Springleik: A Case Study in Small Languages

Mark S. Williamsen

FlexLink AB

November 14, 2026

Abstract

Small languages arise in software engineering for a variety of reasons. A simple command prompt can be useful in bringing up new hardware, particularly on small targets running embedded firmware. This leads naturally to a scriptable interface. Which can then evolve further into a primitive domain-specific language (DSL) that supports use cases in development, test, and even production. This work describes one path through the steps that lead from command prompt to small language, resulting in an abstract syntax tree (AST) interpreter implemented using ordinary programming facilities of the host language. Concrete syntax is generated automatically by serializing the AST, while a parser for the concrete syntax is optional. We describe a regression test framework based on asymmetric tree comparisons of serialized AST outputs.

1 Introduction

Interesting software problems can sometimes give rise to new programming languages, often in the form of a domain-specific language (DSL). For small targets with limited run-time resources, such a language can be implemented as an embedded or internal DSL [1]. If one begins a project with this in mind, they might be tempted to start by defining a concrete syntax and then developing a parser for that syntax. In this paper however we consider the case where the need for a DSL was not anticipated, and therefore no concrete syntax exists. The workflow presented here reverses the process by putting abstract syntax first, allowing concrete syntax to be postponed until needed or perhaps even indefinitely [2].

In this work abstract syntax appears in the form of a composite tree data structure, composed using standard facilities of the host language. We will show that this can be done without having defined a concrete syntax, and therefore without relying on a parser [3]. Examples of abstract syntax tree (AST) interpreters in Python, Java, and C are given in the Springleik release on Zenodo [4]. Some authors have described syntax-directed or projectional editing as techniques for putting abstract syntax first [5]. This however introduces some difficulties, in that it lacks a text-based representation to work with customary versioning and differencing tools. Similarly, test strategies will suffer when a proprietary editor is required for "coding without code" in the DSL. Implementation of an AST interpreter can proceed without a big investment in planning

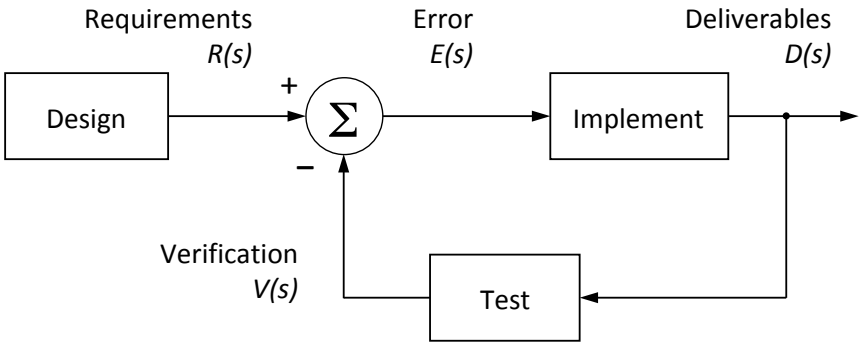


Figure 1: Software development process rendered as a closed-loop simulation diagram. Negative feedback from verification $V(s)$ causes deliverables $D(s)$ to evolve, tracking changes in requirements $R(s)$. When all requirements have been met, the error term $E(s)$ goes to zero and the control loop is in steady state.

or technology [6]. As described here it can be done with an existing development environment using tools and techniques already on hand [7]. Rather than draining resources, it can add value by facilitating robust software implementation, verification, and documentation.

2 Model of software development

Many models of software development are available for this discussion. Here we present a simple model based on closed loop control theory, which includes concepts of time dependence and iteration [8]. Figure 1 is a simulation diagram representing a software development process. The input on the left is requirements $R(s)$, and the output on the right is executable code and other deliverable assets $D(s)$. In between we have a means for creating code *Implement* positioned as the plant in control theory, and a means for verifying code *Test* positioned as the sensor. The summing junction on the left compares verification results $V(s)$ with requirements $R(s)$ to produce an error term $E(s)$, which then drives ongoing implementation. The directed edges connecting these blocks represent N-dimensional vectors where N is the number of requirements. The value of each dimension is the extent to which a given requirement has been met on some arbitrary scale.

A typical software project will begin with all vectors pointing to the origin: no requirements, no implementation, and no tests. As development proceeds we add requirements that drive the implementation, which is then tested and compared with requirements. Each directed edge conveys important information, but the most interesting vector is the error term $E(s)$. This will wander in an N-dimensional problem space, with N steadily increasing as development proceeds. Ideally this vector should always trend towards zero, with distance from the origin giving us an indication of how far we are from done. Time dependence is not shown explicitly but one can imagine one or more integrators in the loop, with time constants we may or may not control, modifying the time

domain response of the development process.

