port on the hardware. The CAN resource allows the software to communicate through the CAN bus on the AUV.

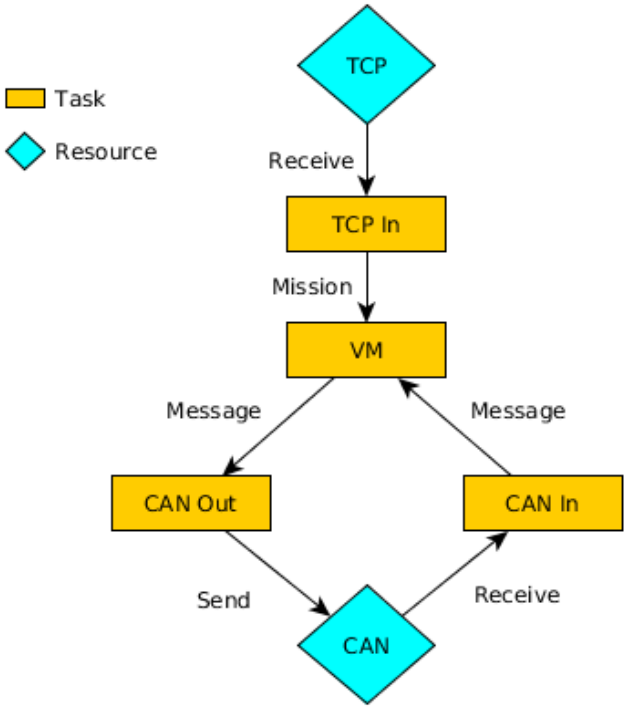

Figure 1: Tasks and resources for the mission control system.

2.1.1 *Virtual machine task*

The task running the virtual machine is responsible for executing the missions. Missions consist of stack machine code, which is interpreted by the virtual machine. This task is constantly looping. Each iteration the virtual machine executes one instruction from the mission.

2.1.2 *TCP In task*

The task handling TCP input is responsible for receiving missions through the TCP resource. The task is listening on a specific port for incoming connections. When a connection is established, the task tries to receive a mission file. When the task has successfully received a mission file, an indication flag is set. The virtual machine task checks for this flag each iteration, and if the flag is set, the new mission file is loaded and executed.

2.1.3 *CAN In task*

The task handling CAN input is responsible for receiving CAN messages from different components in the AUV through the CAN resource. The messages are put in a list which is shared with the virtual machine task, allowing missions to retrieve data from other components of the AUV.

2.1.4 *CAN Out task*

The task handling CAN output is responsible for sending CAN messages to different components in the AUV through the CAN resource. The messages are retrieved from a list which is shared with the virtual machine task, allowing missions to send data to other components of the AUV.

2.2 Mission design

The process of designing missions has been layered in order to keep mission-designing as easy as possible. The user uses an integrated development environment (IDE) to graphically put together missions using different components. The components available are nodes, primitives and objectives.

2.2.1 *Nodes*

Nodes are the most basic components. There are five different nodes.

Start: This denotes the start of execution.

End: This denotes the end of execution.

Constant: This is a constant value. The types of constants are boolean, integer, float, vector, and matrix.

Variable: This is a mutable value. The types are the same as constants.

Branch: This branches the execution. It takes one input and continues on one execution path if the input value is true, and another execution path if the value is false.

2.2.2 *Primitives*

Primitives are written in NaiAda source code (more on NaiAda in Section 3.4). Primitives have inputs and outputs. The inputs provide the primitives with the data needed to execute correctly. Any results that need to be made available outside the primitive after execution are set as output values. In the IDE, a primitive's inputs can be assigned with constants, variables and other primitives' outputs. A primitive's outputs can be used to assign variables and other primitives' inputs.

2.2.3 *Objectives*

Objectives are created using the IDE by stringing together different mission components, including other objectives. An objective has neither inputs nor outputs and can therefore neither accept data, nor pass it on. Technically there is no difference between an objective and a mission, since objectives can contain other objectives. An objective can be said to be a mission when it is intended to be compiled and run on the mission control system.

