TethySL: A Domain-Specific Language for LRAUV Mission Scripts

Eli Meckler, Williams College

Mentor: Carlos Rueda

Summer 2016

Keywords: Tethys, TethySL, AUV, LRAUV, Mission Planning, Scripting

ABSTRACT

The Tethys-class LRAUV's long battery life provides a platform for studying water chemistry and biology far offshore and following longer-term events such as plankton blooms and upwelling events by drifting for long periods of time. These AUVs currently run mission scripts written by operators in XML following custom mission schemas. TethySL is an attempt to improve the user experience of writing mission scripts by designing a domain-specific language that captures the same functionality as the current system with a much simpler syntax and additionally provides immediate and clear error reporting. The prototype, while not yet reaching the full functionality of the XML scripts, provides a much better user experience and lays a solid groundwork for future development.

INTRODUCTION

The Tethys-class long-range autonomous underwater vehicles (LRAUV) have an impressive battery life, able to travel for up to 2000 kilometers at a speed of 1 meter per second or drift under the surface for around 2 weeks. Long-duration and long-range functionality provides the opportunity to perform experiments and

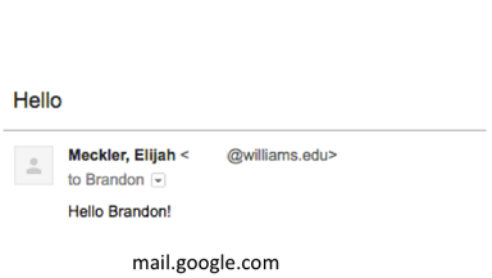
carry out observation not capable by other AUV lines currently in use at MBARI, such as gliders and the Dorado-class AUVs. Possible missions include tracking plankton blooms (Hobson et al. 2012) and upwelling events (Zhang et al. 2012).

Tethys-class AUVs run mission scripts written by operators and scientists in XML. The mission files are kept under version control (at https://bitbucket.org/mbari/lrauv-mission) and made available for submission to the vehicles during operations through the TethysDash application (https://okeanids.mbari.org/TethysDash/). These XML mission scripts are validated against a mission schema, confirming that the new script conforms to the expected format of the mission (Fig. 1). This insures some safety in mission structure: the mission must have certain components (such as a mission timeout) to mitigate risks and make sure the AUV can perform properly. After being validated, the XML is compiled into AUV-recognized code before being sent to the surfaced vehicle.

Element (or *Attribute*)
Id
DefineArg
Timeout
Behavior
Call
Insert
Syslog
Assign

Figure 1. The main components that make up LRAUV mission scripts (Godin et al. 2010).

However, due to XML's inherent structure (Fig. 2), the Tethys mission scripts contain large amounts of boilerplate code, making it difficult to read and time-intensive to write, especially for individuals not already comfortable writing programs.




```
<email>
  <Subject Id="Hello"/>
  <sender Id="Eli"/>
  <receivers>
     Brandon
  </receivers>
  <body>
     Hello Brandon!
  </body>
</email>
```

Figure 2. An email (minus actual email addresses) and how it might be encoded in XML.

Additionally, XML does not provide immediate feedback on errors, as most errors will be caught by the validator and all returned at one time. Although there exist sophisticated XML editors that could greatly facilitate the task for the user, they are not usually intended to be easily integrated in new applications, which is a requirement for the use of the tool in the LRAUV software ecosystem. A language dedicated to a particular domain application can afford a much simpler syntax when compared to XML and additionally provide more meaningful and instant feedback on syntactic and semantic errors.

TethySL is a domain-specific language for LRAUV mission scripting designed with simple syntax and useful, immediate error reporting as objectives.

MATERIALS AND METHODS

Currently, TethySL does not compile the user input into AUV-recognized machine code. Instead, the language produces the XML equivalent of the user's input. This was largely due to time constraints. Creating a compiler for a language is often more challenging and time-intensive than translating the language into another high-level language. By translating into XML, we can still utilize all the existing backend without requiring the user to actually interact with the XML directly. The

user inputs the new syntax, and TethySL produces XML for them, which can then be compiled and sent to the AUV.

