# Implementing Arrays and Macros for TethysL for a More User-Friendly Mission Scripting Language

**Riley Rettig, Wellesley College**

***Mentors: Carlos Rueda, Brett Hobson***

***Summer 2019***

**Keywords: Domain-Specific Language, Autonomous Underwater Vehicle, Tethys, TethysL, Mission Scripting**

## ABSTRACT

TethysL is a Domain-Specific Language being developed at MBARI to promote readability of mission scripts for the Tethys family of Long-Range Autonomous Underwater Vehicles (LRAUV). TethysL is intended to be more user-friendly than XML, which is the core language used by the overall LRAUV mission execution framework. Along with a web-based editor, the TethysL system offers operators an integrated environment for authoring LRAUV mission scripts even with little programming experience. In order to continue the goal of improving readability and reducing repetition, arrays and macros were decided to be helpful TethysL language extensions for mission script authors. We describe the implementation and an evaluation of these extensions. These additions not only have significantly decreased the length and complexity of a number of LRAUV mission scripts, but also, based on user feedback, have been shown to make the authoring process more user friendly and less error-prone.

## INTRODUCTION

A Domain-Specific Language (DSL) is a language specialized to a particular domain (Fowler and Parsons, 2010). DSLs can be helpful for simplifying the definition of complex tasks or making programs more readable and user-friendly. Mission scripting of Long-Range Autonomous Underwater Vehicles (LRAUV) is an example of a concrete domain that can be benefited by a DSL approach. Although XML has traditionally been the general mechanism to define missions in the LRAUV framework, a new DSL, named TethysL, is being developed at MBARI as an alternative to XML. The central motivation for this effort has been to offer mission script writers and maintainers a friendlier and less error-prone language. XML is a very powerful language for machine-based processing, however, it is far from ideal at the user level in terms of readability and authoring especially for users with little or no experience with XML.

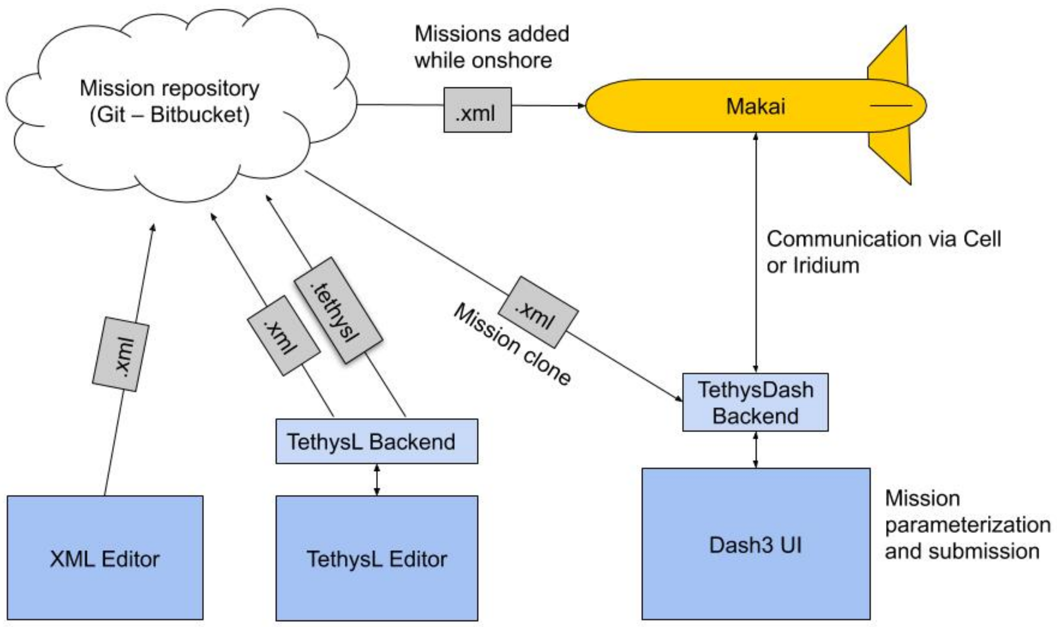


Figure 1. The workflow to how mission scripts are created and stored, then deployed on the LRAUVs ("Makai" used as an example), and accessed by the TethysDash coordination system for operators to parameterize and submit during vehicle deployments.

TethysL has significantly evolved since the project started in 2016, but it is still considered under development. In this project, we focused on extending TethysL to include features that, even with no direct correspondence in XML, would reduce boilerplate code and ultimately create shorter, more readable mission scripts. The decision of what language extensions we should implement was based on what was possible within the context of the underlying framework and what would help make TethysL a successful DSL.

