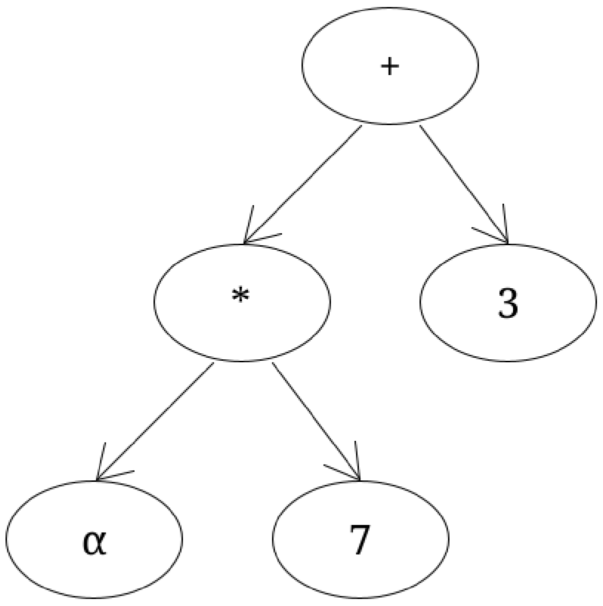
Software Engineering
Programming Languages

ulm university universität ulm



Provenance Tracking in Python for ROS

Alexander Kunz

Bachelor's Thesis

Institute of Software Engineering and Programming Languages

Alexander Kunz:
Provenance Tracking in Python for ROS

Advisor:

M.Sc. Thomas Witte
Institute of Software Engineering and Programming Languages

Examiner:

Prof. Dr. Matthias Tichy
Institute of Software Engineering and Programming Languages

Bachelor's Thesis, University of Ulm, 2023.

Abstract

Keeping track of the origin, history and data flow of values is becoming increasingly relevant in software design. This concept is known as provenance tracking and allows for self explainability features within computer programs. We describe the design and implementation of a Python library for tracking and modifying values based on their runtime history. This library allows users to enable provenance tracking for generic objects and their attribute structure by implementing a generic wrapper type, which makes integration relatively seamless with minimal changes to the source code required. Users can mark the literal origin position of values which we define as a location. By utilizing a location tree we keep track of object structures and source location information to offer transparent tracking of nested structures. This allows us to enable provenance tracking in the context of the Robot Operating System (ROS), which requires provenance to be shared accross multiple program nodes communicating using various message structures.

We keep the memory footprint low by reducing the provenance tree on the fly using specialized operator chaining rules and apply several strategies to effectively transmit the provenance information between program nodes. We supply a monitoring tool for provenance data visualization and demonstrate that it is possible to programmatically calculate the required input values to deterministic functions to return a specified target value. This type of backpropagation works by making use of the provenance tree by evaluating its inverse with the target value. While it is limited to functions in which the control flow does not depend on the input value, it works without code adjustments of the function body even if it is unknown and imported from a library.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

With the progressing complexity of cyber-physical systems in robot applications, self explainability is becoming an increasingly relevant topic for usage and development [BGCG+19]. Often, when the desired outcome of a program depends on source values, we need to know which and how values need to be changed to meet this outcome, which is not always a straightforward process. By tracking values and recording the history of all operations applied to them, we can specify the new desired value and calculate the required source parameter for a given outcome. This is achieved through reversing all operations applied over the course of the program execution, hence the term *provenance tracking* [AACP13]. It refers to the process of retaining the origin, history, and data flow of sequential operations such that they may be analyzed and processed in a reversible fashion. This simplifies basic workflows and rapid prototyping as parameters in the source code can be adjusted in real-time, with changes propagating backwards through the chain of operations. For instance, an interface can show the trajectory of a quadcopter in real-time and update the waypoint nodes as the user drags them around [WT19].

Robot Operating System (ROS) [QCG+09] is an operating system model which supports the development of robotic systems through a node-based communication layer. ROS itself does not support provenance tracking, however external libraries can be used to implement additional functionality. *rosslt* is a C++ library implementation of a provenance tracking system which can be used with ROS [Wit21]. It is demonstrated to support tracking and modification of variables within a single or across multiple ROS nodes. It works by supplying a value wrapper class that can store and handle operation history to provide expression reversing functionality. Most nodes used for ROS are written in either C++ or Python, as these are the officially supported languages. The *rosslt* C++ implementation is only supported on systems using C++ nodes, which limits the scope of its use. By implementing *rosslt* in Python, it is possible to perform provenance tracking on heterogenous systems.

In addition, Python can make the library easier to use, due to it being dynamically typed, and provides better introspection possibilities. The current C++ library needs to make use of macros to support structs such as basic ROS message types, because it does not provide the possibility to override the dot operator. A Python-based implementation reduces the syntactic overhead of these operations and allow for more deeply integrated tracking. This improves usability for developers and makes it easier to add the library to existing applications where extensive code rewrites may not be possible.

The performance of such a solution must be considered, because it directly affects the responsiveness and overall utility of the system. Basic calculations on tracked values need to be fast enough on the weak hardware often used in embedded robotic environments. This may present a challenge, as interpreted Python code can be slow [CGRL15].

The following work examines what can be done to provide a reasonable implementation using Python. First we address various issues which limit *rosslt*'s performance, then we measure how our adjustments affect system resource utilization in simulated workloads. We improve on both usability and performance of *rosslt* through reimplementation in an original Python codebase which is easier to integrate into existing ROS applications with little syntactic overhead. Practical usage on robot systems with low system resources is improved through performance optimizations. Finally we supply a monitoring tool for messages containing tracked data which shows both visualization and debugging is still possible with the addition of provenance information in our optimized binary format.

2. Background

In this section we are providing the background information necessary to understand the implementation, evaluation and ideas of the library. First we explain provenance tracking in general, then we are laying the groundwork for implementing the library by diving into provenance trees and the Robot Operating System.

2.1 Provenance Tracking

Protocolling the origin, history, and flow of data in a system is usually called *provenance tracking* [AACP13]. This is an abstract concept which can apply to any system in which sequential operations are applied to a given object. For instance, provenance tracking can be used with variables, sockets, files, and more. One example of a field where provenance tracking is used extensively is cloud computing [ZKK+12]. In the context of provenance tracking, we refer to the related term "Source Location Tracking" as tracking the *origin* or *source* of data throughout this work. Provenance tracking can also be used to inspect and visualize data flow and operations on data objects at run time. It also allows for controlled automatic changes to program input parameters or source code to adjust the program flow to reach a desired outcome, without requiring any form of static dataflow analysis [WT19]. This tracking often includes protocolling and respecting the order of transmission between processes and program nodes when used within distributed systems. We focus on provenance tracking of objects and their attribute values, which usually originate from a specific line in a given code file and are being modified during program runtime.

2.2 Provenance Tree

The history of a tracked value can be represented in a node based tree structure we call *provenance tree*. In contrast to an abstract syntax tree, it is built live during program runtime. It does not represent program flow, but rather a form of expression that defines what operators have been applied to the tracked value and in what order [AACP13]. The tree can be easily modified, visualized, and optimized. Consisting of both value and operator nodes, the former can be directly evaluated as their content, the latter requires to evaluate all subnodes to be able to calculate the desired result of its operator.

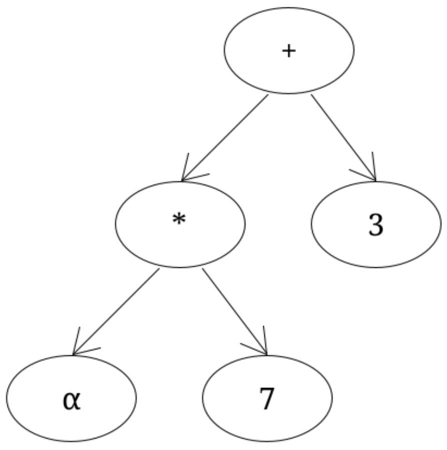


Figure 2.1: Example Provenance Tree

```
x = Location(5)
if x < 6:
    x = x * 7
x += 3
```

Listing 2.1: Example Provenance Pseudocode

By executing the `Location` function in the first line of Listing 2.1, we instantiate a wrapper around the value 5 with the source location set to the first line in the code file. When the program is done executing, we get the provenance tree Figure 2.1 as a result, represented by the expression $(\alpha * 7) + 3$. The expressions $(7 * \alpha) + 3$ and $3 + (\alpha * 7)$ represent the same tree because of the commutativeness of the used operators. As such, provenance trees can often be represented by multiple expressions. The above code results in the variable `x` evaluating to the value of $5 * 7 + 3 = 38$. We now introduce the concept of *forcing* values. This means to make the variable referencing the tracked value evaluate to a desired different value at program re-execution. In order to do that, we need to determine an inverse of the provenance tree attached to our location first.

We can recursively parse the tree to find an inverse of the history. This works as long as the respective operators in the tree have an inverse. Figure 2.2 shows a possible inverse representation of Figure 2.1.

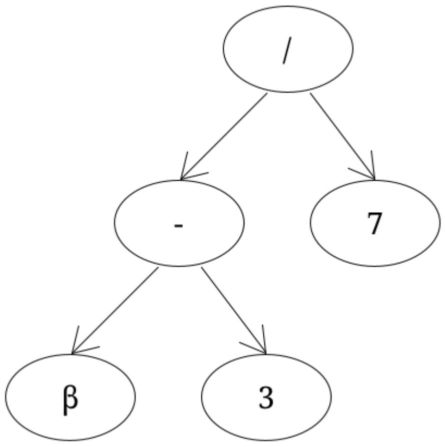


Figure 2.2: Inverse Provenance Tree

By evaluating this inverted provenance tree with the desired target value represented as β, we can calculate the required starting value. For example, if the desired value was to be 31, the inverted tree would evaluate to $(31 - 3)/7 = 4$. By making use of source location information, the origin of the value marked as location in Listing 2.1 at line 1 can be updated programmatically to this new value. Executing the pseudocode again with the location updated to 4, the variable `x` would evaluate to our desired result $4 * 7 + 3 = 31$. Throughout this work, we refer to this method as *backpropagation* [1] in the context of provenance tracking.