TethySL consists of several modular components that transform the user's input into XML. First, input is fed to the parser to syntactically validate the input and to produce a corresponding Abstract Syntax Tree (AST). The validator then semantically validates the AST before the translator produces the input's XML-equivalent. All components are written in Scala for its compatibility with the existing TethysDash architecture (written in Java) and the availability of excellent libraries and tools that greatly facilitate the development of the TethySL language. In particular, these include the FastParse library (Haoyi, 2016) for parsing, and the Scala.js framework (https://www.scala-js.org/), which allows to target web browsers as the execution environment for the tool.

Parser and abstract syntax tree

User input is parsed into different components of the abstract syntax tree using the FastParse library, which enables custom construction of recursive descent parsers. This library allows for composition of *Parser* objects, such that one particular parser can be expressed in terms of other parser objects. In general, during execution, a parser searches for corresponding contextual syntax in the given input (e.g., "define { … }") and can record any information it is asked to capture into elements that can be used to build the AST (Fig. 3). (The FastParse *Index* parser, which is used in most implemented parsers in TethySL, allows to capture the current input stream location, so this information can later be used in semantic error reporting during validation.)

```
val setting = P(Index ~ identifier ~ "=" ~ expression ) map Setting.tupled
val define = P("define" ~/ Index ~ "{" ~ setting.rep ~ "}") map Define.tupled
```

Figure 3. Shown are two of the defined parsers in TethySL. The "setting" parser first records index, searches for an identifier (recording its value), and then the expression it's being set equal to following an equal sign ("="). The define parser, in turn, uses the setting parser: after recognizing the "define" keyword and recording the index, it finds as many settings as there are between to curly braces. The suffixes (beginning with "map") in both lines store the captured information in the corresponding AST component.

Information is captured in an abstract syntax tree (AST) represented by a Scala case class (http://docs.scala-lang.org/tutorials/tour/case-classes.html) hierarchy (Fig. 4).

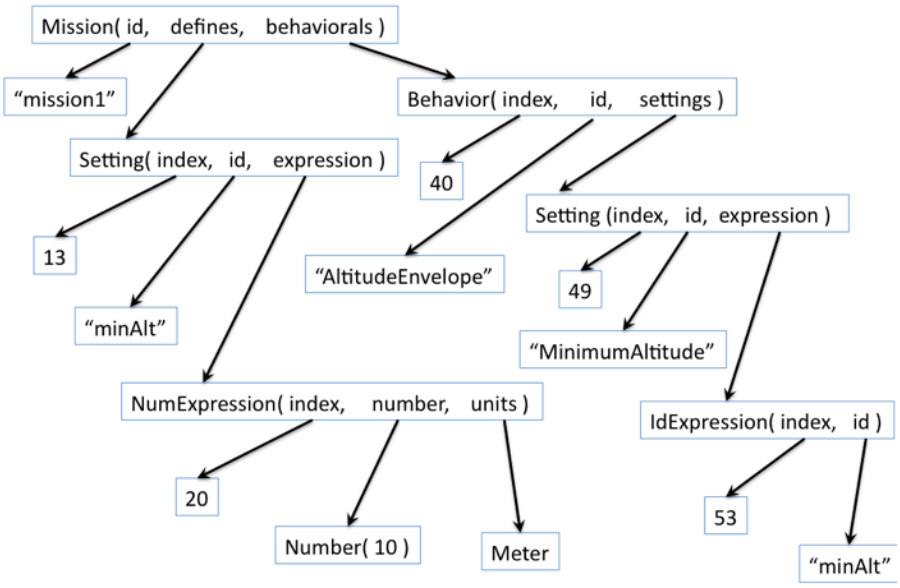


Figure 4. A basic AUV mission that defines one variable and then assigns it to a behavior setting has an abstract syntax tree of this form. Note how most elements track index; knowing where each component occurs in the input enables highlighting in the case of user error.