The overall flow for how mission scripts are deployed on the vehicle is shown in Figure 1. Essentially, a mission script is a parameterized template for what the vehicle is going to do. Specific parameter values can be changed later, but these scripts will tell the vehicle what to do during a deployment. These scripts are traditionally written in XML, using an XML or regular text editor according to user preferences, or generated through the TethysL system (that is, initially written in the TethysL language). Regardless of the source, the mission scripts in XML are maintained in a Bitbucket repository. From here, all the XML files are downloaded to the vehicle before deployment. When the vehicle is in the water, operators can choose which mission is run and set the parameters for the mission script through the Dash3 User Interface[1]. If the vehicle is nearby, cell service can be used to communicate to it, otherwise Iridium allows communication via satellite.

As already said, if using the TethysL editor, the TethysL file is translated to XML, but nothing more (other than also storing it at Bitbucket for reference convenience) is currently done with the TethysL file. This scheme mainly responds to a need to use the existing LRAUV framework without changes. Even though we implemented arrays and macros as the TethysL level, there is no direct concept of these extensions in the LRAUV XML Schemas and, consequently, they are not reflected, for example, in the Dash3 interface. The features we chose must be able to be translated into an XML file that follows the schemas in order to be a valid mission.

The LRAUV mission execution framework follows the State Configured Layered Control approach described in Godin et al. 2010. The LRAUV XML Schemas determine the formal structure of valid mission scripts, which can be very simple or complex depending on the needed logic at the mission level. Figure 2 shows a few of the main

[1] https://okeanids.mbari.org/dash3/

elements of the LRAUV mission model originally described by Godin et al. 2010 that are relevant to the implementation of arrays and macros. The *mission* element defines a mission, whose body usually begins with some *DefineArg* elements ("arguments" in TethysL). Here, variables can be declared with default values that can be changed later when the mission is parametrized in Dash3. An *aggregate* is typically a group of *behaviors* that the vehicle can run. These behaviors can be run in different ways, such as in sequence or in parallel to other behaviors. The behaviors themselves are what makes the XML schemas a useful scripting language in that they don't have to be defined in the mission script, but are actually like something you can pull from a library of common behaviors that allows you to control the vehicle. The behavior you will see as an example in this paper is Guidance:Waypoint, which allows the vehicle to move to different coordinates using longitude and latitude arguments. TethysL follows this framework but eliminates boilerplate code to make it more readable. Our features must also follow this framework.

| ***Element*** |
|---|
| *Mission* |
| *DefineArg* |
| *Aggregate* |
| *Behavior* |

Figure 2. A few important elements in the LRAUV framework, adapted from Godin et al. 2010.

In their paper on Domain-Specific Languages in Practice, Hermans F. et al. identify six factors which lead to a successful DSL and conduct a survey to measure the success of their own DSL, ACA.NET. These factors are learnability, usability, expressiveness, reusability, development costs, and reliability. These could be used to measure the success of TethysL features and, while TethysL is in development, answer the question of whether it will be a successful DSL.

An initial prototype of the TethysL language was developed by Eli Meckler as an MBARI Summer Internship project in 2016. The translation from TethysL to XML is made up of parsing (including Abstract Syntax Tree generation), semantic validation, and translation (Meckler, 2016). The syntax and parsing of TethysL is done using the FastParse library (Haoyi, 2019). During parsing, an Abstract Syntax Tree (AST) is created, which captures each syntactic structure as a node in memory. In AST validation, the AST is traversed to ensure that the script makes semantic sense; this includes type checking, resolving names, and more. Finally, once it is ensured that the mission script is valid TethysL, it is translated to an equivalent XML file which can be uploaded to the Bitbucket repository and deployed on the vehicle. After deciding on arrays and macros through user feedback and discussions, these features were implemented following this same workflow.

## MATERIALS AND METHODS

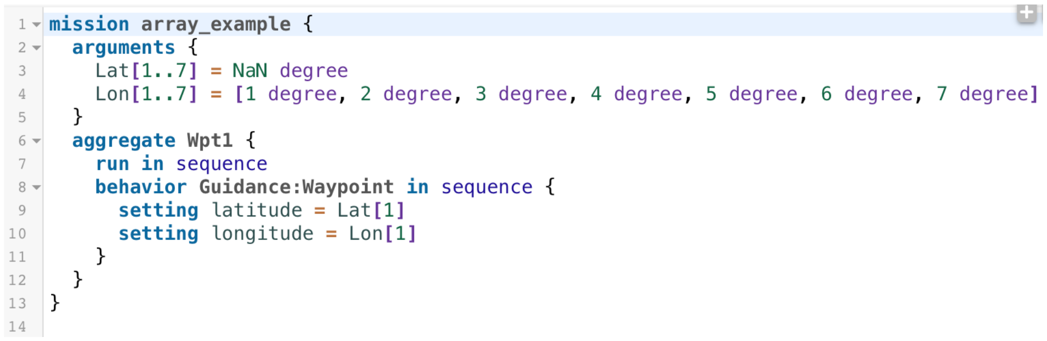
TethysL intends to offer a simpler, more readable syntax compared to XML. However, at the semantic level, this goal is ultimately restricted by the existing LRAUV XML schemas. Although these schemas allow for complex mission scripts, it, at the same time, limits what types of language extensions can be added to TethysL. In particular, when considering what general purpose programming languages typically provide (for example, in terms of control structures), we cannot simply pretend to be able to add any such features in TethysL without taking into account the actual capabilities supported by the XML schemas in the LRAUV execution framework. Some desirable features can be incorporated only at the TethysL level, while others would also require modifications in the LRAUV framework itself, both to the XML schemas and the corresponding support in the execution logic.