Treating the development process as continuous we can write an expression for deliverables $D(s)$ in terms of implementation $I(s)$, test $T(s)$, and requirements $R(s)$, where s is the Laplace variable.

$$D(s) = R(s)\frac{I(s)}{1 + I(s)T(s)} \tag{1}$$

In this model we see that there are three things that matter: requirements, implementation, and tests. These pillars of software development should receive equal priority during development, so that all three can be considered as correct and complete when development ends and maintenance begins. As a practical matter however many projects end with one or more of the pillars incomplete. We claim without proof that given any one of these, the other two can be derived through various means. For the purposes of this paper we use the simulation diagram in figure 1 to define deriving implementation and tests from requirements as "forward engineering." Starting with an implementation or a test suite is then defined as "reverse engineering" [9].

Notably we have left source code out of this picture. Traditional workflows require source code as an input to Implementation. However in this work we describe concrete syntax source code for AST interpreters as an output of Implementation, which is not needed to close the loop. Fortunately the source code is easy to obtain and is useful for communicating about the design, and also in test and verification.

3 Object model for small targets

Developers of embedded software for small targets may be tempted to organize their object model around things the system has, such as inputs, outputs, sensors, and actuators. We submit however that a coherent design is more likely to result from an object model organized around things the embedded system does, such as "home an axis", "measure a sample", or "close a relay." The reason we think in terms of objects is so that we can throw them into a container and iterate over them. While there may be a use case for iterating over peripherals, the important use case here is iterating over a collection of nodes that represent actions a user might want to invoke. These are packaged as command nodes along with the arguments needed to carry out each action. Thus we replace a noun-oriented design with a verb-oriented design so that command nodes can be collected into sequences that can then be interpreted on live hardware, simulated hardware, or a hybrid of both. Let's look at an example of how such a design might arise.

Consider a small target device with embedded processor and several axes of motion control. On first bring-up we want to start characterizing the motion controller. We write a small monitor program to run on the target with a command prompt exposed on a serial port, or a socket port if so equipped. Then we write commands to initialize the motion controller, home an axis, move axis to a new position, and wait for position reached. Each command can be followed by one or more arguments giving details such as velocity, acceleration, current, position, etc. Running a terminal program on a host computer allows us to manually exercise various behaviors of the motion controller. We can stream

a text file containing motion commands from the host to the target's monitor program to execute a motion sequence of arbitrary complexity. Move commands however take finite time, and we wind up overrunning the monitor program with new commands before old ones are complete. So we write a program to run on the host and carry out a series of motions on the target using monitor program commands, but now waiting for a prompt from the target before sending a new command from the host.

4 Minimum footprint

Many readers will have gone through these steps in bringing up a new target. We don't yet have a small language, but many of the pieces needed are already in place. Now we implement our object model by declaring a class object for each command and an instance attribute for each argument the command needs. One can also add an instance attribute for the command's return value, if that value is meaningful. Now we have a small language we can use to carry out the following steps:

- Instantiate one or more class objects representing command nodes to be executed.
- Populate the nodes with arguments appropriate for the desired commands.
- Compose the nodes into a sequence by appending them to an ordered collection.
- Iterate over the collection, interpreting each node as a command to be sent to the target.
- Capture results returned from each command, either as attributes of the command nodes or in a separate trace or listing.

Let's look at what we have. The class objects representing command nodes can live almost anywhere: on the host, on the target, in the cloud, or anywhere else that can interpret them as commands with arguments. For now we think of the arguments as being hard-coded, but they could just as well be derived or symbolic. Various types of collections can be used to iterate over the command nodes, as long as the iteration follows a predictable, repeatable path. The commands themselves can be sent to one or more target devices, or to any other device or simulation with an interface to receive them. Command replies such as motion results or error messages in our motion control example can go back to the host, to some other monitor, or even be ignored. This then is the bare minimum of what's needed to claim we have a small domain-specific language [10]. This can be prototyped and running in a matter of a few days, completely avoiding the trap that "nothing works until everything works."

5 Serialization

Now let's look at what's missing. To obtain a text representation of the interpreted program we'll need a means to serialize the collection of command

```
[
  {"cmd": "initControl"},
  {"cmd": "setRunCurrent", "axis": 1, "current_mA": 500},
  {"cmd": "setHoldCurrent", "axis": 1, "current_mA": 500},
  {"cmd": "setVelocity", "axis": 1, "steps_sec": 10},
  {"cmd": "setAccel", "axis": 1, "steps_sec2": 10},
  {"cmd": "homeAxis", "axis": 1},
  {"cmd": "waitPosition", "axis": 1},
  {"cmd": "setPosition", "axis": 1, "degrees": 200},
  {"cmd": "waitPosition", "axis": 1},
  {"cmd": "setPosition", "axis": 1, "degrees": 0},
  {"cmd": "waitPosition", "axis": 1},
  {"cmd": "setPosition", "axis": 1}
]
```