The top level is the mission, which contains an ID, a list of argument definitions, and then "behaviorals," which is a wrapper for anything contained within the main body of the mission. Argument definitions are a list of setting components, which contain an index, a variable name, and the expression being assigned to that variable name. Expressions are either numerical, other variable names, or arithmetic compositions of simpler expressions (Fig. 5).

```
case class IdExpression(id: String) extends Expression
case class NumberExpression(value: String) extends Expression
/.../
case class ArithExpression(left: Expression, ops: Seq[(String, Expression)]) extends Expression
```

Figure 5. In the AST, expressions take the form of a number followed by a unit (NumberExpression), a variable (IdExpression), or an arithmetic sequence of other expressions (ArithExpression). Unit value is a field of Expression (not pictured) that can be changed for each instance above.

Currently, the only "behaviorals" are behaviors, aggregate blocks, and insert blocks (Fig. 6).

```
sealed abstract class Behavioral
case class Aggregate(id: String, runType: RunType, behaviors: Seq[Behavioral])       extends Behavioral with MissionElem
case class Behavior(index: Int, id: String, runType: RunType, settings: Seq[Setting])extends Behavioral with MissionElem
case class Insert(index: Int, file: String, settings: Seq[Setting])                  extends Behavioral
```

Figure 6. The behavioral components of the AST have different fields reflecting their implementations. An aggregate block contains behaviorals (possibly including another aggregate) and tracks RunType, which describes when the aggregate block is run in comparison to other aggregates or components. Behaviors and inserts both contain lists of definitions represented by settings. These two are semantically validated, so their indices are also recorded.

Validator

The validator semantically analyzes the given mission script by traversing the AST built during the parse stage. Semantic validation has many steps depending on which component of the AST is being analyzed. Validation functions are called on subcomponents to construct a list of errors, if any. The first error in that list is reported to the user. Validation checks take place on numerous component levels,

often using a maintained symbol table for reference. All validation checks return the same class of error for ease of display in the user interface. The error objects returned contain "index," "expected," and "found" fields used for detailed error reporting.

At the mission level (Fig. 7), the validator calls helper functions to validate the list of argument definitions. Argument definitions are checked for uniqueness: no variable should be defined twice, as well as the assigned expression being semantically sound. As the helper function is called on each definition in the list, the symbol table is referenced to check if that variable has already been defined. If not, no error is returned and the symbol table is updated to include the variable name and its associated units.

```
private def validate(mission: Mission): AstValidationResult = {
  val Mission(id, assigns, behaviorals) = mission

  validate(assigns) match {

    case Some(efmi) => FailureAstValidationResult(info, efmi)
    case _ => {
        val behavioralErrors: Seq[EFMI] = for {
          behavioral <- behaviorals
          validated <- validate(behavioral)
        } yield validated

        if (behavioralErrors.nonEmpty) {
          val errorMsg = behavioralErrors.head
          FailureAstValidationResult(info, errorMsg)
        }
        else SuccessAstValidationResult(info)
      //}
    }
  }
}
```

Fig 7. Mission validation first validates all of the mission's argument definitions (referred to here as "assigns") in a helper method. If that process doesn't return an error, the behaviorals are validated in a similar way, finally returning an all-clear if no errors are thrown. If any of the helper functions do throw errors, they are passed up and shown to the user.

If the argument definitions produce no errors, the list of behaviorals is iterated across for semantic validation and broken down further from there depending on the behavioral case. For behaviors, the validator searches through a pre-constructed library of valid names, associated settings, and setting units to confirm that the given behavior and its settings are recognized (Fig. 8). If a behavior and its settings are found (and thus valid), the validator checks the

settings to insure that the user assigned a value of the appropriate measurement family (e.g., distance, temperature, etc.).

```
def validate(behavior: Behavior): Option[EFMI] = {
  if (behaviors.contains(behavior.id)) {
    behavior.settings.view.map(a => validate(behavior, a)).collectFirst{case Some(efmi) => efmi}
  }
  else Some(EFMI(
    index = behavior.index,
    expected = s"a valid behavior",
    found = s"${behavior.id} is an undefined behavior"
  ))
}
```

Figure 8. The behavior validator checks the pre-constructed list of valid behaviors to verify that the user-specified behavior is recognized by the the AUV. If so, its settings are validated in a similar manner. If the behavior isn't recognized, it creates an error instance with enough information to produce a helpful error message.