While choosing the most useful extensions for TethysL, discussions with the mission script authors were critical. Features that theoretically could be eloquent would not be useful if they were superfluous to the mission script authors, in which case, they would only make the language needlessly more complex, going against the original goal of TethysL. Arrays could be useful because a number of missions involve cycling through waypoints made up of latitudes and longitudes. All of the latitudes, for example, could be stored in one variable, rather than multiple variables for each waypoint. This prevents many variables with similar names (latitude1, latitude2, etc.). Additionally, in talking with mission scripts authors, it was realized that arrays could be even more helpful. One author in particular has run missions that would collect many scientific ESP[2] samples and therefore had up to 60 variables that had the same name root. This mission, named isotherm_depth_sampling, has 7 sets of information that are required for each of these samples. That means there are 420 variables created which could be reduced to 7 arrays.

[2] The Environmental Sample Processor - https://www.mbari.org/technology/emerging-current-tools/instruments/environmental-sample-processor-esp/

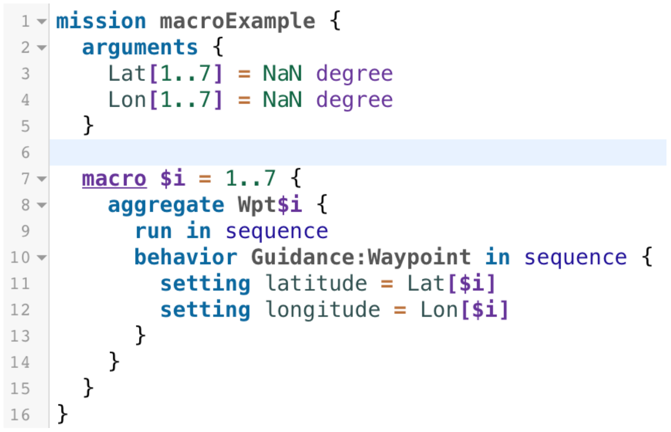
During user testing for arrays, we realized that macros would be a natural extension for the project. In this same mission, there was an aggregate for every sample. These aggregates were identical other than the variables or array access indices that varied with each sample, so the user had to copy the first aggregate and paste it 59 times, changing the variable names or array access indices manually each time. If we just created a special variable that kept track of the current sample number, we could access the array at the correct index for each sample and reduce the number of aggregates per sample to a single generalized one. This would save rewriting 59 aggregates and many lines of code.

## DESIGN



```
mission array_example {
  arguments {
    Lat[1..7] = NaN degree
    Lon[1..7] = [1 degree, 2 degree, 3 degree, 4 degree, 5 degree, 6 degree, 7 degree]
  }
  aggregate Wpt1 {
    run in sequence
    behavior Guidance:Waypoint in sequence {
      setting latitude = Lat[1]
      setting longitude = Lon[1]
    }
  }
}
```

Figure 3. An example of a simple mission using arrays shows the two types of array declarations as well as array accesses.

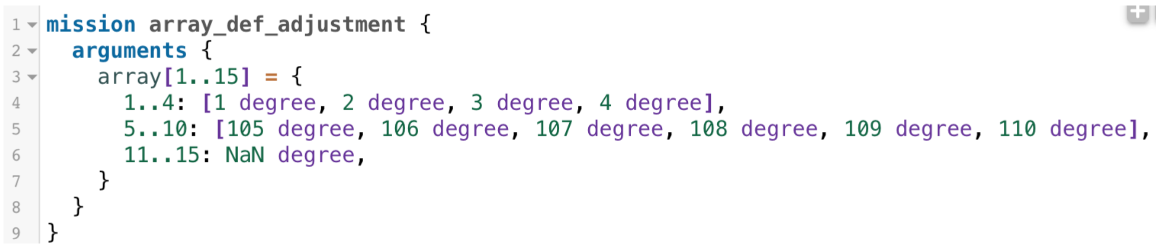


```
mission macroExample {
  arguments {
    Lat[1..7] = NaN degree
    Lon[1..7] = NaN degree
  }

  macro $i = 1..7 {
    aggregate Wpt$i {
      run in sequence
      behavior Guidance:Waypoint in sequence {
        setting latitude = Lat[$i]
        setting longitude = Lon[$i]
      }
    }
  }
}
```

Figure 4. An example of a simple mission using arrays and macros shows the syntax for a macro header and block.