Figure 2: JSON representation of a simple command list being sent to an embedded motion controller. The top-level container is an array that preserves order. Arguments here are named, but could also be positional. JSON text copied from file *two.json* in the Springleik project release on Zenodo [11].

nodes. We have many well-established choices such as JSON, XML, YAML, etc. We also have the freedom to design our own representation if warranted, or dispense with source code altogether. In this paper we will assume without loss of generality that JSON has been chosen. Experience shows that it is trivial to give command node classes a serialize method, which is specialized for specific commands and arguments. To obtain a program listing we iterate over the collection while serializing each node. If we have multiple interpreters (host, target, cloud, etc.) all should produce identical program listings for a given collection of command nodes. Returning to the example of an embedded motion controller we can look at what the serialized JSON representation might look like in figure 2. Here we have used the *setPosition* command node as both a setter with the *degrees* argument, and as a getter without it. It should be mentioned here that while off-target interpreters can use the monitor command prompt interface, on-target interpreters don't have to since they can call directly into functions on the target.

6 Control nodes

Real programming however implies branching and looping, which is easily added by defining control nodes for conditional and repeated execution where the arguments represent loop count, limits for comparison, etc. At this point we've only described programs that are a simple list or sequence of nodes. Now we need to expand our definition of command nodes to include a node list attribute, allowing us to represent composite tree structures. But not every node needs to have a list. So we divide node classes into composite or *branch* nodes that have a list, and terminal or *leaf* nodes that don't. Clearly conditional and repeating nodes should have lists, to which we append nodes that will be executed only if the condition evaluates as true, or that will be repeated in iterations controlled by the repeating node. Our class hierarchy can begin with the base class *node*

```
{"cmd": "initControl", "list": [
  {"cmd": "setRunCurrent", "axis": 1, "current_mA": 500},
  {"cmd": "homeAxis", "axis": 1},
  {"cmd": "setVelocity", "axis": 1, "steps_sec": 10},
  {"cmd": "setAccel", "axis": 1, "steps_sec2": 10},
  {"cmd": "setRunCurrent", "axis": 2, "current_mA": 500},
  {"cmd": "homeAxis", "axis": 2},
  {"cmd": "setVelocity", "axis": 2, "steps_sec": 10},
  {"cmd": "setAccel", "axis": 2, "steps_sec2": 10},
  {"cmd": "iterateRegister", "reg": 0, "start": 0, "stop": 200,
    "step": 20, "list": [
    {"cmd": "setPosition", "axis": 1, "degrees": "reg"},
    {"cmd": "waitPosition", "axis": 1},
    {"cmd": "iterateRegister", "reg": 0, "start": 0, "stop":
      100, "step": 10, "list": [
      {"cmd": "setPosition", "axis": 2, "degrees": "reg"},
      {"cmd": "waitPosition", "axis": 2}]}]},
  {"cmd": "testFlag", "flag": 1, "trueIf": 1, "list": [
    {"cmd": "setHoldCurrent", "axis": 1, "current_mA": 50},
    {"cmd": "setHoldCurrent", "axis": 2, "current_mA": 25}]}]}
```

Figure 3: JSON representation of a composite command tree with conditional and repeating nodes. The top-level container is a dictionary and command node that does not preserve order. Subordinate node lists are contained in arrays that do preserve order. JSON text copied from file *three.json* in the Springleik project release on Zenodo [11].

which has two subclasses, *branch* and *leaf*. Command and control nodes can then be derived from *branch* and *leaf*.

All node classes should have methods to execute and to serialize them while traversing a command tree by recursive descent [12]. The execute and serialize methods of composite nodes must also iterate over the node list beneath them, calling the execute or serialize method of each subordinate node in turn. The execute method of a composite node can include entry and exit code, to run before and after iterating over the node list. This implements an AST interpreter, fulfilling the application's semantic model without requiring concrete syntax [13]. In fact, the concrete representation can be obtained by projection using the tree's serialize methods [14].

In figure 3 we look at the JSON representation of a command tree that includes conditional and repeating nodes. Iteration is done by the new "iterateRegister" control node that changes the value of a register attribute, which is visible to nodes beneath the control node. Nested loops are supported here by placing the iteration of axis 2 inside the iteration of axis 1. At the end of the sequence a "testFlag" control node tests a flag to check whether to return to holding current.