Similarly, the insert instance of a behavioral is checked for file existence on the server. Since files can be saved without being error-free, the inserted file is recursively parsed and validated. If successful, the inserted file's symbol table is returned and used to confirm that any changed argument definitions do exist (this is not currently functional; see Discussion below) and that they are assigned an appropriate value.

An aggregate instance of a behavioral is not itself checked (as all it contains is information validated by the parser), but the helper function recursively validates the behaviorals the aggregate contains, returning the first error found within (if any).

At the bottom level of most branches, the validator semantically checks expressions, tracking unit family consistency and existence of referenced variable names. Arithmetic expressions must have all individual expressions validated and compared to make sure units are consistent. If no error is found, then the helper function returns the expression's units, which the symbol table stores.

Translator

The validated AST is then translated to XML component-by-component, as each AST element corresponds to an existing structure in the XML representation of a mission (Fig. 9). The AST is traversed and XML is generated with the specific values stored in the AST. The output is a string representation of the XML.

```
private def insertToXML(source: Insert, indents: Int): String = {
  val indent = "    " * indents

  if (source.settings.isEmpty) {

    s"""
       |$indent<Insert Filename="${source.file}"/>
     """.stripMargin
  } else {

    s"""
       |$indent<Insert Filename="${source.file}">
       |
       |${redefinesToXML(source.settings, indents + 1)}
       |$indent</Insert>
     """.stripMargin
  }
}
```

Figure 9. Translating an insert component to XML can happen one of two ways. If the insert contains any redefinitions (marked here as "source.settings"), it takes the form <Insert Filename="…"> … </Insert>. If there are no redefinitions, only one tag is needed, and it takes the form <Insert Filename="…"/>

Error reporting

Both the parser and the validator can return errors that are passed along to the user. FastParse provides built-in error reporting functionality by returning a failure instance after an unsuccessful parse that includes the failed parser, what string it failed to parse, and the index at which the failure occurred. Combined with the user interface, these provide a meaningful error message to the user.

The validator, in the case of a semantic error, returns an error message containing an index, what was expected, and what was found. Unlike in the parser, the "expected" and "found" messages are often hard-coded messages specific to the helper function that originally caught the error, sometimes including specific contextual information.

User interface

Carlos Rueda, my project mentor, developed the user interface (UI). It has a window for the user to type the new language's syntax, and to the right it displays the XML translation in real time (Fig. 10). In the case of an error, the user's input that triggered the error is highlighted, and an error message appears on the right side.