Experienced programmers and mission authors can learn essentially any syntax for the arrays and macros; however, we wanted to create something that would be very familiar and easy to learn for new operators. TethysL is meant to be readable so that anybody with a little programming experience could look at the mission scripts and get an idea of what they do. For this reason, we chose the syntaxes shown in Figure 3 and 4 for arrays and macros respectively.

For arrays, we decided to allow the user to pick their own range to allow flexibility according to what the user prefers. For example, programmers tend to zero-index their arrays, but scientists start their samples at 1 and may prefer that the arrays reflect this. We also limited the size of arrays to 100 elements, to prevent the accidental declaration of large amounts of variables. Arrays can be declared in two ways. The author can use a simple expression on the right-hand side as shown in line 3. This creates an array where every element is set to that default value. Alternatively, the author can write out explicitly what each value in the array is, as shown in line 4. Array accesses are designed as expected, simply with brackets and the index of the element desired. My mentor, Carlos Rueda, also added the option of declaring sub-ranges, as shown in Figure 5. This came after a discussion that an operator may want to specify the first few values for testing purposes but have the rest of the values be a default value. The elements can then be accessed at any index of those ranges.



```
mission array_def_adjustment {
  arguments {
    array[1..15] = {
      1..4: [1 degree, 2 degree, 3 degree, 4 degree],
      5..10: [105 degree, 106 degree, 107 degree, 108 degree, 109 degree, 110 degree],
      11..15: NaN degree,
    }
  }
}
```

Figure 5. An adjustment was made to array declarations to allow for declaring subranges of an array.

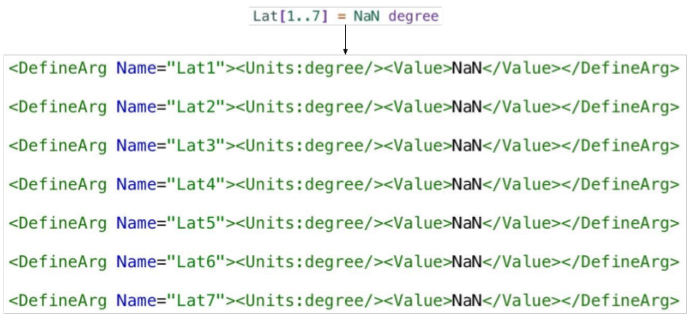
In designing the macros, we wanted something that was fairly simple, so that those who don't know what macros are could easily learn to read and write them. Aggregates are the most common element of mission scripts which would benefit from the macro addition because there is often an aggregate for each element in an array. For this reason, we decided to implement macros such that the body of the macro block is only ever an aggregate. The header itself declares the macro variable, whose name must be prefixed with a "$" to highlight its special meaning, as well as the range of numbers which will be substituted into each aggregate. In the translation to XML, the aggregate will then be repeated through that range, substituting in each number within that range where the macro variable is found.

## IMPLEMENTATION

Implementing the arrays and macros applies the same workflow followed by Meckler when TethysL was first created, that is, parsing, AST validation, and translation. The new consideration in our case, however, is that during translation there is no direct equivalent of the array and macro constructs in the XML schemas; instead, as already explained, we expand these constructs into variables and aggregates in XML accordingly. As part of this workflow, we also included helpful and relevant error messages as appropriate.

### Arrays

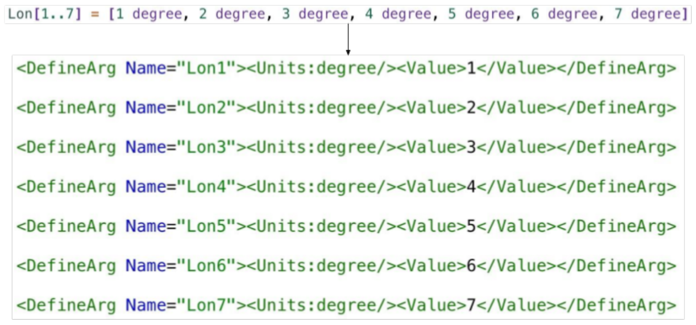
The parsing had to be expanded for arrays to recognize declarations ("Lat[1..2]"), add array declaration expressions ("[1 degree, 2 degree]") as a valid expression, and allow for array accesses ("Lat[1]"). This was done using FastParse (Haoyi, 2019) and converted into an Abstract Syntax Tree. Semantic checks that have to be done included units, size, and bounds checks. All elements of an array must have the same units, arrays cannot be assigned to non-array type variable, arrays must not be greater than 100 elements large[3],



```
Lat[1..7] = NaN degree
```

```
<DefineArg Name="Lat1"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat2"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat3"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat4"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat5"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat6"><Units:degree/><Value>NaN</Value></DefineArg>
<DefineArg Name="Lat7"><Units:degree/><Value>NaN</Value></DefineArg>
```