However, if we were to force the value to 45, the calculated starting value would be $(45 - 3)/7 = 6$, resulting in a change of program control flow. Executing the code with this new starting value, the variable `x` would not be less than 6, and therefore not multiplied by 7. This would result in `x` evaluating to 9 at the end of code execution, which is not equal to our desired target value 45. We note, that the provenance tree does not represent nor respect the control flow of the program, as it is built on program runtime.

In our work, we ignore the combination of multiple tracked values in the same provenance tree by always interpreting the right hand side of a given operator as raw value without tracking information, in the case that both sides consist of tracked values. This is done to simplify situations that occour when multiple solutions for forcing a value would be possible, or a given literal is reachable through multiple paths when inversing a provenance tree (also see aliasing problem [WT19]).

[1]This does not to refer to the term *backpropagation* as used in machine learning [Roj96]

2.3 Robot Operating System

ROS, or Robot Operating System, is an open-source framework designed to provide developers with tools and libraries to create complex robotic systems. It is not a traditional operating system, but provides a structured communication layer above the host operating systems of a heterogenous compute cluster [QCG+09]. Applications making use of ROS are usually split between several processes and hosts, which are independent of each other by making use of the node system that ROS provides [MSCG23]. In its second iteration called “ROS 2” [MFG+22], the main programming languages supported are C++ and Python, which means that nodes are usually using either language to implement their program logic.

Nodes communicate using publishers and subscribers of messages, which are bound to a node instance when initialized. Publishers are used to post messages to specified topics, which are identified by a string path. Subscribers can attach themselves to the topics and receive posted messages in a manner that is defined by their configuration.

Topics are usually used for a single message type, which is structurally defined using the Interface Definition Language (IDL) that ROS 2 provides [Tho20], which is a subset of the OMGL IDL 4.2 specification [ISO20]. These definitions are limited to a set of predefined native data types, including integer types such as `int16`, `int32`, `uint64`, floating point types like `float`, `double`, `long double`, and two string types: `string`, `wstring`. To define sets of multiple values, these types can be used to define arrays `T[N]` and vectors `sequence<T>`.

We can also define custom type structures, which can be imported and referenced in ROS IDL files, however there can not be any recursion. This means that structures like trees can not be defined as linked object graphs and thus have to be serialized to a compatible data structure before being able to be stored in a ROS message.

```
int32 id
uint16 node
string name
rosslt_py_msgs/Expression expr
```

Listing 2.2: Location Structure in IDL

On compilation, these IDL files are being used to generate language specific type implementations, which the programs can be linked against. This allows for flexible definitions shared among the code of the node cluster using one or more languages supported by ROS. However, it is not feasible to make any manual adjustments to the generated code, which means that adding custom functions or attributes to the class definitions is not being done in practice, and developers are being limited to what is available using ROS IDL alone when designing message types.

There is extensive tooling available for ROS applications to analyze nodes, messages, and events on live instances. The command line application *rostopic* [Ros23] can be used to monitor topics for raw message data logging. Visualization of positional vectors, paths, shapes, and interactive markers can be done in "RViz" [HGF22], which makes it easier to inspect live data in robotic applications. The program *rqt* [rqt23] is able to combine multiple tools into a single combined custom application specific guided user interface using bundled plugins. Using its node explorer, we can generate a graph visualizing the node cluster structure of ROS applications, as seen in Figure 2.3. Each of the arrows represent a subscription of a node to a given topic, which is specified by its identifying name above the arrow.

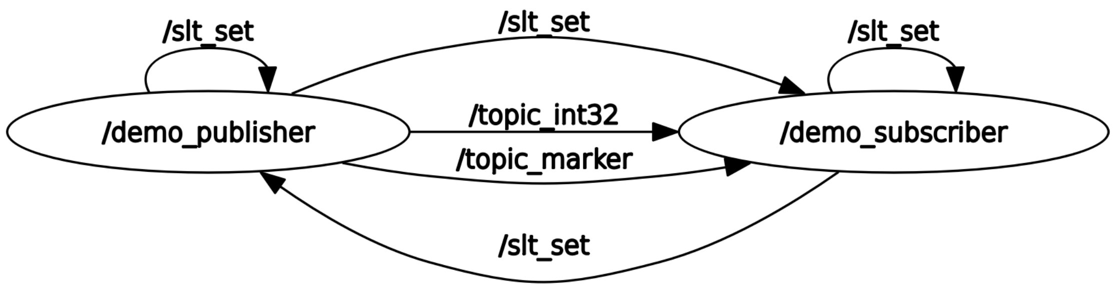


Figure 2.3: Example Nodes in ROS for Communication

3. Related Work

In this chapter we compare our solution with related work incorporating forms of provenance tracking of values within programming languages.

3.1 C++ Library "rosslt"

The *rosslt* library is made to provide provenance tracking functionality for C++ within ROS context [WT21]. It's name is an abbreviation of "Robot Operating System Source Location Tracking". By making use of a generic value wrapper, it is able to provide provenance tracking of attributes contained in generic structures. This allows for keeping track of object histories over multiple ROS nodes, as the library supports ROS message structures and transmission of provenance information over customized messages.

```
auto message = Tracked<visualization_msgs::msg::Marker>();
auto pose = GET_FIELD(message, pose);
auto position = GET_FIELD(pose, position);

SET_FIELD(position, x, loc(1.5));
SET_FIELD(position, y, loc(2.5));
SET_FIELD(position, z, loc(3.0));

SET_FIELD(position, x, GET_FIELD(position, x) + 1);
SET_FIELD(position, y, (GET_FIELD(position, y) * 2) / 3);
SET_FIELD(position, z, 5 - (GET_FIELD(position, z) / 4));

auto m = static_cast<rosslt_msgs::msg::PoseTracked>(pose);
publisher->publish(m);
```

Listing 3.1: Example Code using "rosslt"

By using `loc(value)` as seen in Listing 3.1, a location with its own identifier number will be allocated to the ROS node that refers to the code file position the function was called from. To store the provenance information for each attribute, a hashmap is used to map the attribute position within the class structure (e.g. `"/pose/position/x"`) to a location object, which stores the node name and location identifier the attribute originated from. This `Location` class instance also stores the current provenance tree of the attribute in form of an *expression* string, which stores the operands and operators as tokens similar to reverse polish notation (postfix notation) [Ham62]. An advantage of the format of the expression string is that the tokens can be evaluated from left to right, putting all values on a temporary stack and executing the operators with the top of the stack as operands. The `swap` token inverts the order of the stack and is used to correctly represent left and right hand side operators of tracked values. For example, `5 - (x / 4)` will result in the expression string `"4;/;swap;5;-"` as seen in Listing 3.2, representing the location map of the message object in Listing 3.1.

```
std::unordered_map<std::string, Location> {
  {"/pose/position/x", {node, id, expr="1;+"}},
  {"/pose/position/y": {node, id, expr="2;*;3;/"}},
  {"/pose/position/z": {node, id, expr="4;/;swap;5;-"}},
}
```

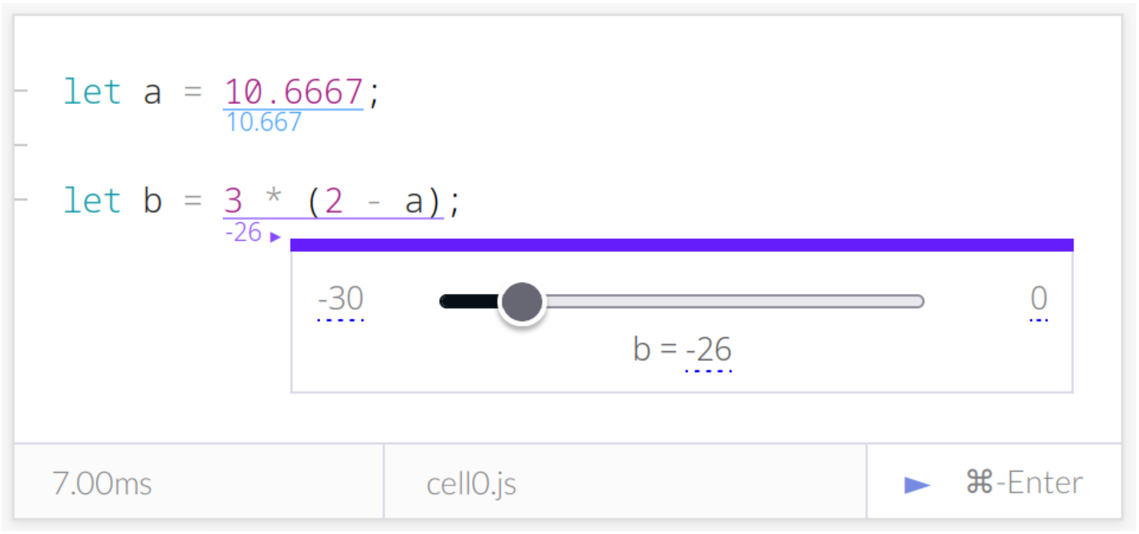
Listing 3.2: Location Map Pseudocode

The Python library *rosslt_py* made in the context of this work is a complete reimplementation with several different design choices made to improve on the C++ rosslt library. By making use of dynamic type introspection features available in Python, we are able to provide an improved architecture allowing for much easier usage and integration into existing projects.

While the rosslt C++ library already makes tracking features accessible to ROS nodes, the required usage of macros forces application developers to make more code adjustments than desired. Additionally, while the current usage of comma separated string representations of operator history makes it possible to easily visualize and debug tracking messages using ROS native tools, the required string parsing and processing introduces quite some overhead that can be a problem within applications on embedded systems with constrained system resources. This work attempts to solve these problems by using dynamic typing features available in Python to try reducing the required code changes for integrating value tracking to a minimum and provide a stable framework which makes the library more usable in practice. The performance adjustments help making data flow tracking for automatic source change on variable adjustments viable within interpreted program code in ROS Python nodes.