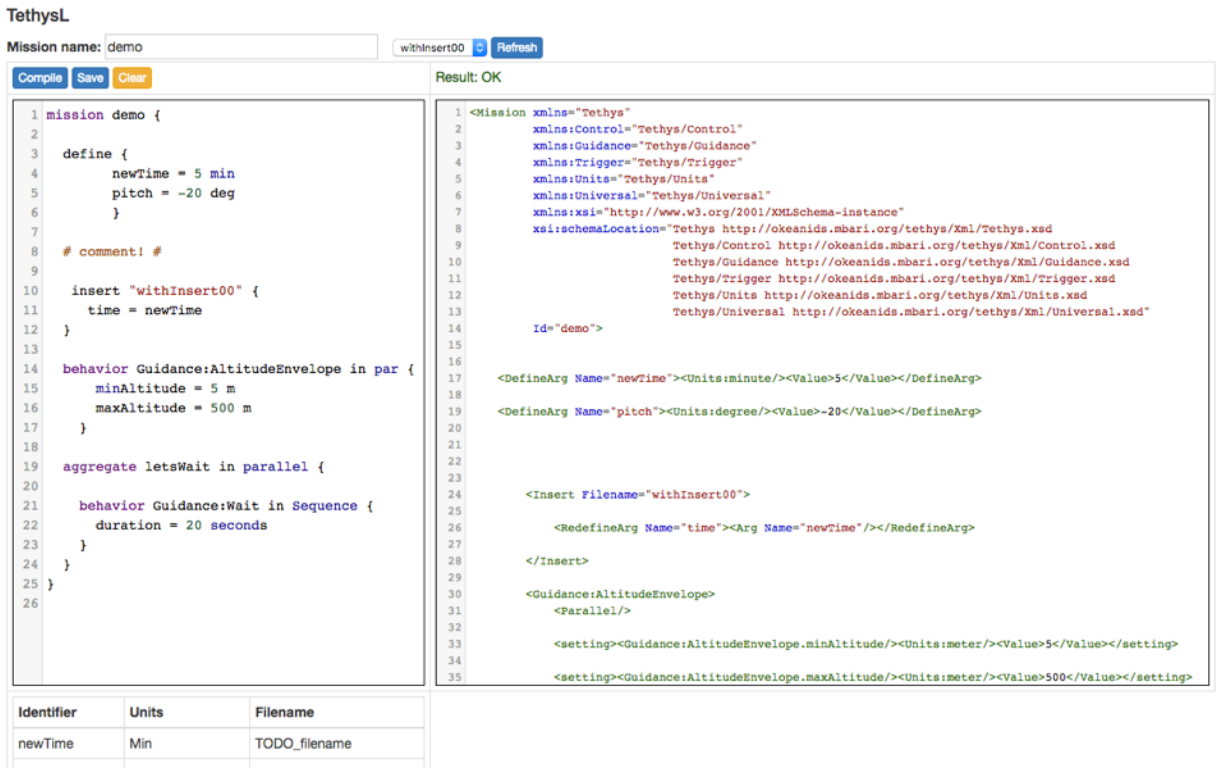


Figure 10. The user writes the mission script in the new TethySL syntax on the left while XML (or error messages) are automatically generated on the right. Additionally, the user can save files and open ones saved earlier.

RESULTS

Syntax

TethySL's user input syntax follows a much more streamlined structure when compared to equivalent statements in the XML mission script (Fig. 11). By replacing XML-style open and close tags with a keyword and brackets, TethySL

mission scripts appear much less cluttered and provide the same necessary information with much less boilerplate syntax.

```
behavior Guidance:AltitudeEnvelope in
parallel {
      minAltitude = 5 m
      maxAltitude = 500 m
    }
```

```
<Guidance:AltitudeEnvelope>
    <Parallel/>

    <setting><Guidance:AltitudeEnvelope.minAltitude/><Units:meter/><Value>5</Value></setting>

    <setting><Guidance:AltitudeEnvelope.maxAltitude/><Units:meter/><Value>500</Value></setting>

</Guidance:AltitudeEnvelope>
```

Figure 11. Both of the above sections of code provide the same information. The top section, written in TethySL, is much clearer and easier to write.

Error Reporting

Errors from both the parser and the validator are shown to the user through the UI. The erroneous input is highlighted, and the "expected" and "found" messages display for the user. The UI does this in real time in response to user error (Fig. 12).

```
define {
      newTime = 5 min
      pitch = -20 deg
      } &
```

Expecting: behavior | aggregate | insert | "}"

Result: Error

Expected:	behavior | aggregate | insert | "}"
Found:	&
Position:	Line: 6, Column: 12

Figure 12. After the user makes an error, it is immediately highlighted. An error message displays, telling the user what went wrong.

Code repository and test deployment

All code developed for TethySL can be found on its Bitbucket repository at https://bitbucket.org/mbari/tethysl. The test system has been deployed at okeanids.mbari.org/tethysl.

DISCUSSION

Improved user experience

To reiterate, the goal of TethySL is to improve the user experience for writing mission scripts. Specifically, the language sought to accomplish this by making mission scripts easier to read and write and by providing immediate and useful feedback on user input. These two characteristics provide a clear benefit over writing mission scripts in XML. The prototype as described in this report demonstrates that this is certainly obtainable. While not yet implementing all features and elements of AUV mission scripts, the language does showcase how a domain-specific language can improve the user experience.

The syntax, when compared to XML mission scripts, is much simpler and easy to read. In Fig. 3, we see a comparison of argument definitions in TethySL versus in XML. Note that the latter contains redundant syntax and obscured readability. Additionally, the syntax can be further updated in response to user feedback in order to maximize ease of use. This will be an ongoing process as more scripts are written in TethySL.

The error messages provided by TethySL are instantaneous and incredibly useful. Appearing immediately after the error is made, the messages inform the user where the error is in the script, what the language was expecting to find, and what was found instead. A user with limited exposure to programming and script writing can easily understand these messages and fix the script in response to this feedback.