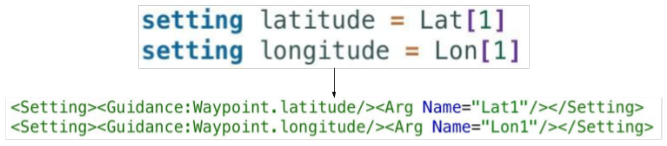
Figure 6a. An array declaration using a default value is expanded to XML during translation.



```
Lon[1..7] = [1 degree, 2 degree, 3 degree, 4 degree, 5 degree, 6 degree, 7 degree]
```

```
<DefineArg Name="Lon1"><Units:degree/><Value>1</Value></DefineArg>
<DefineArg Name="Lon2"><Units:degree/><Value>2</Value></DefineArg>
<DefineArg Name="Lon3"><Units:degree/><Value>3</Value></DefineArg>
<DefineArg Name="Lon4"><Units:degree/><Value>4</Value></DefineArg>
<DefineArg Name="Lon5"><Units:degree/><Value>5</Value></DefineArg>
<DefineArg Name="Lon6"><Units:degree/><Value>6</Value></DefineArg>
<DefineArg Name="Lon7"><Units:degree/><Value>7</Value></DefineArg>
```

Figure 6b. An array declaration using explicit values is expanded to XML during translation.



```
setting latitude = Lat[1]
setting longitude = Lon[1]
```

```
<Setting><Guidance:Waypoint.latitude/><Arg Name="Lat1"/></Setting>
<Setting><Guidance:Waypoint.longitude/><Arg Name="Lon1"/></Setting>
```

Figure 6c. An array access is translated to XML.

[3] Limit determined from user feedback.

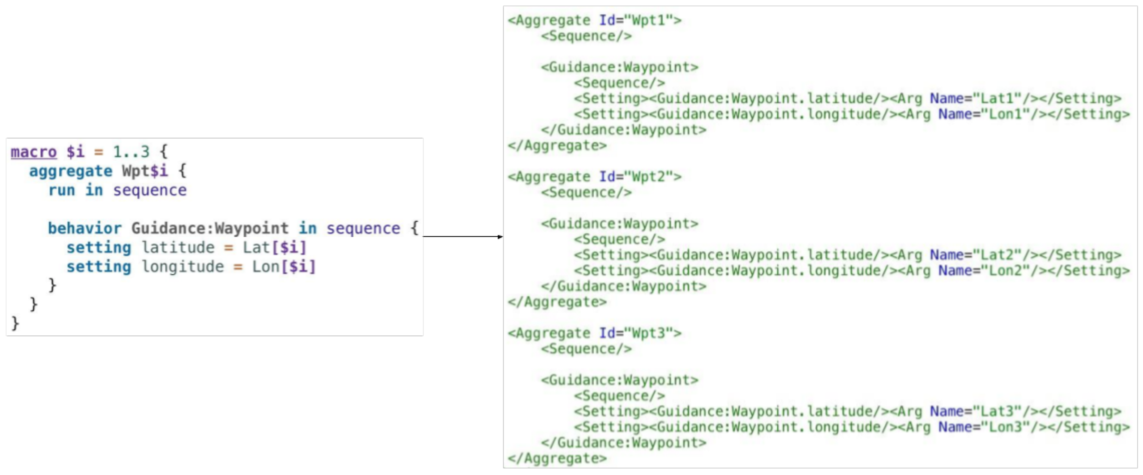
the lower bound must be less that the upper bound during array declarations, and array accesses must be within the bounds of the arrays. These are done by keeping the info for an array's bounds attached to the symbol table entry which keeps track of variable names and types. If any of these rules are broken, an appropriate error is reported to the user after compilation failure, giving the expected result, found result, as well as location of the error. When arrays are translated into XML, each element is treated like a unique variable whose name is the root of the array plus its index in the array. This expansion from TethysL to XML is shown in Figure 6a and 6b. This creates an issue if a user declares a variable that ends in a number and an array where the translation to XML would create a variable with the same name. For example, a user declares someVar1 as well as the array someVar[1..2]. The variable someVar1 would be overwritten in XML by the first element in the array. For this reason, we added another check to AST validation that would review the names of arrays and return an error if they would overwrite existing variables in the mission. This is also done the other way around, so all variables are checked to see if they would overwrite any of the variables that will be created in XML by an array. Array accesses are similarly translated so that the variable used in XML is the root of the array plus the index that is being accessed as shown in figure 6c.[4]