The ability to implement tree structures composed of command and control nodes, using all of the infrastructure we've developed so far, brings into focus a scalable, portable, and verifiable framework for representing interpreted programs optimized for the application at hand. The composite tree structure maps directly to structured text representations such as JSON, XML, or YAML.

```
    # composite command tree from figure 3
    elif index == 3:
        node.tree = initControl()
        node.tree.append(setRunCurrent(1, 500))
        node.tree.append(homeAxis(1))
        node.tree.append(setVelocity(1, 10))
        node.tree.append(setAccel(1, 10))
        node.tree.append(setRunCurrent(2, 500))
        node.tree.append(homeAxis(2))
        node.tree.append(setVelocity(2, 10))
        node.tree.append(setAccel(2, 10))
        outerLoop = iterateRegister(0, 200, 20)
        node.tree.append(outerLoop)
        outerLoop.append(setPosition(1, 'reg'))
        outerLoop.append(waitPosition(1))
        innerLoop = iterateRegister(0, 100, 10)
        outerLoop.append(innerLoop)
        innerLoop.append(setPosition(2, 'reg'))
        innerLoop.append(waitPosition(2))
        testBranch = testFlag(1, 1)
        node.tree.append(testBranch)
        testBranch.append(setHoldCurrent(1, 50))
        testBranch.append(setHoldCurrent(2,25))
```

Figure 4: Python code snippet from function *figureFactory* in file *host.py* in the Springleik project release on Zenodo [11]. Shown are the steps to instantiate and compose the command tree serialized in figure 3.

Human-readable program listings are easily obtained by serializing nodes while traversing the tree by recursive descent. In this picture we can now say clearly that command nodes are dictionaries composed of key-value pairs representing the command name and all of its arguments. AST nodes can also contain an ordered list identifying subordinate nodes.

7 Memory allocation

Command tree programs that rarely change can be composed of statically allocated nodes. Programs that could change at run-time however will be composed of nodes taken from a node pool, or using dynamically allocated memory. In garbage-collected environments dynamic nodes will evaporate after they are no longer referenced. Releasing tree nodes without garbage collection is also easy, since we can ask the root node to delete itself and all subordinates recursively. For this to work all nodes in the tree should be dynamically allocated and each node should appear in the tree exactly once. It's easy to maintain a global node counter to check for leakage (block not freed) and double frees (block freed twice) at run-time. An example of this appears in the variable *nodeCount* in file *asynTree.c* in the Springleik project release on Zenodo [15].

In figure 4 we show an example of instantiating and composing a command tree in a Python function that implements a tree factory. That combined with a

serialization function gives us a human-readable representation of the command tree program as structured text. In other words, the source code describing the AST program is written for us, and we can put off writing the DSL source code parser as long as we like. One can instrument the command tree by adding node attributes at run-time such as time stamps, tally counters, iterator values, and reply strings. When added to serialization functions, these attributes then appear in the structured text source code with each attribute conveniently attached to the node where it originated. We describe this transformation as "decorating the tree." Serializing the tree before and after execution shows clearly where the active paths lie. If memory permits, one can consider keeping time series or position series data in the same way, appending new data points to a list maintained by the node where the data points originated. In motion control applications, for instance, one can record axis position while polling for motion done. Such data nodes can be given a serialize function, while their execute function should be a simple pass-through.

8 Analysis

Capturing data points like shaft position will be fairly lightweight, perhaps just reading a shaft encoder. Ideally this can be done with no impact on system performance. However in some cases we will want to perform further analysis, for instance extracting fitted parameters like velocity and acceleration from time series data. Since the raw data exists in any serialization done after execution, we can perform analysis wherever the serialized command tree exists. An obvious choice would be to run analysis in the same process that runs the interpreter. In other words we take a tree that already supports execution and serialization, and then add an analysis method, all traversing the tree by recursive descent. Alternately one can insert specialized analysis nodes into the tree, for the purpose of analyzing data nodes beneath them. Here we encounter the principle of *Separation of Execution and Analysis*. This will be discussed further in section 9.

One might wonder where to put the results of analysis, and we now propose that the best place is to append analysis nodes to the command tree, just as we previously appended data nodes. So any serialization of the tree done after analysis will contain the analyzed results attached to the nodes where they originated. Some applications will require multiple passes of analysis, which is easily accommodated in this technique by writing several analysis functions to be called in sequence. This is a design decision that needn't be taken until analysis is needed.

Reporting in any desired format can also be carried out by recursive descent, facilitated by inserting report nodes or adding a report method to existing node classes. Reporting can take many forms, such as tables, charts, graphs, or even pseudocode. Now we see the pattern, that each step in the workflow can be thought of as traversing a command tree by recursive descent. With each pass performing the desired operation and appending or outputting the results in a process that is scalable and verifiable across multiple levels.