3.2 Carbide Alpha

Carbide Alpha is an experimental browser-based javascript programming environment which allows the user to change interim results of expressions or functions within the code with automatically generated widgets, while the programming viroment itself updates the assigned value of related variables or function parameters within the code automatically. Carbide Alpha calls this feature *backpropagation*. Carbide Alpha achieves backpropagation through an implementation using NumericJS's uncmin function [Loi23], containing a generalization of Newton's root-finding method [HP76] for multiple parameters. By evaluating a given function multiple times with slight parameter modification, an approximation is attempted to be found. When complete, this approximated result can then be used to substitute the assigned values of the related variables in the source code automatically [Car23].



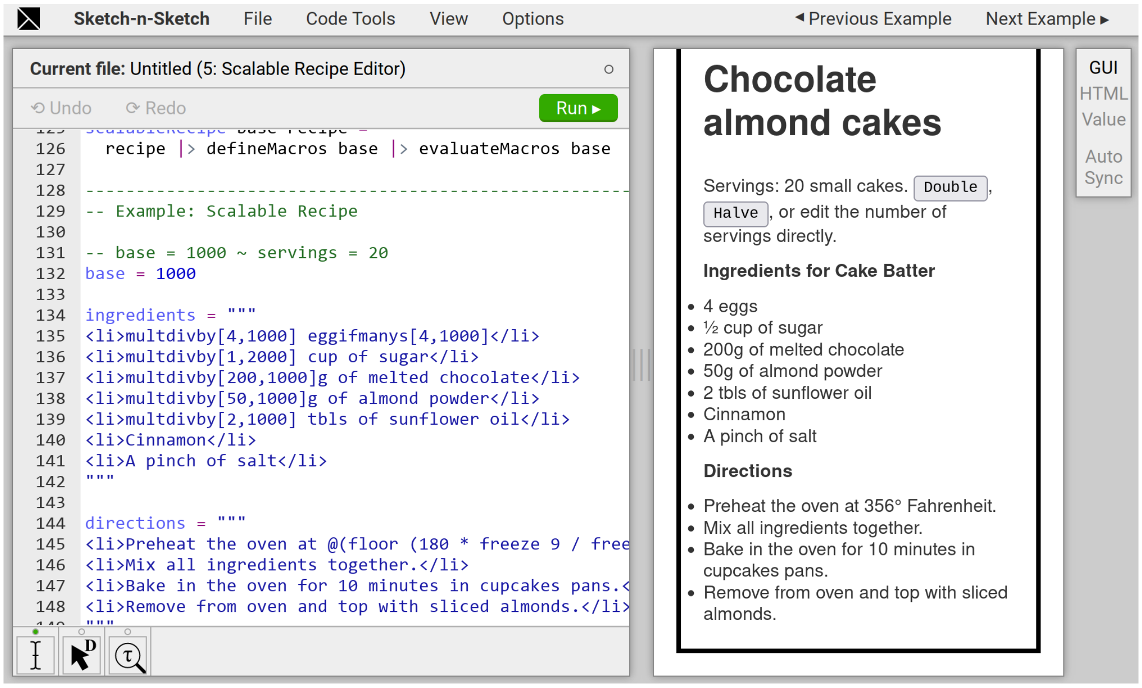
```
let a = 10.6667;
        10.667

let b = 3 * (2 - a);
        -26
```

Figure 3.1: Carbide Alpha

While Carbide Alpha involves provenance tracking to achieve backpropagation, it does this in an algorithm similar to limited-memory BFGS (L-BFGS) [LN89]. In contrast, rosslt uses the inverse of the expression string of its generic value wrapper to calculate the desired result instead of approximation. Carbide Alpha supports backpropagation even when multiple parameters are involved in the process [Car23], however it is not always successful in finding the correct values in its approximation process.

3.3 Sketch-n-Sketch

Sketch-n-Sketch is a direct manipulation programming system that allows the user to change the generated output of a given code in a corresponding view [HLC19]. This works in multiple ways, for example by editing output text like shown in Figure 3.2 or by manipulating scalable vector graphics on a two-dimensional plane inside the output view. Varibles that are directly responsible for the modified output get updated accordingly automatically. Sketch-n-Sketch works by utilizing provenance tracking to backpropagate changes to the source code when objects in the output view are manipulated by the user.



```
  recipe |> defineMacros base |> evaluateMacros base

--------------------------------------------------------
-- Example: Scalable Recipe

-- base = 1000 ~ servings = 20
base = 1000

ingredients = """
<li>multdivby[4,1000] eggifmanys[4,1000]</li>
<li>multdivby[1,2000] cup of sugar</li>
<li>multdivby[200,1000]g of melted chocolate</li>
<li>multdivby[50,1000]g of almond powder</li>
<li>multdivby[2,1000] tbls of sunflower oil</li>
<li>Cinnamon</li>
<li>A pinch of salt</li>
"""

directions = """
<li>Preheat the oven at @(floor (180 * freeze 9 / free
<li>Mix all ingredients together.</li>
<li>Bake in the oven for 10 minutes in cupcakes pans.<
<li>Remove from oven and top with sliced almonds.</li>
```

Chocolate almond cakes

Servings: 20 small cakes. `Double`, `Halve`, or edit the number of servings directly.

Ingredients for Cake Batter

- 4 eggs
- ½ cup of sugar
- 200g of melted chocolate
- 50g of almond powder
- 2 tbls of sunflower oil
- Cinnamon
- A pinch of salt

Directions

- Preheat the oven at 356° Fahrenheit.
- Mix all ingredients together.
- Bake in the oven for 10 minutes in cupcakes pans.
- Remove from oven and top with sliced almonds.

Figure 3.2: Sketch-n-Sketch Editor

To implement this functionality, Sketch-n-Sketch introduces an “almost superset” [Ske23] of the Elm syntax [Cza12] and relies on an algebra system called *REDUCE* [Fit85] to solve multi-equation systems which it communicates with. In contrast to that, rosslt can be imported as a standalone library into existing programs and is designed for provenance tracking with ROS in mind.

4. Problem Statement

As Python is a dynamically typed language which provides more advanced introspection possibilities than C++, it may make it easier for developers to integrate provenance tracking into existing projects and reduce the necessary code adjustments. *rosslt* requires inconvenient macros to be used to correctly work with common data structures, which hinders adoption for developers. In similar scenarios, commonly-used built-in operator functions can be overriden in Python, which can greatly improve usability. For instance, we can override the *getattr* operator to provide a tracked wrapper to an attribute of the class when it is being queried by use of the dot operator on its instance. This child wrapper has to update the data in the location map of the parent tracking wrapper for the history data to be valid. Using a custom tree based location map, the child wrappers can directly reference the child nodes of the tree and the data can be up to date all the time while aiming for as little overhead as possible (RQ1).

A big drawback of the current C++ implementation is that the history is being recorded in an inefficient and big comma separated string representation. This results in countless conversions between strings and numbers, memory copy operations because the used strings objects are immutable, and string split operations to access and process the elements contained (RQ2).

In addition, *rosslt*'s location map system has some flaws. The paths to all members of a tracked structure are absolute, which consumes extraneous memory, besides being inperformant to work with. Employing an object oriented tree structure for keeping track of location information should theoretically solve these problems while also being more efficient (RQ3).

The expression history of tracked values can grow large, which can be a problem during transmission of provenance data within ROS messages. We attempt to implement a new serialized format for provenance data that can be used with the message structure features ROS provides, while trying to not rely on string representations like rosslt does. Compression can also be used to reduce system memory usage. Each message begins with a header which may contain a lot of redundant data, especially for big tracked structures where multiple values can share the same history. These cases would likely benefit from the use of compression, while for smaller data sets the resulting processing overhead could lead to worse execution time performance. Compression only being enabled for large headers could provide a good middle ground, however a practical threshold must be found for this to be useful (RQ4).

The command line tools supplied with ROS can be used to monitor messages and debug programs. However, the serialized data needs to be decoded as it is not human-readable in contrast to the old string representations of location map paths and expression history. For this use case, a tool for decoding and displaying the messages containing the tracking data needs to be created. Integrating a simple guided user interface besides basic logging could help easily visualizing the data that gets exchanged between the application nodes (RQ5).

In practice, the better the library performs, the more values can be tracked without impairing the performance. It is of general interest how the redesigned rosslt library performs with the design modifications taken (RQ6), even though Python interpreters usually come with various disadvantages over compiled code with optimizations [CGRL15].

RQ 1: How can alternative approaches be employed for extracting nested attributes from tracked objects?

To achieve provenance tracking for data structures in Python with minimal changes to the source code required, we search for a tracking solution supporting nested attributes that does not modify the underlying object structure.

RQ 2: Which data structures are suitable for the provenance tree?

We try to improve on the rosslt C++ implementation, which is using a comma separated string to store the provenance tree in postfix notation. We explore how the provenance tree can be stored to suit our Python library implementation.

RQ 3: How can the source location information be stored for nested attributes?

Using our tracking solution of RQ1, we try to find an alternative approach to storing source location information for each attribute in nested data structures.

RQ 4: Which improvements can be made to reduce the message overhead of the provenance data?

As the provenance data needds to be stored in the ROS message structure to be propagated to other program nodes, it is of advantage to store it as efficiently as possible. We research what can be done to minimize memory overhead by changing the storage format of the provenance tree.

RQ 5: How can live tracking data be decoded and displayed?

To answer this question we develop a monitoring tool integrating our library, which makes it possible to display live provenance data in order to show, that the non human-readable format of our message serialization is still usable for inspecting messages on the fly.

RQ 6: How do the library adjustments impact the performance and usability?

For this research question we analyze how our library design decisions and adjustments impact runtime execution time of the tracking wrapper and look at the memory overhead of the serialized provenance data propagated through the ROS message structure.

5. Implementation

To provide provenance tracking in Python, we implement a library that can be easily imported into existing projects. There are several requirements to approach the goal of having to adjust as few code as possible, while being compatible with typical ROS applications.