**Macros**

For macros, we stepped all the way back to the lexer and the grammar definition. We introduced "macro" as a new keyword to be used in the parser to initiate a macro block, consisting of the header and an aggregate within the block. A TethysL file can define either a mission or an aggregate. In the latter case, the aggregate is referred to as *top-level*; however, aggregates can also be defined and used as an element within a mission or top-level aggregate. These *non-top-level* aggregates are the only construct that is allowed within a macro block. During AST validation, this aggregate is essentially validated the same way as other non-top-level aggregates just taking into account any replacements of the macro variable. Additionally, the bounds specified in the heading of the macro are checked, both ensuring that the upper bound is greater than the lower bound and that if

[4] This compilation was then further expanded on by my mentor to allow setting particular sub-ranges in an array declaration.

there are array accesses within the macro, the bounds of the macro are within the bounds of the array. The macro variable that is declared in the header is saved in a name resolver object to indicate when the program is within a macro block and what the current macro variable is. This information is an optional non-sequence variable, so either the program is in a single macro or it is not, but as of now, no nested macros are allowed. An error will be thrown if a user tries to nest them. Within a macro block, macro variables can be used in array accesses, names of aggregates, and in simple number expressions. Each of these must be identical to the macro variable declared in the header and if used in the name of an aggregate, must only appear at the end of the name (macro variables as a prefix or in the middle of the name are not supported yet). Like the arrays, macros must also not exceed 100 iterations. Translation from TethysL to XML for macros is shown in Figure 7. For each number in the range of the bounds described in the header, the aggregate within the macro block is translated to XML with each macro variable occurrence being replaced by the number. This expands the macro into multiple aggregates and substitutes in the correct value for each variable.



```
macro $i = 1..3 {
  aggregate Wpt$i {
    run in sequence

    behavior Guidance:Waypoint in sequence {
      setting latitude = Lat[$i]
      setting longitude = Lon[$i]
    }
  }
}
```

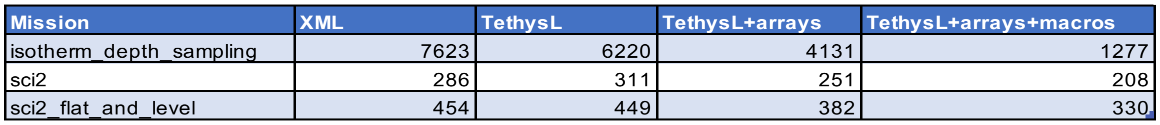
```
<Aggregate Id="Wpt1">
    <Sequence/>

    <Guidance:Waypoint>
        <Sequence/>
        <Setting><Guidance:Waypoint.latitude/><Arg Name="Lat1"/></Setting>
        <Setting><Guidance:Waypoint.longitude/><Arg Name="Lon1"/></Setting>
    </Guidance:Waypoint>
</Aggregate>

<Aggregate Id="Wpt2">
    <Sequence/>

    <Guidance:Waypoint>
        <Sequence/>
        <Setting><Guidance:Waypoint.latitude/><Arg Name="Lat2"/></Setting>
        <Setting><Guidance:Waypoint.longitude/><Arg Name="Lon2"/></Setting>
    </Guidance:Waypoint>
</Aggregate>

<Aggregate Id="Wpt3">
    <Sequence/>

    <Guidance:Waypoint>
        <Sequence/>
        <Setting><Guidance:Waypoint.latitude/><Arg Name="Lat3"/></Setting>
        <Setting><Guidance:Waypoint.longitude/><Arg Name="Lon3"/></Setting>
    </Guidance:Waypoint>
</Aggregate>
```

Figure 7. A TethysL macro block and its translation to XML.

## RESULTS AND USER FEEDBACK

After implementing arrays and macros, discussions with Yanwu Zhang, one of the mission script authors, were used to gain feedback as well as qualify our results. We also used mission script length to quantify our results, as fewer lines is an indication that the scripts will be more readable and allow users to more easily understand what the vehicle is doing. It is important to remember that improvements to TethysL will not impact the actual execution speed of missions on the LRAUV because all TethysL files are compiled into XML files. The readability of TethysL and any following language extensions are syntactic sugar meant to improve user experience. Therefore, results are dependent on this user experience and what is most readable for the operators. A few of the success factors for DSLs include Learnability and Usability (Hermans F et al.). Through user feedback, we can begin to speculate whether arrays and macros increase these factors for TethysL, making it more likely to be a successful DSL.

Three mission scripts were chosen based on frequency of use and potential impact by arrays and macros. We then edited these to use arrays and macros and recorded the new mission script lengths shown in Figure 8; Figure 9 shows these numbers as graphs.



| Mission | XML | TethysL | TethysL+arrays | TethysL+arrays+macros |
|---|---|---|---|---|
| isotherm_depth_sampling | 7623 | 6220 | 4131 | 1277 |
| sci2 | 286 | 311 | 251 | 208 |
| sci2_flat_and_level | 454 | 449 | 382 | 330 |

Figure 8. Three mission scripts were chosen and edited to use macros and arrays; the length of the scripts in number of lines was recorded.