It is especially convenient to carry out syntax and integrity checks when required in a given application. One can establish rules as to which node classes can be subordinate to which other node classes, set limits on attribute values,

check that every node is appended exactly once (no cycles), and so forth. Integrity can be verified by computing a hash over the tree. If data nodes are included in the hash, it can be used to detect new data to be uploaded [16]. We list here some possible workflow steps that may be useful and practical in this technique. Once the nodes have been instantiated and composed into a tree structure, the remaining methods can be invoked in any order by recursive descent against the whole tree or any subtree within it, thus establishing a life cycle for the command tree:

- Instantiate – construct or allocate nodes needed for the initial tree,
- Compose – establish parent/child relationships between all nodes, extending in a branching structure from a single root node,
- Initialize – set default values for node attributes, realizing the canonical semantic model of the program,
- Summarize – project the command tree to obtain node count, memory usage, and other metrics,
- Serialize – project the command tree to render it as structured text,
- Verify – traverse the tree while checking syntax and verifying integrity, according to the needs of the application,
- Execute – carry out a program of interactions with hardware and communications, transforming the tree as node attributes change and data nodes are appended,
- Analyze – gather and summarize data found in subordinate data nodes, transforming the tree as analysis nodes are appended,
- Report – project the tree to render formatted reports of gathered data and analysis nodes,
- Release – deallocate or free memory used for the command tree and all subordinate nodes.

9 Deserialization

There are several features we take for granted in established programming languages that might not be needed in a small bespoke language. One is deserialization, used to parse a structured text representation into nodes or tokens in memory. Another is the ability to parse and execute general math and logic expressions. Experience shows that these features impose a burden of time and expense to implement, document, and verify. The techniques described here give one the choice of when and how to implement additional features, according to the needs of the project.

There are several use cases that justify the need for deserialization. In our view the most important of these is testability, to allow for round-trip testing. We propose here that since serialization is so much easier than deserialization, serialization should be implemented and verified first to use as a trusted reference. Which in turn makes the deserializer, that is the source code parser, easier

to implement and verify. Archived output from the serializer can be deserialized and then serialized again. With the new code being compared to the archived code for equality. This will be discussed more in section 11.

Because the deserializer isn't needed to create and execute AST command trees, it can be implemented class by class, as and when needed. The way this works is that we begin with all source code nodes deserializing to the default base classes for leaf and branch. Then we add specialized implementation for each node class in turn. The deserializer is typically a factory method that instantiates and composes the nodes encountered while parsing source text. As the deserializer starts working, we can begin to add test cases that rely on it. Specifically, we can have tests where instead of executing the tree on live hardware to populate it with data nodes, we can deserialize an archived command tree that already has data nodes attached. Thus removing the dependence on live hardware for all steps in the workflow that come after execution. This principle of *Separation of Execution and Analysis* opens up many possibilities for testing on-target and off-target, and for verifying with test cases obtained directly from archived live data, live data modified to stress limits and edge cases, synthetic data meant to prove mathematical robustness, or hybrid combinations of these sources.

A secondary use of deserialization is to move data analysis and reporting off-target to host software running on a workstation or in the cloud. Here too one only needs partial deserialization to support analysis and reporting, leaving hardware-dependent nodes as pass-through base class nodes. This is especially helpful in applications with multiple stages of analysis, or multiple report formats. The state of the command tree can be serialized at each stage for use in testing the following stages.

As a practical matter, experience shows that implementation of deserialization is best accomplished in two steps. First create a means for deserializing structured text such as a JSON file into a composite tree structure in memory, and providing a serialization method to allow round-trip verification of the structured text parser. Then create a separate means for rendering the composite tree structure in memory into an interpreter command tree. The benefit in this approach is that the structured text parser can enforce all of the rules of the text representation, while not needing any domain-specific knowledge defined by the application. The rendering component can then be assured of receiving valid inputs, while enforcing domain-specific rules and requirements. Now we can use the previously verified command tree serialization method to obtain a structured text representation to compare with the deserializer input.

10 Math and logic expressions

The work described here has its origins in the logic scan of industrial programmable logic controllers [17, 18]. Having said that, we will only discuss math and logic expressions in an overview. Evaluation of such expressions by recursive descent implies subclasses of nodes that return math and logic values when executed. Leaf or terminal nodes would return a value obtained from hardware such as analog and digital inputs, or a constant literal value. Leaf nodes could also return values from internal registers as might be used in communicating with other subsystems. Branch or composite node classes would be

provided that return values when executed for the operations of Boolean logic, and for those math operations needed for the application.

The ability to parse general math and logic expressions is probably not a requirement for most embedded interpreters, especially in domain-specific applications where needed computations can be coded into pre-defined node classes. If however the need for mutable math and logic expressions can't be avoided, we would advocate for adding new execute methods to math and logic leaf nodes that return the needed values. Similarly branch nodes that need those return values can invoke the new execute methods of their subordinates to obtain them. This is clearly a step change in complexity for implementation, verification, and documentation.