In this chapter we are explaining the implementation and decisions of the creation of the Python library *rosslt_py* providing provenance tracking features for ROS 2. We are discussing various aspects, such as data structures and algorithms, which are chosen for achieving effective provenance tracking within the ROS framework while showing code examples for utilization and implementation.

5.1 Location

We refer to an abstract data origin as a location. To differentiate between source locations of provenance information, such as in which code file and line the starting value was set, we map this meta information to a dynamic set of location identifiers specific to a node. Referencing the data origin works by supplying a pair of node name and location identifier. This identifier can be generated by calling the `location` helper function supplied by our library for nodes incorporating tracking, which automatically receives the code file and line of the calling source location from the stack data at runtime using the python integrated *traceback* module. As of Python 3.11, it is not possible to get the column offset in the code line of the calling location by using the stack data, because the interpreter currently only keeps a mapping of bytecode to line numbers in the compilation process [GTA21].

5.2 Operators

We supply a broad range of commonly used operators that can be seen in Table 5.1. These target the generic supported numerical operations available to class instances in Python, though additional custom operators unavailable to override can be added as part of future work, as shown by the implementation of `sin/asin` and `cos/acos`. The groups are being used to determine if a set of two operators with their arguments can be chained together, which is required in Section 5.6. If an operation involves an operator that is not commutative (`Comm.` attribute), the SWAP operator can be used to invert the argument order of a node in the provenance tree. The argument and result counts are used to evaluate the provenance tree in postfix form as a variable number of operands are pushed and popped from the temporary stack depending on the operator. These operators are implemented as static object instances and can be added as lightweight reference in a variable type array to store the provenance tree in postfix notation (RQ2).

ID	Operator	Identifier	Attributes	Args	Results	Group	Inverse
0	SWAP	swap	-	2	2	0	SWAP
1	ADD	+	Comm.	2	1	1	SUB
2	SUB	-	Negate[1]	2	1	1	ADD
3	MUL_INT	*	Comm.	2	1	0	DIV_FLOOR
4	MUL	*	Comm.	2	1	2	DIV
5	DIV	/	-	2	1	2	MUL
6	DIV_FLOOR	//	-	2	1	0	MUL_INT
7	SIN	sin	Comm.	1	1	0	ASIN
8	COS	cos	Comm.	1	1	0	ACOS
9	ASIN	asin	Comm.	1	1	0	SIN
10	ACOS	acos	Comm.	1	1	0	COS
11	POW	pow	-	2	1	0	IPOW
12	IPOW	ipow	-	2	1	0	POW

Table 5.1: Implemented Operators

[1]Used for a special case in the chaining algorithm seen in Listing A.1

5.3 Location Tree

The library has to keep track of each attribute in a wrapped object containing data that is being modified by the program, in order to keep the provenance information. As the attributes can not hold this information themselves, because they can not be stored as wrapped objects, there needs to be a mapping of attribute name to provenance information, which the wrapped root object holds. The original rosslt C++ library solves this by using a hashmap of path string to location object.

This approach has multiple problems. It is inefficient to parse and work with, especially if wrapped attributes a few layers deep have to alter it. Additionally it is not memory efficient, as it contains duplicate path prefixes which get longer the deeper the object tree is.

We get rid of the hashmap and work with a tree graph representing the original object structure. Using this method we can avoid using paths altogether. Instead we can directly use standard methods to create, traverse, and modify trees in an object-oriented manner while still being able to reference partial trees without copying node contents. (RQ3).

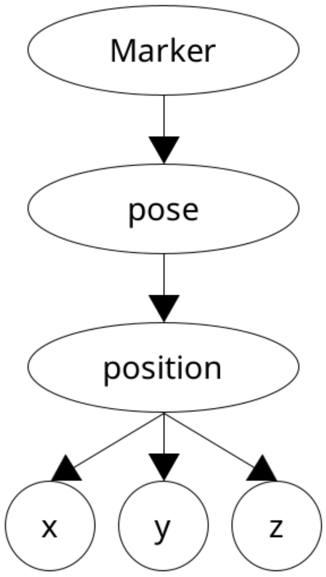


Figure 5.1: Location Tree Node Graph

5.4 Library Design

The only way for a pure Python library to keep track of operators being applied to variables without adjusting the source code is to create a wrapper class that overrides all possible operators. This way the wrapper can get notified on operator execution to append both operator and operands to its provenance tree.

Using this wrapper implemented in our library, we can also programmatically calculate the required parameter to a function, for it to return a desired value using the functionality and restrictions of the backpropagation method explained in Section 2.2. This does not require any modifications to this function so it works even for unknown functions imported from other Python libraries. We provide an example of this in Listing A.2 using an artificial deterministic "blackbox" function, of which the control flow does not depend on the parameter.

Messages in ROS are hierarchies of class instances containing data in attributes that are typically directly assigned.

```python
# create message
marker_value = self.location(Marker())

# set initial values
marker_value.pose.position.x = 1.5
marker_value.pose.position.y = 2.5
marker_value.pose.position.z = 3.0

# apply operations
marker_value.pose.position.x += 1
marker_value.pose.position.y *= 2
marker_value.pose.position.z /= 3

# publish message
self.publish(self.pub_marker, marker_value)
```

Listing 5.1: Example ROS Message Publisher

Additionally ROS generated assertions require the hierarchies to contain the original type, which means we can not assign wrapped message data (see Listing 5.2). We solve this by cloning the class hierarchy structure of the wrapped variable into a provenance tracked wrapper tree. Making use of `__getattr__` and `__setattr__`, we can build the tree on program runtime as needed.

It is possible using `__getitem__` and `__setitem__` to wrap both lists and dictionaries as list indices and dictionary keys. These can be used in place of the attribute name to reference nodes in the location tree of the wrapped object.

```
@pose.setter
def pose(self, value):
    if __debug__:
        from geometry_msgs.msg import Pose
        assert \
            isinstance(value, Pose), \
"The 'pose' field must be a sub message of type 'Pose'"
    self._pose = value
```

Listing 5.2: Type Assertions in ROS Messages (Generated Code)

Using a wrapper, the type of the variable will be different to the original type. Specifically, this means that both `type()` and `isinstance()` will result in different behavior. We assume that most of the time it is not an issue in practice, but suggest to check against the unwrapped variable if required. In Listing 5.3, we demonstrate the type problem using the interactive Python shell. The evaluation result of code lines starting with `>>>` follows on the line below.

```
>>> val = Tracked(Marker())
>>> type(val) is Marker
False
>>> type(val) is Tracked
True
>>> type(val.unwrap()) is Marker
True
```

Listing 5.3: Wrapper Type Problem

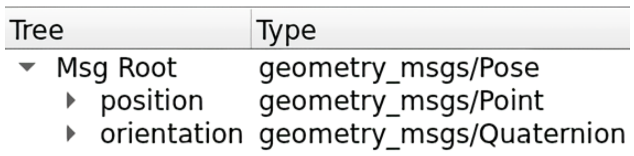
Trigonometric functions such as sin/asin are often required in robotic applications. While the integrated math functions such as `math.sin` can be made working by implementing `__float__` in our wrapper class, they are not able to retain tracking information of a variable, as they will return a floating point number without the required wrapper containing the provenance tree. When the result is directly assigned to the variable, all the tracking information is discarded. This behavior can not be altered without source modifications of the Python interpreter, however we provide custom functions in our wrapper class to provide tracking functionality for sin/cos/tan/asin/acos/atan.

```
>>> var = Tracked(1.0)
>>> var
Tracked(1.0, Location(Expression()))
>>> var = var.sin()
>>> var
Tracked(0.8414709848078965, Location(Expression(sin)))
>>> var = math.sin(var)
>>> var
0.7456241416655579
```

Listing 5.4: Type Loss in Standard Library

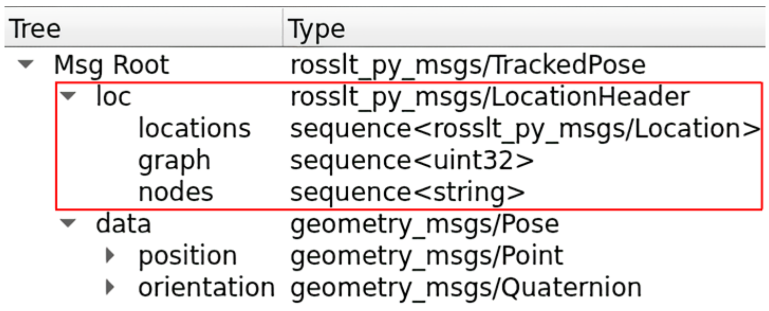
We require provenance support for all of the three native data types for numbers: `int`, `float`, and `complex`. To preserve the correct datatype on reversion for integers, we define the additional operators `MUL_INT` and its inverse `DIV_FLOOR`. This way we can differentiate between integer and float division operators. We also support provenance tracking for the `str` type by complementing the Python native string addition and multiplication operators with custom string subtraction and division implementations.

To make message publishers in ROS work with our wrapper class, we implement functions to serialize and deserialize the provenance information to be transmitted across node borders. As it is impossible for the additional data to be transparently added to existing message types, we define custom message types appending a `LocationHeader` containing the necessary provenance information.



Tree	Type
▾ Msg Root	geometry_msgs/Pose
▸ position	geometry_msgs/Point
▸ orientation	geometry_msgs/Quaternion

Figure 5.2: Original Pose Message



Tree	Type
▾ Msg Root	rosslt_py_msgs/TrackedPose
▾ loc	rosslt_py_msgs/LocationHeader
locations	sequence<rosslt_py_msgs/Location>
graph	sequence<uint32>
nodes	sequence<string>
▾ data	geometry_msgs/Pose
▸ position	geometry_msgs/Point
▸ orientation	geometry_msgs/Quaternion

Figure 5.3: Tracked Pose Message

Figure 5.2 shows the default ROS *Pose* message. By adding the provenance information to the message structure, we end up with a new message type shown in Figure 5.3. Wrapping the original structure in the *data* field, we can store the provenance information in the *loc* field. While *locations* holds a list representing each tracked value with their source location information, *graph* contains pairs of integer indices to this list representing the relations of the location tree.