Incomplete validation

The semantic validation does not catch all errors as of yet. The known errors with validation are very unusual scenarios, but are nonetheless important to note and/or fix:

- User should not be able to give variables the name of unit identifiers.
- The inserted file's symbol table is not being properly used, as non-existent variables can be redefined within an Insert block.
- Inside a behavior block, a user cannot reset one setting to be equal to another one of the behavior's settings. This may or may not be a benefit: eliminating the possibility enforces good programming practice, and it does not limit the expressiveness of TethySL. Operators should give feedback on this matter.

Incomplete implementation

Not all mission script elements and functionality are implemented in TethySL as of this prototype. Still to be implemented are the following:

- Timeout command: while existing in the code, it is not complete and thus left commented out in the final implementation. In the AST, it should be categorized as a Behavioral.
- Syslog command: exists in the AST, but is not currently being used by any other components. It should be categorized as a Behavioral.
- Call command: exists in the AST, but not currently used. It should be categorized as a Behavioral.
- Assign command: not implemented in the AST. It should be categorized as a Behavioral and needs to be able to update an inserted mission file's settings by using the file's alias.

- Insert alias: inserted mission files can be given an alias name for use within the mission script for use with Assign, but it is not mandatory.
- Boolean units: Booleans are special units in the XML mission script in that they are implemented differently than all other units.
- Boolean expressions: similar to numerical expressions but not yet implemented due to trouble with Boolean units. However, they can be modeled after the arithmetic expression structure to avoid infinite recurrence with a recursive descent parser.
- Conditionals: this includes "while," "when," and "until" (and maybe others, unknown). Conditionals are used as control blocks inside aggregates.
- Non-integer number support: this should be a very straightforward implementation, but it does depend on which number types (floats, doubles, etc.) are used by the AUV.
- Repeat: RunTypes can set how often they repeat. See sci2.xml's "Lap" aggregate block for a usage example.

Once these remaining mission elements are implemented, TethySL will be able to write any mission script that can be written in XML.

CONCLUSIONS/RECOMMENDATIONS

Tethys-class AUVs provide scientists with the opportunity to collect data not previously possible with autonomous vehicles, and easily written mission scripts are a large part of that process. When compared to XML, TethySL vastly improves user experience by providing an easier language to understand and write and by relaying immediate feedback to the user. The language has the potential to remove XML from the user's experience entirely.

Future work

I suggest future work be done to make TethySL an integral part of LRAUV missions. Firstly, I recommend implementing the remaining mission script elements as outlined above. This is the most crucial since it will allow a user to write any and all valid LRAUV mission scripts. Secondly, I recommend increasing semantic validation sophistication. While only one of the caught errors is a serious problem, it is possible that other errors exist; it warrants a more thorough investigation. Thirdly, I suggest adding advanced editing features such as auto-complete further down the line. Features frequently found in development environments are often there to improve user experience and should be emulated if possible. Finally, eventual integration with the TethysDash web application such to remove XML entirely from the mission pipeline would simplify the code base and homogenize all AUV mission scripts.

ACKNOWLEDGEMENTS

Many thanks to Carlos Rueda, my mentor, for all he taught me and for his involvement in this project. I couldn't have done it without his help. Thank you to Mike Godin and Brian Kieft for providing clarification on the XML mission scripts. Special thanks to George Matsumoto and Linda Kunhz, the intern coordinators for all their work this summer. I also want to thank the other interns and the MBARI community as a whole for a great summer experience.

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. (2016). Scala FastParse API. lihaoyi.com/fastparse

Hobson, B. W., Bellingham, J. G., Kieft, B., McEwen, R., Godin, M., & Zhang, Y. (2012, September). Tethys-class long range AUVs-extending the endurance of propeller-driven cruising AUVs from days to weeks. Presented at the Autonomous Underwater Vehicles (AUV), 2012 IEEE/OES, Southampton, UK.

Zhang, Y., Godin, M. A., Bellingham, J. G., & Ryan, J. P. (2012). Using an autonomous underwater vehicle to track a coastal upwelling front. IEEE Journal of Oceanic Engineering, 37, 338–347.