3. IMPLEMENTATION

This section covers the implementation of all the components making up the mission control system for Naiad.

3.1 Hardware

It was established early on that the system was going to be run on a BeagleBone Black board [3]. This allowed the mission control system to be developed using a less stripped down version of Ada, which can facilitate the development.

Limited versions of Ada, for instance the version used on the AT90CAN128 [2] cards, don't support object oriented language features.

3.2 Virtual machine

The virtual machine was implemented to interpret stack machine code. Instructions are read and executed from a binary object file, which makes the interpreter perform operations on a memory stack.

3.3 Toolchain for creating missions

The stack machine code may be difficult to read and understand, so the system was designed to have different levels of source code. On the lowest level there is stack machine code. One level up is a language similar to Ada, which has been named NaiAda (see Section 3.4). The highest level is Extensible Markup Language (XML), which is used to aggregate different NaiAda source code files to create an entire mission which can then be turned into stack machine code. An overview of the toolchain can be seen in Figure 2.

.

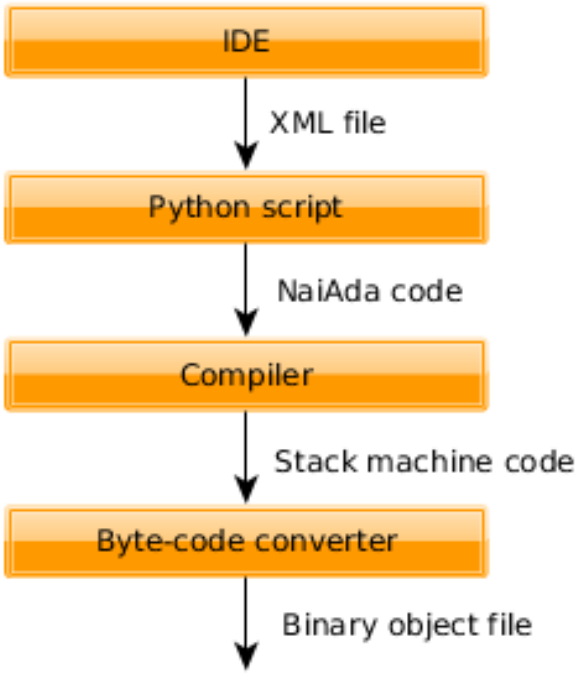

Figure 2: The toolchain for creating missions. The arrow captions denodes the output from each step.

3.3.1 *Integrated development environment*

The integrated development environment is a graphical application written in Java. It is essentially a drag-and-drop graphical programming application, which allows entire missions to be programmed by stringing together different components. The output from the IDE is an XML file, containing all the components of the mission and directions where the source code for each component can be found.

3.3.2 *Aggregator*

The next part in the toolchain is a python script which aggregates the source code for each component of the mission, and puts everything in one file containing only NaiAda source code. The aggregation script also inserts a collection of basic NaiAda functions from a single file, which can be used by any primitive. These functions have collectively been called the standard library. This library can be expanded at any time by adding more functions to the same file.

3.3.3 *Compiler*

The compiler's purpose is to translate NaiAda source code into stack machine code. It has been developed in C, using the tools Flex [6] and Bison [4]. Flex is a tool for creating a lexical analyser to use as a scanner for a compiler. Bison is a tool for creating compilers, which can use scanners created with Flex. The compiler has been designed to report errors as precisely as possible. In order to do this, the compiler has different passes.

Syntax analysis: This pass checks to see that the syntax is correct according to NaiAda's grammar, which is presented in Appendix A.

Name analysis: This pass checks to see that all the names for variables, functions and labels are not defined more than once nor undefined.

Type analysis: This pass checks to see that the types match up for all the functions, variables and constants.