5.5 Nested Attributes

The rosslt C++ library implementation uses macros to support provenance tracking for nested attributes in class instances such as ROS messages. This requires code adjustments to integrate tracking into existing programs and can lead to bugs if not done correctly.

```
auto message = Tracked<visualization_msgs::msg::Marker>();
auto pose = GET_FIELD(message, pose);
auto position = GET_FIELD(pose, position);

SET_FIELD(position, x, loc(1.5));
SET_FIELD(position, y, loc(2.5));
SET_FIELD(position, z, loc(3.0));

SET_FIELD(position, x, GET_FIELD(position, x) + 1);
SET_FIELD(position, y, GET_FIELD(position, y) * 2);
SET_FIELD(position, z, GET_FIELD(position, z) / 3);

auto m = static_cast<rosslt_msgs::msg::PoseTracked>(pose);
publisher->publish(m);
```

Listing 5.5: Macros in the C++ "rosslt" library

We seek a solution that does not require special care and works transparently without the need to write extra code for attribute read and write (RQ1).

To achieve this, we return temporary wrapper instances on attribute access. By overriding the Python class functions `__getattr__` and `__setattr__`, we can take control over the dot operator and reference the respective attributes on wrapper instantiation. This is currently not possible for the C++ library to implement, as C++23 does not allow for the dot operator to be overriden.

To keep everything in synchronization, the temporary wrapper instance is given a reference to the node within the location tree that represents the attribute's provenance and source location information.

In the future, upcoming C++ introspection features may allow for a similar approach. There is already a specification available that may help implementing the feature [ISO21], however it has not found its way into mainline compiler branches.

5.6 Operator Chaining

In this chapter we will explore a solution to minimize the expression history by optimizing the provenance tree (RQ4). We present an algorithm to reduce the number of operations in a provenance tree while preserving its expression. We do this by chaining operators together where possible and getting rid of operations involving their neutral element. For example, we want $2 * (\alpha/6) + 0$ to result in a provenance tree represented by $\alpha/3$. To do that, we first define the following:

- There are two groups of operators, (`+`, `-`) and (`*`, `/`), with the neutral elements 0 and 1. All other operators do not belong to a group and are ignored for expression reduction.
- The operators `+` and `*` are marked as commutative.
- The last applied operator in the provenance tree is being refered to as the *chain operator*, as this is the one that is being chained to the new operator. This is also the operator that is biased to stay after reduction, as there are always two solutions for different operators in the same group, e.g. $(\alpha + 2) - 3$ equals both $\alpha + (-1)$ and $\alpha - 1$. The operator to be applied is referred to as the “new” operator.
- There is a left and right hand side mode for the cases $operator(\beta, \alpha)$ and $operator(\alpha, \beta)$. This is important to note, as the rules for reduction differ between them. Internally the library handles this by using a `SWAP` token to flip the order of operands in the provenance tree.

The actual reduction works by applying a selection of common rules based on inspection of all possible cases:

- Ignore all cases where the old and new operator are of a different group, as these can not be reduced further.
- If the new operator is in left hand mode, swap the order of operands.
- If the chain operator is in left hand mode, apply the new operator in right hand mode.
- If the chain operator is in right hand mode, we apply the reverse of the new operator in the case that the chain operator is not commutative. Only if the new operator is in left hand mode, we set the chain operator to left hand mode as well, otherwise right hand mode is set.

Operations that have no effect due to usage of the neutral element can be removed. We do this after the reduction step to be able to get rid of operator chains that cancel each other out, such as $(\alpha * 2)/2$. If the last operand on the stack matches with the neutral element of the group of the last operator, the operator and operand are removed from the provenance tree. This includes addition/subtraction of 0 and multiplication/division with 1. If the old or new operator is in left hand side mode, the old operator has to be commutative as $0 - \alpha$ and $1/var$ can not be reduced, but $0 + \alpha$ and $1 * \alpha$ can. This extra condition does not matter for the right hand side cases $\alpha - 0$ and $\alpha/1$. If everything is done correctly, $((2 * (1 + \alpha))/2) - 1$ will result in an empty provenance tree represented by α, as it can be reduced completely.

An implementation of these rules can be found as a recursive function in Listing A.1 and in our library, as it is being called every time operators are executed on a tracked wrapper instance to add the data to the provenance tree. We have verified the correctness of the algorithm by creating test cases for all operator combinations and checked the results by evaluating the reduced provenance tree in normal and inverted form. It may also be possible to combine more groups, such as sequencial `POW` operations for exponents, or the `DIV_FLOOR` and `MULT_INT` operator variants, however this has not been implemented as part of this research.

5.7 Provenance Tree Serialization

The rosslt Python library provides an efficient way to track provenance information in ROS messages using a compact binary format. Previously, this was done by storing a serialized string representation of the provenance tree for each attribute in each message, which resulted in overhead while transmitting and parsing data with large provenance histories.

Our proposed approach uses operator codes (1 byte) and native byte representations of values instead of strings (RQ4). For floating point numbers, we use double precision (8 bytes) matching the native Python datatype for less error, even though it is double the size of a 4 byte float. The approach results in smaller message sizes if the string representation exceeds the 4 or 8 bytes the native format uses.

Shell	Size

`>>> str(1)`
`"1"`
	1 byte

`>>> str(1 / 3)`
`"0.3333333333333333"`
	18 bytes

`>>> list(struct.pack("d", 1 / 3))`
`[85, 85, 85, 85, 85, 85, 213, 63]`
	8 bytes

`>>> list(struct.pack("f", 1 / 3))`
`[171, 170, 170, 62]`
	4 bytes

Table 5.2: Sizes of Stored Values

The serialization currently supports the data types shown in Table 5.3. For the Python `complex` datatype, we store two 8 byte double values for both the real and imaginary parts sequentially into the value array. Strings are being serialized by storing the byte length into the value array followed by the UTF-8 encoded byte representation of the string.

Code	Name	Size
`0x01`	`INT32`	4 bytes
`0x02`	`INT64`	8 bytes
`0x03`	`DOUBLE`	8 bytes
`0x04`	`COMPLEX`	16 bytes
`0x05`	`STRING`	variable

Table 5.3: Serialization Data Types

A message containing provenance information includes two byte arrays: one containing the operator identifier codes shown in Table 5.1 plus data references, and one byte array containing the number data itself in native byte representation. The operator identifiers are stored in the code array with an offset of 0x40 to be able to distinguish them from the data type codes shown in Table 5.3.

Name	Data	Size
Expression String	`-1.3833333333333333;swap;-;4.2;*;sin`	36 bytes
Value Array	`222222222222F6BFCDCCCCCCCCCC1040`	16 bytes
Code Array	`034042034447`	6 bytes

Table 5.4: Serialization Sizes

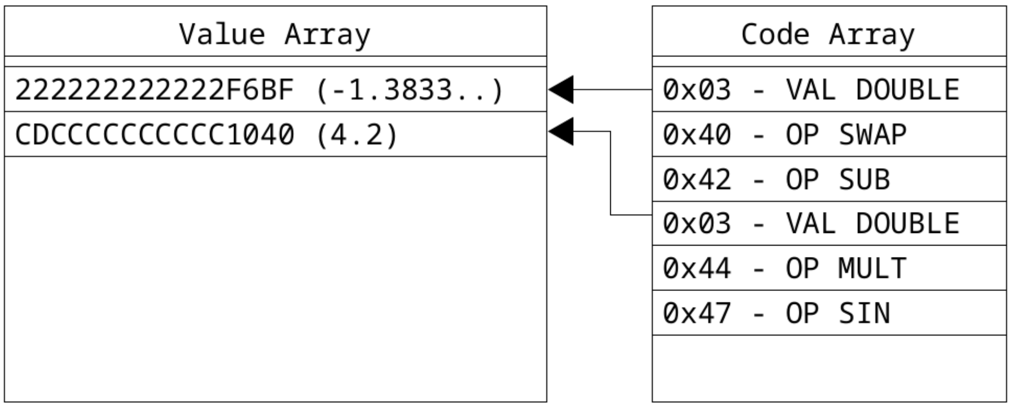


Figure 5.4: Serialization Data

In order to deserialize the data, the arrays are traversed sequentially from left to right. We move through the code array to append the identifier codes of the operators of the expression representing the provenance tree. Data references are processed by retrieving the corresponding values from the value array at the current data cursor position. Every time the value array is being accessed to read a number, the data cursor is being advanced by the size of the respective data type. This process continues until all operators have been processed or the end of either array has been reached.

To reduce the size of the provenance information even further, compression can be applied if the length of the expression exceeds a configured threshold. Since the data reference and operator code lists are stored as byte arrays, they can easily be compressed using standard compression techniques. In our case, we have decided to use zlib for making use of the DEFLATE compression algorithm, as it is part of the Python standard library. In distributed ROS applications, this can significantly decrease the amount of data being sent between nodes over network, while still maintaining full access to the provenance information. For small provenance tree sizes, the DEFLATE algorithm can lead to bigger data, which is why the threshold is necessary. We also note, that prefixing with the zlib header adds two bytes to the length of the data: `789C` for “default compression” [DlG96].

Name	Data	Size
Original	`222222222222F6BFCDCCCCCCCCCC1040`	16 bytes
Z-Lib	`789C535202816FFBCF9E010101070035D9079B`	19 bytes

Table 5.5: Compression of Small Data

Nodes do not usually modify each tracked attribute that a message contains. As such, the provenance information for the untouched attributes does not have to be altered, as no operation is taking place on them. In these cases, we can leave the data in the packed state as stored in messages. This gets rid of the overhead of unpacking the provenance tree, reduces memory overhead as the data may still be in compressed state, and makes it possible to completely skip packing the provenance tree if it is being sent in a new message to another node. We refer to this as *lazy decoding* as in our library the provenance tree data is only being decoded if necessary.