11 Asymmetric tree comparison

Performing a directional comparison is important when comparing serialized test outputs with reference files, so that the test files can include additional comments, annotation, and instrumentation. Additionally, items you don't want to compare (timestamps, sequence numbers that change often, analog values such as voltage or temperature, etc.) can be ignored in the comparison by removing them from the reference file. The test file can then have varying values for these items and still be considered as equal, in order to pass a regression test for instance. If you actually need an exact match in both directions, simply test the files twice, reversing their order so both are evaluated as test files. If both comparisons are True, then both files have exactly the same dictionary keys and array elements. Following is a list of rules used in this work to compare test outputs with reference files:

- Any dictionary in the test file must have all of the keys in the corresponding dictionary in the reference file. Order is ignored. Dictionaries in the files being compared can have a completely different ordering, and still test as equal. Additional keys in the test file are ignored.
- Any array in the test file must have at least as many elements as the corresponding array in the reference file. Order is maintained and must match, up to the last element in the reference file. Additional array elements in the test file are ignored.
- Each value in a dictionary in the reference file must be equal to the corresponding value in the test file, according to these rules.
- Each element in an array in the reference file must be equal to the corresponding element in the test file, according to these rules.
- Integers must have the same value and sign.
- Floats must have the same value, sign, and exponent. Floats with the value Infinity in the reference file must have a corresponding Infinity in the test file. Sign is not considered in this comparison. Floats with the value NaN in the reference file must have a corresponding NaN in the test file.

- An optional argument can be used to set an error delta. If used, float values will be considered as equal if the absolute value of their difference is less than the error delta value.
- Booleans must either be both True, or both False.
- Text strings must be an exact case-sensitive match, including white space and escape characters inside the string. White space outside of text strings is ignored. The two files being compared can have a completely different formatting in terms of white space, and still test as equal.
- Null values in the reference file must have a corresponding Null value in the test file.

Reference files should be under version control, ideally alongside the implementation source code. Test files can also be versioned, especially if they contain additional content not in the reference files. This style of testing encourages the use of large scale bulk tests, which may seem unfamiliar to those accustomed to long lists of fine-grained assertions. In either case the workflow is the same: find all the deltas, rebaseline those that are expected, and repair those that aren't. One might ask where the reference files come from. Experience shows that a variety of sources can be used including outputs from previous versions, synthetic data created programmatically, hand typed data, or hybrid combinations of these. The content of this section has been published previously in reference [19].

12 Execution model

Much of the work reported here was done on targets that support a real-time operating system (RTOS). Since the Springleik command tree carries along with it all of the arguments and parameters needed for execution, a task thread to execute the tree can be started by simply passing a pointer or reference to the root node into the thread create function. On exit the pointer or reference will identify a tree now having updated state information in its attributes, along with any instrumentation and data nodes appended during execution. Having a unique pointer or reference allows the developer to think clearly about the life cycle of each command tree. By limiting access to hardware and global data to specific commands or node classes, it becomes straightforward to implement synchronization primitives to maintain thread safety between command trees. Springleik command trees easily support cooperative multitasking by inserting appropriate thread waits. While one could introduce a node class to do this, it's easier to put waits into branch node execute methods to relinquish control between calls to subordinate nodes. Memory used for each command tree can be released or otherwise disposed of recursively when no longer needed.

13 Reuse and granularity

One may well wonder how much of this can be reused across multiple projects. Our experience has shown that command nodes are clearly domain specific and