The mission isotherm_depth_sampling was chosen due to the large number of scientific ESP samples involved. For this reason, the length of the mission was the most impacted by macros and arrays of all three missions chosen. The other two were sci2 and, the more recently created, sci2_flat_and_level. These both saw improvement by the array and macro extensions, although it was less dramatic as there were only 14 variables (7 longitude and 7 latitude) and 7 aggregates used as waypoints which could be condensed into 2 arrays and 1 macro. This contrasts the 420 variables and 60 aggregates in isotherm_depth_sampling which were reduced to 7 arrays and 1 macro. However, including comments and white space, there was still a noticeable difference in readability, even with these smaller scripts. A number of mission scripts use ESP samples or waypoints, therefore the potential to reduce the number of lines in LRAUV missions is high.

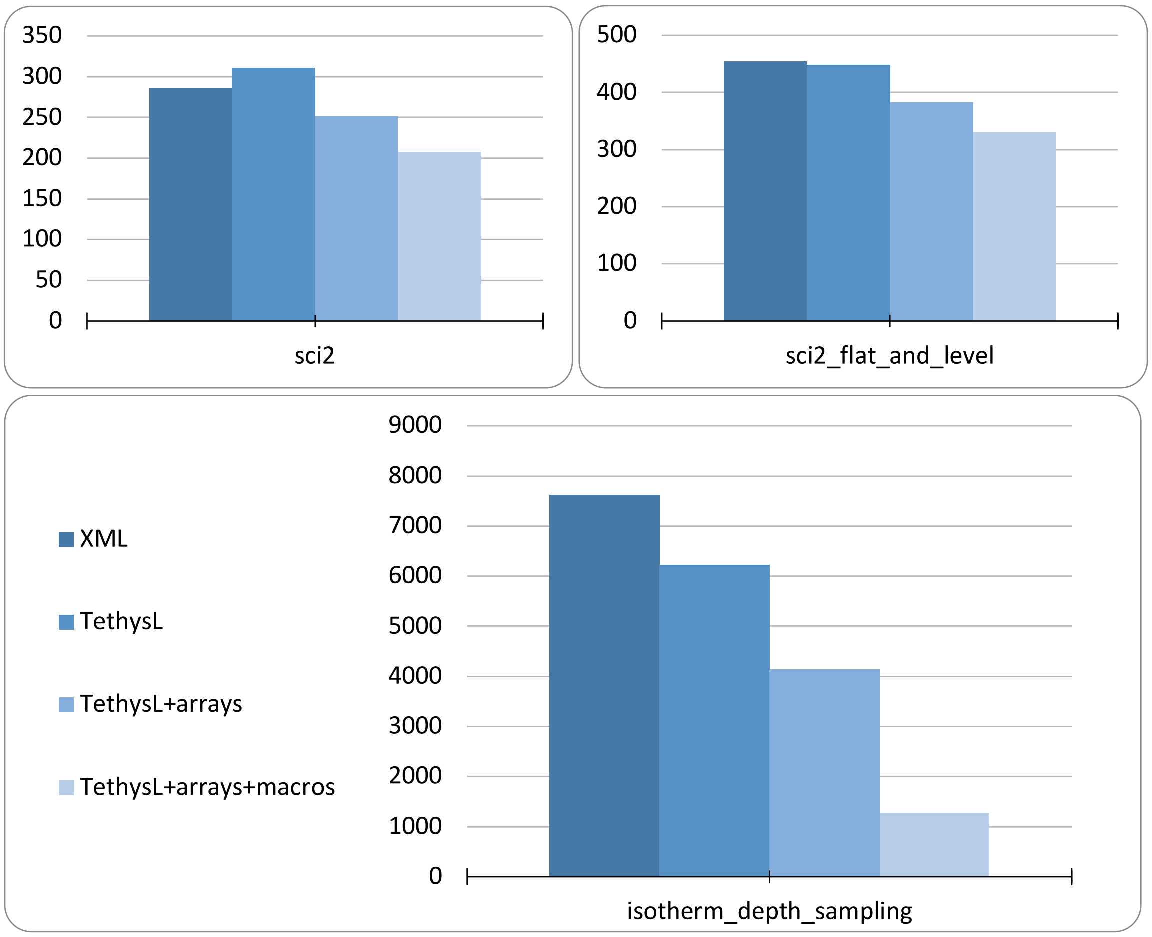


Figure 9. The number of lines in each mission decreased with the addition of arrays and with the addition of macros.

Through discussions with Zhang, we received feedback that array and macros would help operators save time and make less errors that often come from having to copy and paste code. This turns TethysL into a more reliable language, as it limits potential user errors.