In the future, the code array size may be reduced even further by combining operator and value codes using 4 bits each to fill a single byte. However, this approach would decrease the operator and value maximum count to 16 each, limiting extensibility.

5.8 Data Monitoring

It is often of advantage to access live provenance information for inspecting the flow of data in complex robotics systems. As the new serialization format of the provenance information is not human-readable anymore, we develop a utility tool to easily inspect the history for tracked objects between ROS nodes (RQ5). The decoding of our serialized data is possible by integrating our Python library, which demonstrates that the library is usable for custom tooling. Additionally, we provide the possibility of forcing desired values by backpropagation at any time by making use of source location information and the provenance tree.

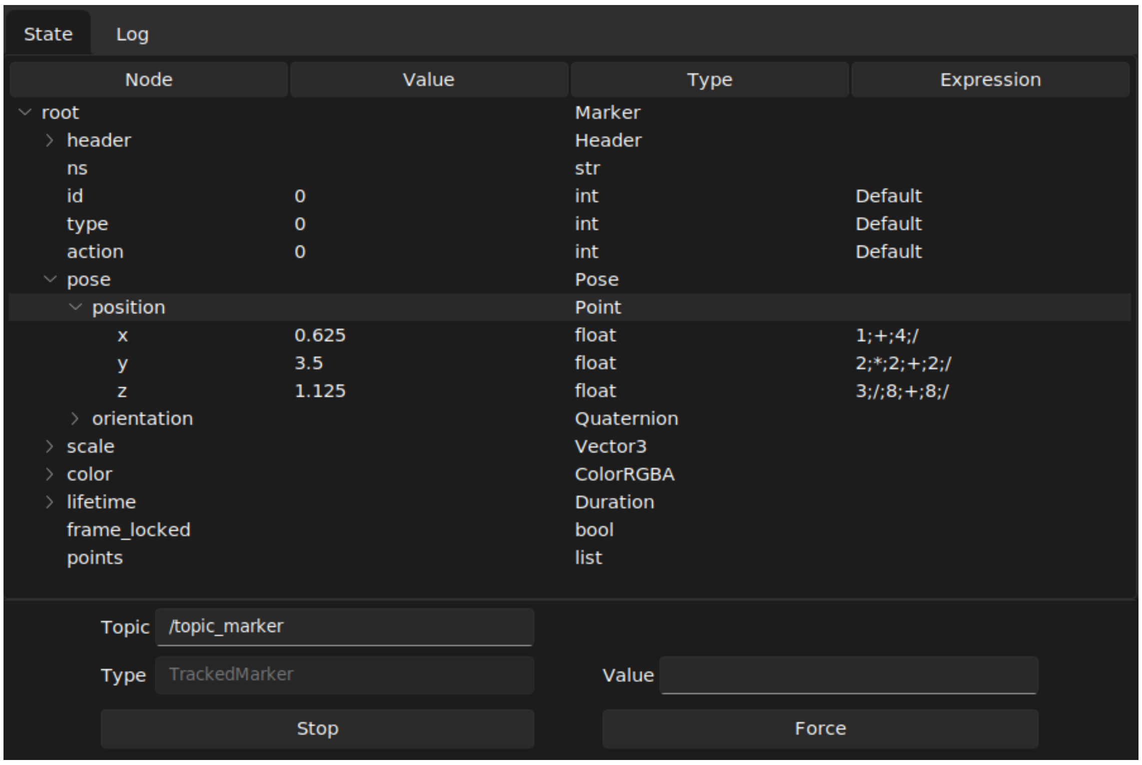
To make this work, we need to dynamically support any message type, as custom types containing provenance information are not known to the Python library at time of creation. This is made possible by making use of the *rclpy* ROS support library and dynamic type information features available in Python. Our *rosslt* Python library allows us to parse the provenance information message header to read both location and provenance tree structures stored in our custom format. The node names of the decoded location tree can then be matched to the introspected message type attribute names, resulting in a combined human-readable view.

Intercepting generic ROS messages is possible by making use of subscribing to a given topic path, while querying the type information from the “ros2topic” API in the support libraries available by default. This allows for displaying any transmitted data currently in use by the system, as ROS nodes are meant to only communicate via messages and topics.

The state frame shown in Figure 5.5 allows accessing the most up to date data point that has been transmitted to the specified message topic. Displaying the data requires building a node tree based on the message structure without the provenance information and combining it with the location tree, accumulating any provenance tree information being sent over time to the topic by any active node. By setting a custom value and activating the “Force” button, we can make use of the provenance tree of the given attribute to backpropagate changes to update the value to the specified value. The changes are being sent to the origin ROS node the attribute is instatiated in, so that this node can apply the new starting value when it updates its own state.

In the logging frame shown in Figure 5.6, we provide information about changes to the data flow. By recording all messages we can see the exact value operation history in form of an expression at any point in time.

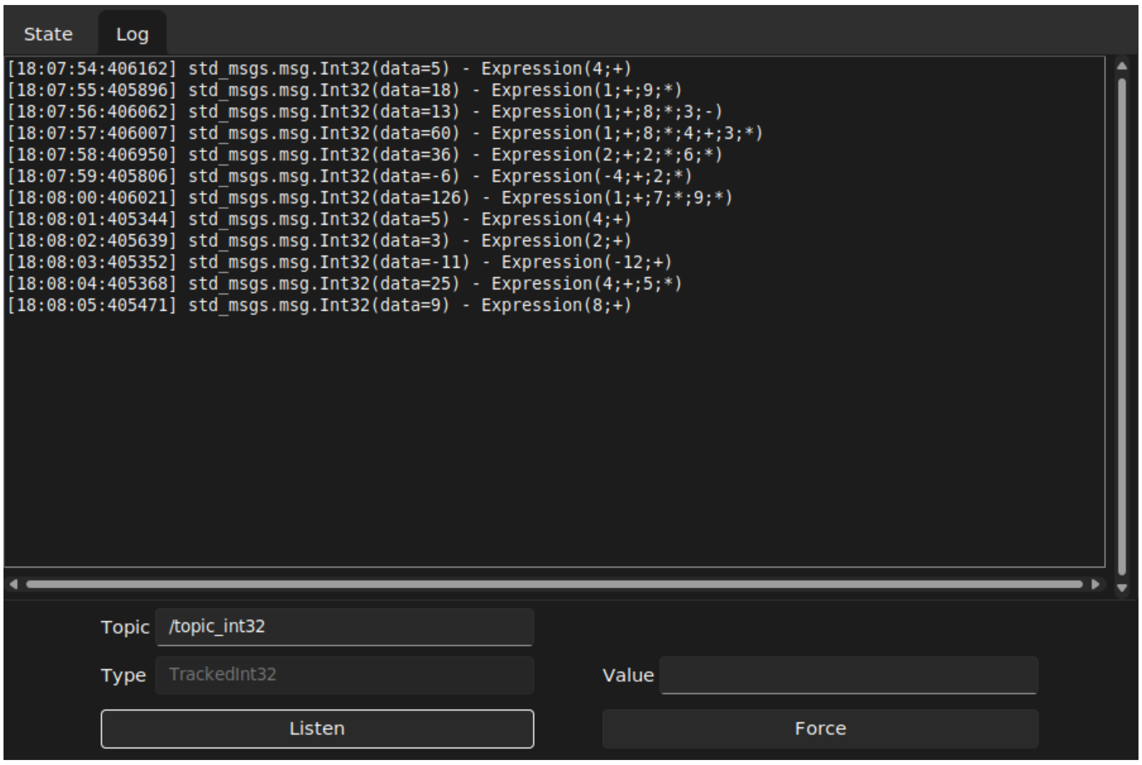
Future work may include integrating provenance tree changes directly into the state display, so that any change of operator application order can be followed more closely. This would allow to more easily inspect node internal logic and state changes.



Node	Value	Type	Expression
root		Marker	
header		Header	
ns		str	
id	0	int	Default
type	0	int	Default
action	0	int	Default
pose		Pose	
position		Point	
x	0.625	float	1;+;4;/
y	3.5	float	2;*;2;+;2;/
z	1.125	float	3;/;8;+;8;/
orientation		Quaternion	
scale		Vector3	
color		ColorRGBA	
lifetime		Duration	
frame_locked		bool	
points		list	

Topic /topic_marker
Type TrackedMarker
Value

Figure 5.5: State Frame



```
[18:07:54:406162] std_msgs.msg.Int32(data=5) - Expression(4;+)
[18:07:55:405896] std_msgs.msg.Int32(data=18) - Expression(1;+;9;*)
[18:07:56:406062] std_msgs.msg.Int32(data=13) - Expression(1;+;8;*;3;-)
[18:07:57:406007] std_msgs.msg.Int32(data=60) - Expression(1;+;8;*;4;+;3;*)
[18:07:58:406950] std_msgs.msg.Int32(data=36) - Expression(2;+;2;*;6;*)
[18:07:59:405806] std_msgs.msg.Int32(data=-6) - Expression(-4;+;2;*)
[18:08:00:406021] std_msgs.msg.Int32(data=126) - Expression(1;+;7;*;9;*)
[18:08:01:405344] std_msgs.msg.Int32(data=5) - Expression(4;+)
[18:08:02:405639] std_msgs.msg.Int32(data=3) - Expression(2;+)
[18:08:03:405352] std_msgs.msg.Int32(data=-11) - Expression(-12;+)
[18:08:04:405368] std_msgs.msg.Int32(data=25) - Expression(4;+;5;*)
[18:08:05:405471] std_msgs.msg.Int32(data=9) - Expression(8;+)
```