If any of these passes produces an error, the kind of error will be described in an accurate way pointing to a specific line in the source code. The output from the compiler is a textfile with stack machine code.

3.3.4 *Byte-code converter*

The final step is the byte-code converter, which is used to convert the stack machine code to a binary object file. The motivation for this conversion is the decrease in filesize, and the fact that it is easier for the virtual machine to read binary structures, instead of parsing string data.

3.4 NaiAda

NaiAda is the language created to write primitives (see Section 2.2.2) in. It is designed to be as close to Ada as possible, and has been developed to have the ability to realize any task needed by the AUV.

3.4.1 *Types*

Boolean: Has two values, true or false.

Integer: Represents integers.

Float: Represents decimal numbers.

Vector: Represents vectors in 3D space.

Matrix: Represents 3x3 matrices in 3D space.

3.4.2 Access types
Any variable can be pointed to using access types. This allows functions to change the values of variables outside scope.

3.4.3 Functions and procedures
Functions and procedures work in a way similar Ada. The difference is that functions can return a value whereas procedures can not.

3.4.4 Labels
Labels were included in the language to enable the IDE to branch the execution path. A branch in the IDE makes the aggregator produce goto-statements in the NaiAda code.

3.4.5 Typecasting
Typecasting exists in the language to convert from float to integer and vice versa.

3.4.6 Basic statements
The language features a couple of basic statements which basically works the same way as they do in Ada.

if-then-else: Allows the execution path to branch.

while-loop: Allows the execution to loop as long as a specified condition is true.

3.4.7 Basic math functionality
Some basic math functionality has been implemented in the language. The math functions that are native in NaiAda are:

sin: Calculates the sine value for an angle.

cos: Calculates the cosine value for an angle.

arcsin: Calculates the angle from a sine value.

arccos: Calculates the angle from a cosine value.

abs: Returns the absolute value for a value.

sqrt: Calculates the square root of a value.

4. RESULT
This section presents the results of the project regarding the mission control system, and the future work that needs to be done.

4.1 Ravenscar profile restrictions
The system is implemented to compile with Ravenscar profile restrictions [5] and successfully does so.

4.2 Receiving missions
The system's ability to receive missions through TCP and begin execution is finished and has been successfully tested.

4.3 CAN communication
The system currently lacks the functionality to interpret CAN messages sent from other systems in the AUV. The system is implemented to the point where it can receive and send CAN messages, however there is no functionality for handling the incoming messages.

4.4 Virtual machine
The virtual machine is implemented to the point where it interprets and executes any stack machine code compiled from NaiAda source code. However, it lacks functionality for communicating with the other systems in the AUV. Stack machine code instructions that are to be used for communication need to be implemented.

4.5 Integrated development environment
The IDE is still under development. The IDE also needs primitives in order to be usable, which need to be written in NaiAda.

4.6 Aggregator
The python script which uses XML files created by the IDE to aggregate NaiAda source code files into one file is finished and tested.

4.7 Compiler
The compiler is finished and tested. It compiles correctly written NaiAda code produced by the aggregator into stack machine code and reports naming errors, type errors and syntax errors.

4.8 Byte-code converter
The byte-code converter is finished and tested. It's a small program implemented in Ada, which takes a file containing stack machine code in text, and produces a binary object file which can be run on the VM as a mission.

5. CONCLUSION
Work on the mission control system development has been completed to a satisfactory level. The system is not yet finished, but a good foundation to build upon has been made.

6. REFERENCES

[1] Ada reference manual. `http://www.ada-auth.org/standards/12rm/html/RM-TTL.html`. Accessed 2014-01-14.

[2] At90can32/64/128 manual. `http://www.atmel.com/Images/doc7679.pdf`. Accessed 2014-01-14.

[3] Beaglebone black system reference manual. `https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true`. Accessed 2014-01-14.

[4] Bison manual. `http://www.gnu.org/software/bison/manual/bison.html`. Accessed 2014-01-14.

[5] A. Burns, Alan Burns. The ravenscar profile.