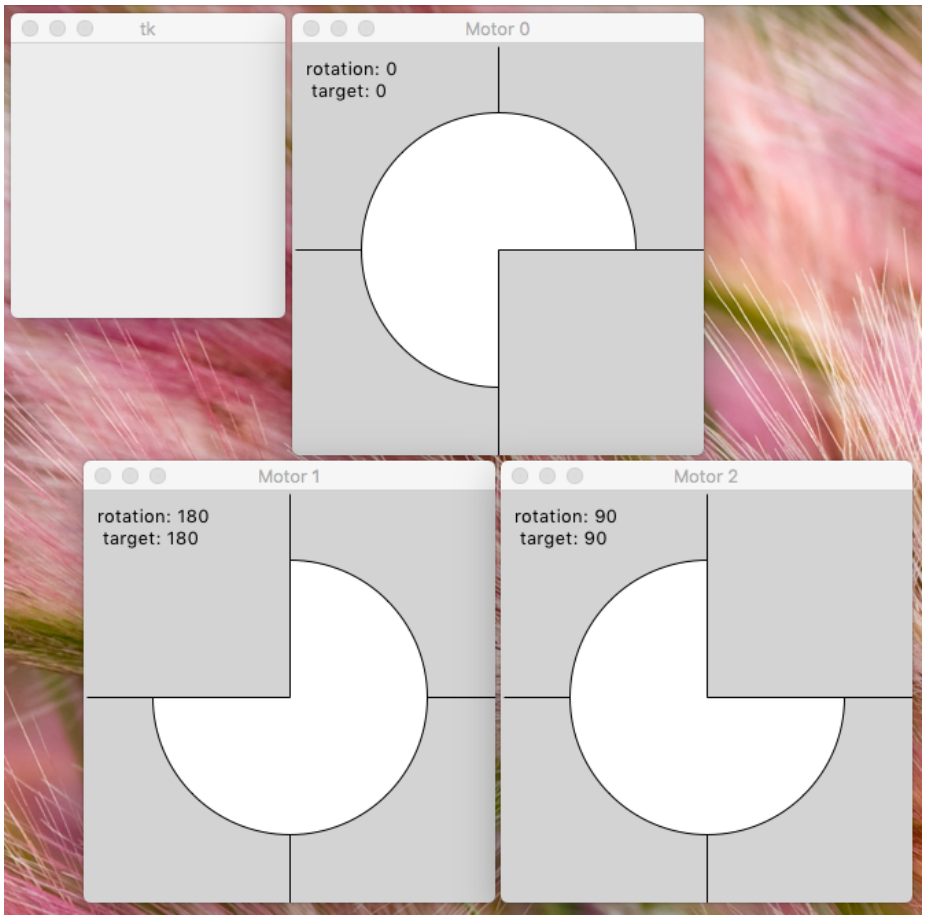


Figure 5: Three axis motion control simulation implemented in a target process. A separate host process implements an AST interpreter using Springleik techniques. Source code is posted in the Springleik project release on Zenodo [11].

can only be reused for very similar applications, while control nodes are not domain specific and can be reused freely. The resulting plethora of small languages can be justified by our need to maintain a small footprint while optimizing the command tree language for each application. A related question is how to set the granularity of command and control nodes. We propose starting with one command node class for each monitor command on the target, and one control node class for each branching or gating operation. If math and logic expressions are to be supported, start with one math or logic operation per control node. As development progresses common patterns will soon become apparent, which then become candidates for integration into more functional node classes. These are design decisions that don't have to be decided up front.

14 Host target demo

The Springleik release on Zenodo [11] includes a simulation model of a three axis motion controller, written in Python, that implements a target process (in file *target.py*) and a host process (in file *host.py*). These programs communicate with each other through a TCP socket port. As described in the subproject's *ReadMe* file, both processes expose a command line console. The target process command line accepts commands related to the simulated hardware. The host process command line accepts commands related to the AST interpreter. Any commands the host doesn't recognize are passed along to the connected target process.

The target process uses Tkinter to render the animated shaft position indicators shown in figure 5. The host process includes a command tree factory method that can instantiate and compose the structures given here in figures 2 and 3. In addition to the interactive command lines, the target and host programs also accept script files containing commands and arguments appropriate for each. Script files read by the target program contain target commands that interact with simulated hardware, while script files read by the host program contain host commands that interact with the AST interpreter. The subproject's *ReadMe* file gives examples of this.

15 Conclusion

We have developed a workflow for embedded DSLs that puts abstract syntax first, and postpones concrete syntax until actually needed in the application. In this workflow the AST is the primary artifact, which can be transformed and projected in various ways to meet the needs of the application. The reason this is so efficient is that serialization is much easier to implement, document, and verify than deserialization. Serializing the AST by recursive descent gives a complete picture of the DSL program, while at the same time creating a concrete syntax to be used as a reference when deserialization is needed. A closed-loop feedback model of software development shows how an AST program can be realized without source code. When a DSL program is constructed using abstract syntax, it can then be ported to new hardware or a new host language easier than would be the case if using concrete syntax.

We have defined a command tree architecture and life cycle based on a verb-oriented object model, organized around things the target does rather than things the target has. Which results in a malleable AST structure offering excellent support for ongoing software development and maintenance. Non-executing data and analysis nodes can be appended to the tree as needed, to support instrumentation, verification, analysis, and reporting of run-time results. By explicitly separating execute and analyze methods for traversing the tree, we introduce the principle of "Separation of Execution and Analysis." Which overcomes the need for live hardware when developing and verifying analysis and reporting code in the host language.

One can organize multiple DSL programs as a simple list of pointers or references to their root nodes. Command tree life cycles can begin, end, and overlap when and as needed by the application. Each step in the life cycle can be documented and archived by serializing the AST. Regression tests are done

by comparing new outputs with archived versions. While a proof-of-concept can be done with typical file comparison tools, production testing should done using the asymmetric tree comparisons described herein. Aside from this tree comparison tool, no other proprietary software or tooling is needed to investigate and implement a DSL based on a Springleik AST.