Topic /topic_int32
Type TrackedInt32
Value

Figure 5.6: Logging Frame

6. Experimental Evaluation

Our evaluation benchmarks are created on an AMD Ryzen™ 2700X CPU with 3.70 GHz and 8 cores on the GNU/Linux based ROS 2 Humble distribution using Python 3.10.6. However, all of the tasks are being run in a single thread. We are looking at the general overhead of our tracking wrapper in Figure 6.1 by creating a tracked instance and applying a number of random operators on it. An important fact to note is, that in contrast to the inline operators (`+=`, `-=`, `*=`, `/=`), the operators `+`, `-`, `*`, `/` are creating a new instance, which leads to the provenance tree being copied 50% of the time on average within our benchmarks.

```python
def apply_random(val, rng, operand):
    case = rng.randint(1, 8)
    if case == 1:
        val = val + operand
    elif case == 2:
        val = val - operand
    elif case == 3:
        val = val * (operand + 1)
    elif case == 4:
        val = val / (operand + 1)
    elif case == 5:
        val += operand
    elif case == 6:
        val -= operand
    elif case == 7:
        val *= (operand + 1)
    elif case == 8:
        val /= (operand + 1)
    return val
```

Listing 6.1: Random Operator Benchmark

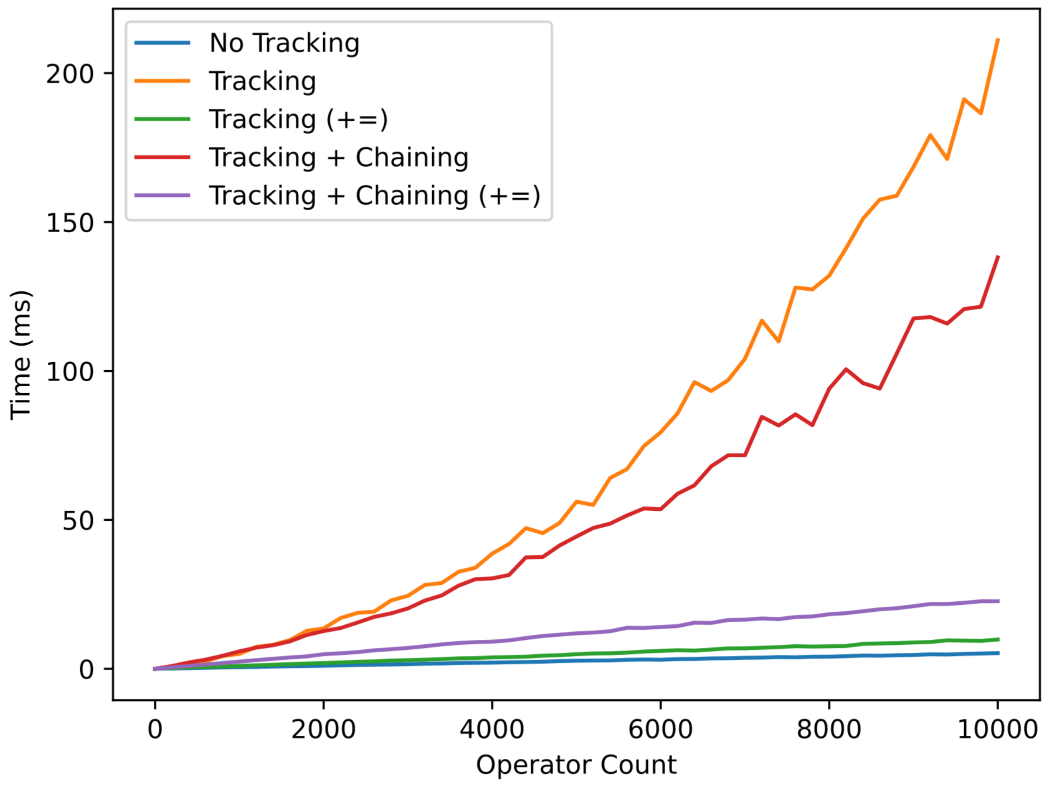


Figure 6.1: Provenance Tree Runtime

The dataset without tracking in Figure 6.1 serves as a baseline reference to measure the time it takes to apply the operators to a raw native value. As expected, the execution time grows linearly to the amount of operators being applied. The tracking dataset grows faster than linear, because the time taken to copy the provenance tree increases with its size. By limiting ourselves to only the `+=` operator, the provenance tree does not get copied, allowing the chaining to apply on each operator, and ultimately results in linear execution time growth and much smaller overhead. Enabling the chaining of operators adds a considerable execution time overhead (∼250%) to the tracking without provenance tree copy, due to the extra instructions that need to be processed on every tracked operation. However, this results in smaller provenance trees, which means less data to copy when a new instance is being created. This makes the chaining outperform tracking without chaining for random operations being applied, with the relative difference increasing with the operator count. In Figure A.1 we supply the accompanying serialized provenance tree size that affects the execution time on copy operations. We assume that in practice, it is more likely that the same operator is being applied over and over again, allowing the chaining to take advantage of the simplification. We conclude that it is generally always worth it to enable the chaining of operators even for execution time.

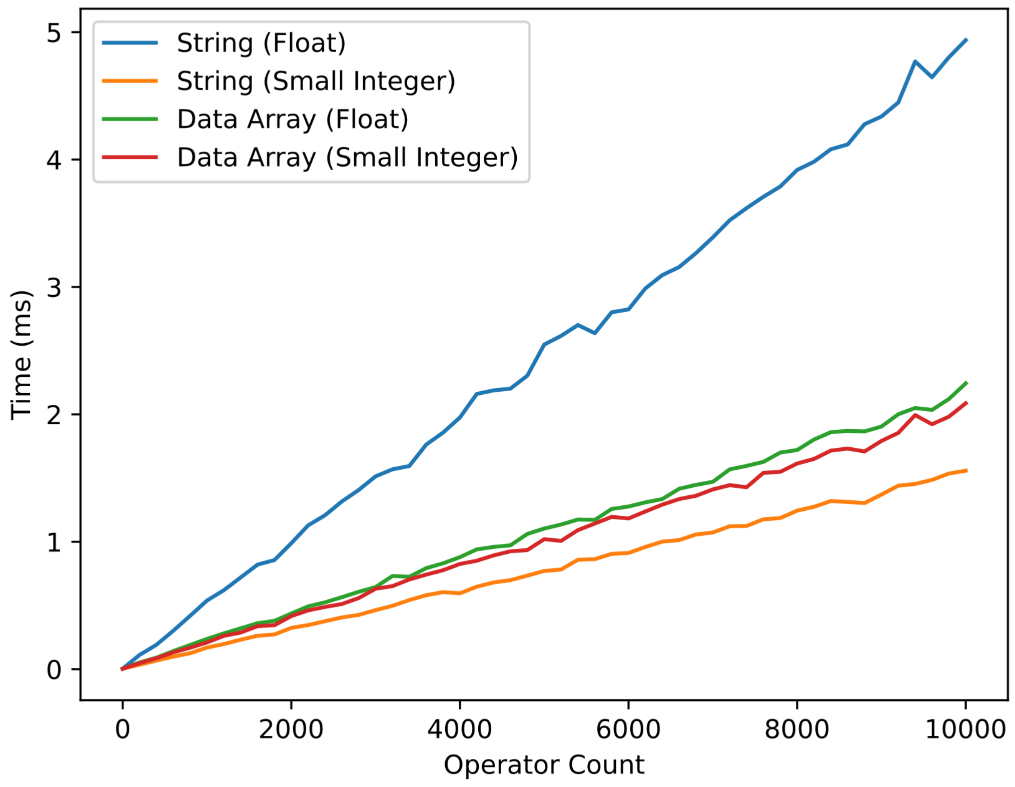


Figure 6.2: Message Runtime Serialize

Serialization to ROS message structures is necessary to propagate the provenance information across nodes. Performance depends on the methods and data structures used in the process. In Figure 6.2 we compare UTF-8 encoded strings like they are used in the C++ rosslt library against the data arrays we introduced previously in Section 5.7. As the message size increases linearly with the operator count, it is proportional to the execution time of serialization. We benchmark two categories: random floating point numbers in the range [0,1] that serve as the worst case for strings, and small integers in the range [1,9] that serve as the best case for strings as they can be represented by one character. As the string joining functionality in Python, which we are using to build the expression strings, is well optimized and accelerated by native code in the Python interpreter, there is not much headroom available when building the data arrays in pure Python instead of strings. However, we can see that our proposed data arrays show great improvement (∼57%) compared to the string data set of random floats, which we consider the worst case for the string representation. Limiting ourselves to small integers which can be represented by one character, the string representations are able to outperform the data arrays by a small amount (∼30%). We conclude that the data arrays are generally faster in practice, where the values are not limited to small integers that can be represented by one character, especially for robot applications where non-integer floating point values are widespread.