[6] Flex website. `http://flex.sourceforge.net`. Accessed 2014-01-14.

APPENDIX

A. NAIADA LANGUAGE GRAMMAR SPECIFICATION

```
program     → comp_units

comp_units  → comp_units function
            | comp_units primitive
            | function
            | primitive

functions   → functions function
            | function

primitive   → 'primitive' ID 'is'
    prim_decls functions 'end' ID ';'

function    → 'procedure' ID '('
    formals ')' 'is' decls 'begin'
    stmnts 'end' ID ';'
            | 'procedure' ID 'is'
                decls 'begin' stmnts '
                end' ID ';'
            | 'function' ID '('
                formals ')' 'return'
                BASIC_TYPE 'is' decls
                'begin' stmnts 'end'
                ID ';'
            | 'function' ID 'return'
                BASIC_TYPE 'is' decls
                'begin' stmnts 'end'
                ID ';'

prim_decls  → prim_decls prim_decl
            | prim_decl

prim_decl   → ID ':' 'in' BASIC_TYPE
    ';'
            | ID ':' 'out' BASIC_TYPE
                ';'

formals     → formals ';' formal
            | formal

formal      → ID ':' type

decls       → decls decl
            | ϵ
decl        → ID ':' type ';'

type        → BASIC_TYPE
            | 'access' '(' BASIC_TYPE
                ')'

stmnts      → stmnts stmnt
            | stmnt

stmnt       → ID ':=' expr ';'
            | 'asm' '(' STRING_CONST '
                )' ';'
            | 'if' expr 'then' stmnts
                'end' 'if' ';'
            | 'if' expr 'then' stmnts
                'else' stmnts 'end' '
                if' ';'
            | 'while' expr 'loop'
                stmnts 'end' 'loop' ';
                '
            | 'return' expr ';'
            | 'return' ';'
            | ID '(' actuals ')' ';'
            | 'loop' stmnts 'end' '
                loop' ';'
            | 'exit' ';'
            | 'null' ';'
            | '<<' ID '>>'
            | 'goto' ID ';'

expr        → '-' expr
            | 'not' expr
            | unaryop '(' expr ')'
            | expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr
            | expr 'and' expr
            | expr 'or' expr
            | expr '=' expr
            | expr '/=' expr
            | expr '<' expr
            | expr '<=' expr
            | expr '>' expr
            | expr '>=' expr
            |'(' expr ')'
            | ID '(' actuals ')'
            | ID
            | 'access' '(' ID ')'
            | INT_CONST
            | 'true'
            | 'false'
            | FLOAT_CONST
            | '[' expr ',' expr ','
                expr ']'
            | '[' '[' expr ',' expr ',
                ' expr ']' ','
                    '[' expr ',' expr ',
                        ' expr ']' ','
                    '[' expr ',' expr ',
                        ' expr ']' ']'
            | VEC_COMP
            | MAT_COMP

unaryop     → BASIC_TYPE
            | MATH_TYPE

MATH_TYPE   → 'sin'
            | 'cos'
            | 'arcsin'
            | 'arccos'
            | 'abs'
            | 'sqrt'

BASIC_TYPE  → 'integer'
            | 'boolean'
            | 'float'
```

```
                | 'vector'
                | 'matrix'

actuals         → exprs
                | ϵ

exprs           → exprs ',' expr
                | expr

ID              → regexp{[A–Za–z_](([A–Za–
    z_]|[[0−9]])*}

VEC_COMP        → regexp{[A–Za–z_](([A–Za–
    z_]|[[0−9]])*\.[XYZ]}

MAT_COMP        → regexp{[A–Za–z_](([A–Za–
    z_]|[[0−9]])*\.[XYZ]Vector}

STRING_CONST→ regexp{"\""([^\"])
    *"\""}

INT_CONST       → regexp{[0−9]+}

FLOAT_CONST  → regexp{[0−9]+\.[0−9]+}
```