Macros and arrays are the first language extensions in TethysL which deviates from to original XML schemas. Until now, each element in TethysL had an equivalent one in XML. Operators who are accustomed to using XML have even more incentive to switch to using TethysL because there are now tools for them not available in XML and they can express what they want to vehicle to do in fewer lines of code. Explicitly, they no longer have to copy and paste aggregates or variables in order to just change one number.

The extra features make TethysL more readable, and therefore more learnable to those who do not know the LRAUV mission framework. Operators who already understand the ideas of arrays and macros will be able to more quickly get the idea of what a vehicle is doing given a macro or array, rather than a whole bunch of aggregates or variables.

## DISCUSSION

The reason we chose to implement what we did and where we did it, depended on several factors and could have been done differently if we were not limited by the framework for LRAUV missions. One example of this is the reason we chose macros rather than implementing something more powerful like functions like you would see in a general-purpose programming language. Functions are not supported in the XML schemas and there would be no way to implement them in TethysL so that it is equivalent to the XML. Additionally, functions would not even be able to be implemented according to the underlying LRAUV mission script format described in Godin's paper. This model can be seen as a state machine, which calls behaviors, but there is no real concept of functions in a state machine. Since macros are simpler and can be expanded to XML, this was a reasonable alternative. The macros were also a natural continuation of arrays, in fact, since the benefits of macros are almost completely dependent on arrays and vice versa, the two features could be considered a single array-macro feature. It would be a common workflow for similar variables to be added to an array and then have a macro that does something with each of these elements, so the two really go hand in hand. This is also part of the reason we stuck to only allowing aggregates in macros, as this would be the most commonly-used feature.

As an alternative to implementing arrays and macros in TethysL, they possibly could have been added to the XML schemas instead and then implemented in TethysL. Arrays would be possible, but macros may not be possible at that level. If they were implemented in the XML schemas, this would have kept XML and TethysL more directly comparable. However, given the longer-term goal of getting rid of the XML step and translating directly to the C++ framework, there would be no use in continuing support for the XML schemas. Furthermore, if new features were implemented on both ends, operators that are used to XML would not have as much incentive to switch over to TethysL, instead just learning the new features in XML. This would make the transition to TethysL as the default scripting environment more difficult.

## CONCLUSIONS/FUTURE WORK

Overall, the array-macro extension has worked to give more usability to TethysL and hopefully in the future will be used by mission script authors to write simpler and more readable missions scripts. Building on the codebase that has been in development since 2016, starting with the initial prototype put together by Meckler (2016), we were able to build new features to increase the usability and learnability of TethysL. These two are both important factors to a DSLs success according to Hermans F. et al. and will contribute to encouraging operators to switch to TethysL from the currently used XML. Although XML allows for the required functionality, its usability from the user point of view is far from ideal and TethysL aims to solve this. The array-macro extension has helped to work towards this goal by allowing for shorter, more readable mission scripts.

There are still many ideas to explore to continue the development of TethysL. One of the larger, long-term goals for TethysL is to cut out the XML stage and translate directly to the C++ framework. In this case, new modules would be required in the LRAUV framework to directly process the TethysL format. Before this is done, it would be a good idea to include further testing to ensure the equivalence between TethysL and XML. Currently operator precedence and expression evaluation is being looked into and further developed to match the underlying LRAUV framework, but still has to be finished. On top of these more fundamental projects, it would be helpful for mission script authors to have even more tools. This could be more language extensions; for example, *structs* for composite data elements would be helpful in storing latitude and longitude variables in a single variable entity. More discussions with users would be helpful in facilitating this to choose more useful extensions.

## ACKNOWLEDGEMENTS

Firstly, I would like to thank Carlos Rueda for all of the support in this project and teaching me about the many facets of the TethysL translator. His passion and patience helped to make this project possible. I would also like to thank Brett Hobson for welcoming me to MBARI and into the LRAUV team for the summer. A big thanks to George Matsumoto and MBARI for making this internship program possible.

## References:

Godin, M. A., Bellingham, J. G., Kieft, B., McEwen, R. (2010). Scripting language for state configured layer control of the Tethys autonomous underwater vehicle. Presented at the OCEANS, 2010 MTS/IEEE, Seattle, OR.

Haoyi, L. (2019). Scala FastParse API. lihaoyi.com/fastparse . Last accessed: August 12, 2019

Hermans F., Pinzger M., van Deursen A. (2009) Domain-Specific Languages in Practice: A User Study on the Success Factors. In: Schürr A., Selic B. (eds) Model Driven Engineering Languages and Systems. MODELS 2009. Lecture Notes in Computer Science, vol 5795. Springer, Berlin, Heidelberg

Martin Fowler, Rebecca Parsons. (2010) Domain-Specific Languages. Addison-Wesley, 2010.

Meckler, Eli. (2016). TethysL: A Domain-Specific Language for LRAUV Mission Scripts.