A GitHub repository for the Springleik project has been posted on Zenodo [4]. It contains example code for AST interpreters in Python, Java, and C language. The Python folder includes a demonstration program that simulates a three-axis motion controller, incorporating many of the concepts introduced here. Our experience across multiple projects, from new development to legacy upgrades, has encouraged us to continue the development and application of Springleik AST interpreters.

References

[1] J. Hage, "Bookreview JFP: Domain-Specific Languages by Martin Fowler The Addison Wesley Signature Series," Journal of Functional Programming. 2012 May;22(3):375-7.
https://doi.org/10.1017/S0956796812000123

[2] J. Clements, K. Fisler. "Little language project modules," Journal of Functional Programming. 2010 Jan;20(1):3-18.
https://doi.org/10.1017/S0956796809990281

[3] M. Voelter, J. Siegmund, T. Berger, B. Kolb, "Towards user-friendly projectional editors," in International Conference on Software Language Engineering 2014 Sep 15 (pp. 41-61). Cham: Springer International Publishing.
https://doi.org/10.1007/978-3-319-11245-9_3

[4] M. Williamsen, "Abstract Syntax Tree (AST) interpreters in Python, Java, and C-Language," Springleik software release 1.0.1, Zenodo, 2026.
https://doi.org/10.5281/zenodo.23595064

[5] C. Simonyi, "The death of computer languages, the birth of intentional programming," in NATO Science Committee Conference 1995 Sep (pp. 17-18). Available as Microsoft Research Technical Report MSR-TR-95-52.

[6] M. van den Brand, "A personal retrospective on language workbenches," Software and Systems Modeling. 2023 Jun;22(3):847-50.
https://doi.org/10.1007/s10270-023-01101-9

[7] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R.Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, "The state of the art in language workbenches: Conclusions from the language workbench challenge," in International Conference on Software Language Engineering 2013 Oct 26 (pp. 197-217). Cham: Springer International Publishing.
https://doi.org/10.1007/978-3-319-02654-1_11

[8] C. L. Phillips and R. D. Harbor, "Feedback Control Systems," 3rd edn. (Prentice Hall 1996).

[9] I. D. Baxter and M. Mehlich, "Reverse engineering is reverse forward engineering," Proceedings of the Fourth Working Conference on Reverse Engineering, Amsterdam, Netherlands, 1997, pp. 104-113.
https://doi.org/10.1109/WCRE.1997.624581

[10] M. Fowler, "A Pedagogical Framework for Domain-Specific Languages," *IEEE Software*, vol. 26, no. 4, pp. 13-14, July-Aug. 2009.
https://doi.org/10.1109/MS.2009.85

[11] M. Williamsen, "Abstract Syntax Tree (AST) Interpreter for Host and Target," Springleik software release 1.0.1, Zenodo, 2026.
https://doi.org/10.5281/zenodo.23595064
See file Python/hostTarget/ReadMe.html.

[12] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, C. Wimmer, "Self-optimizing AST interpreters," in Proceedings of the 8th Symposium on Dynamic Languages 2012 Oct 22 (pp. 73-82).
https://doi.org/10.1145/2384577.2384587

[13] D. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, vol. 2, no. 2, June 1968.
https://doi.org/10.1007/BF01692511

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," (Addison-Wesley 1994).

[15] M. Williamsen, "Abstract Syntax Tree (AST) interpreter in C-Language," Springleik software release 1.0.1, Zenodo, 2026.
https://doi.org/10.5281/zenodo.23595064
See file C-Lang/ReadMe.html.

[16] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," Pomerance, C. (eds) Advances in Cryptology — CRYPTO '87. CRYPTO 1987. Lecture Notes in Computer Science, vol 293. Springer, Berlin, Heidelberg.
https://doi.org/10.1007/3-540-48184-2_32

[17] J. E. Reich, B. H. Krogh, I. D. Baxter, "Symbolic Simulation-Based Techniques for Debugging Discrete Control Programs," 13th IFAC World Congress, vol. J, pp. 413-418, 1996.
https://doi.org/10.1016/S1474-6670(17)58446-3

[18] M. Williamsen, "Compiler Test Case Generators," retrieved online 16 October 2026.
http://www.williamsonic.com/CompTree/index.html

[19] M. Williamsen, "PompTree," *GitHub*, 12 October 2026.
https://github.com/springleik/PompTree

Biography



Mark Williamsen received a B.S. degree in physics and a Ph.D. degree in physics from the University of Wisconsin at Milwaukee in 2003 and 2010 respectively. He has previously worked at Rockwell Automation in Milwaukee and Quantum Design in San Diego, USA. He is currently employed at Coesia FlexLink in Gothenburg, Sweden working on embedded firmware applications for the industrial internet-of-things (IIoT).