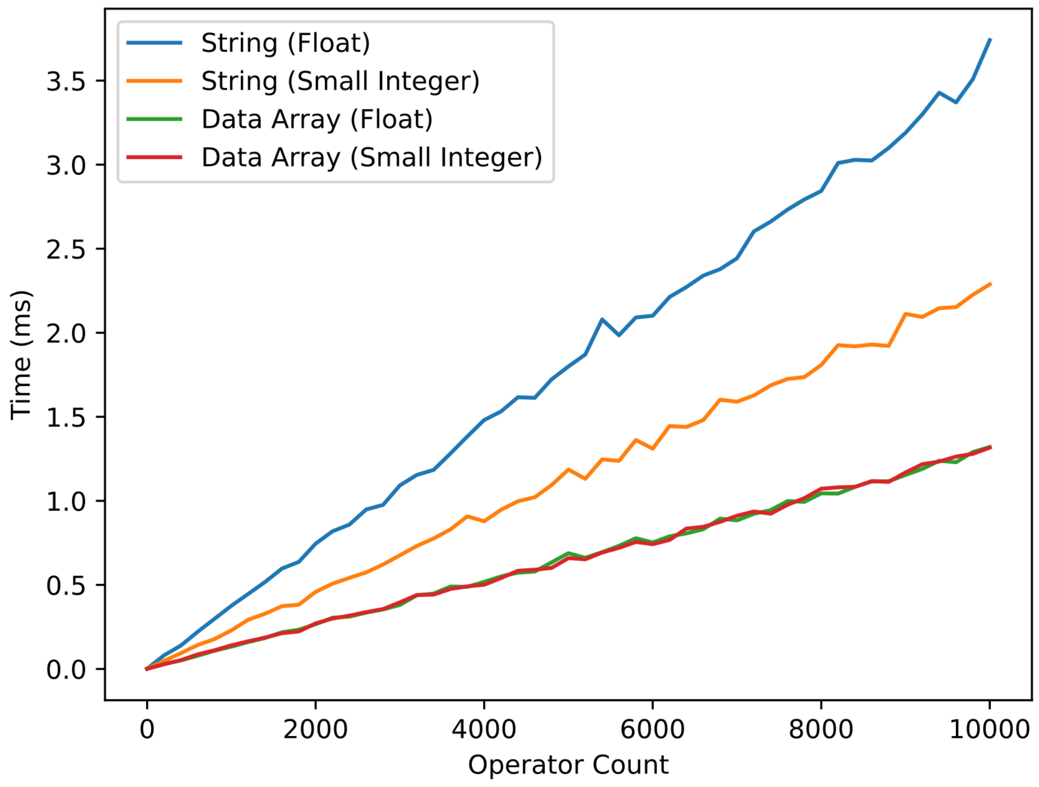


Figure 6.3: Message Runtime Deserialize

When a ROS node receives a message containing provenance data, it needs to deserialize it in order to apply any further operations on the data. This results in execution time overhead which we measure in Figure 6.3. On deserialization the strings take longer to process than the data arrays for both cases, the small integers in range [1,9] (∼76% slower) and the floating point values in range [0,1] (∼182% slower). The data type does not affect the data array representation in a meaningful way and both cases of our proposed data array representation show only neglible difference (<1%) in execution time. We conclude, that for deserialization of the data arrays, execution time is improved compared to the string processing if a minimum amount of data is being involved.

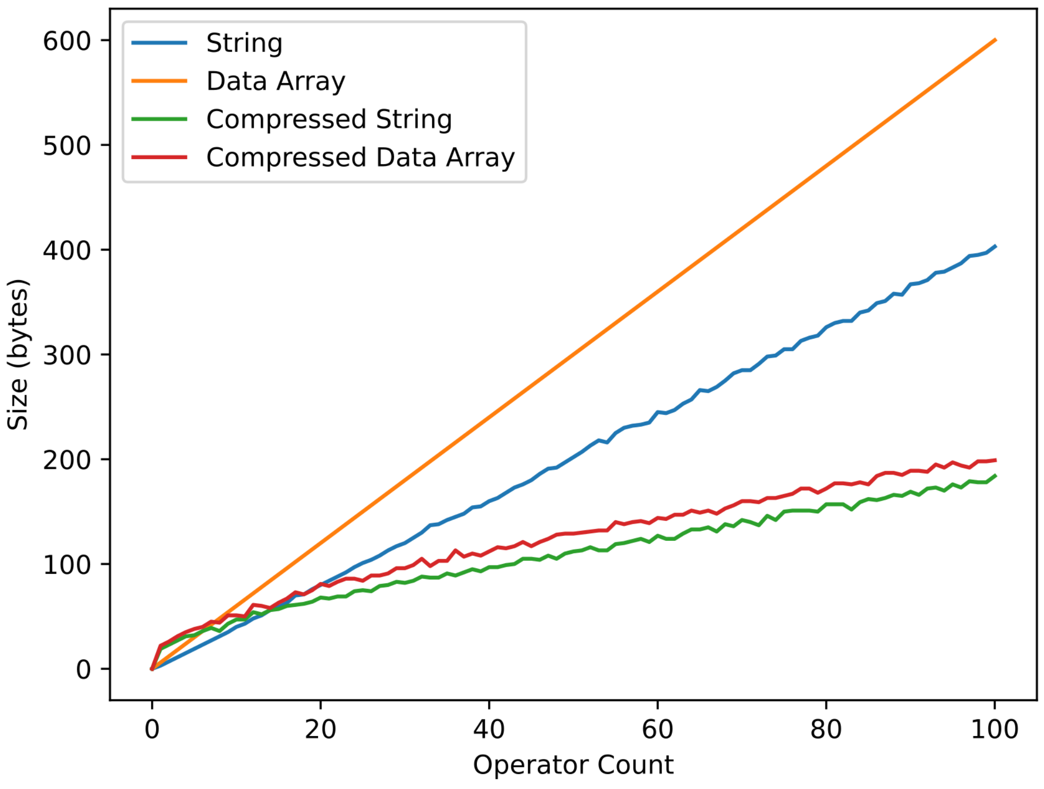


Figure 6.4: Provenance Tree Data Size (Small Integers)

As provenance trees grow in size per operator applied on their tracked object, it introduces memory overhead compared to raw values without tracking. We analyze the data size of the serialized provenance tree in Figure 6.4 for small integers in the range [1,9], for both string representation and data arrays, and compare it to their zlib compressed variant as introduced in Section 5.7. The small integers serve as the best case for strings, as these values can be represented by a single character, which explains the strings outperforming the data arrays in this benchmark. Both compressed variants are generally comparable and do not show a big difference in size to each other within our data set (∼10%), however they show a significant improvement to their uncompressed variants starting at 20 or more operators. The size of the uncompressed data array can be directly calculated in this case by multiplying the operator count by 6, as the integer is stored using 4 bytes, and the operator and type reference are using 1 byte each. As the data arrays only support 32 and 64 bit integers, the strings outperform the data arrays in uncompressed form (∼34%) for the small integers in range [1,9], as there is a 3 byte overhead for the used 32 bit format. This also reflects itself in compressed form, as the zlib dictionary can be better utilized for the smaller string representation, however the difference is neglible.

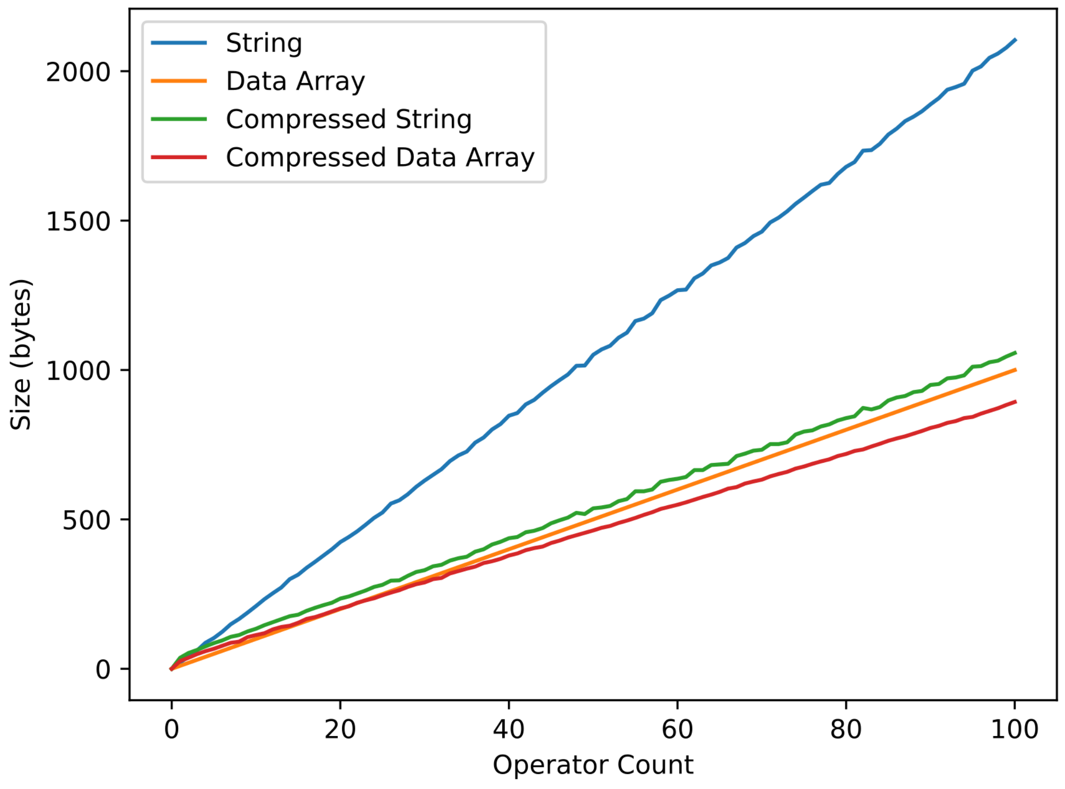


Figure 6.5: Provenance Tree Data Size (Random Floats)

In Figure 6.5 we analyze the data size of the serialized provenance tree for floating point values in the range [0,1], which serves as the worst case for the string representation. In constrast to the small integers used in Figure 6.4, we can see the data arrays outperforming the string representation (∼53%). In fact, the uncompressed data arrays are slightly smaller than the compressed strings (∼6%). However, compressing the data arrays results in only small size improvements compared to the uncompressed data arrays at 100 operators (∼9%), due to it having only small amounts of redundant data. We conclude that for floating point values, it is generally not worth it to enable compression for the data arrays, due to the added runtime execution penalty of zlib. If the inefficient string representation of the floating point data set is used however, we can approach the memory efficiency of the data arrays by enabling compression.

Threats to Validity

In our benchmarks, execution time is measured as the average runtime for a given task instance. To minimize error spikes by system background processes, we execute multiple runs per data point and use the minimum average runtime in our graphs which we assume is representative of the actual performance. The absolute values of our execution time benchmarks can not be taken literally, as we are limited to only one test system using a specific processor. As such, we focus on the differences and growth rates of our selected data sets, and compare them in order to take conclusions. While we acknowledge the runtime penalty of the zlib compression, we did not measure it due to the general availability of this information.

To compare the provenance tree memory overhead, we measure the data size in serialized state which is assumed to be consistent, as the serialization format is system independent. However, we do not take the actual Python interpreter system memory overhead of the runtime objects in account, which is generally harder to measure and may differ between computer systems and runs. We also do not measure huge data sets when comparing compression efficiency on serialization, but limit ourselves to 100 operators. This enables us to identify the points where the efficiency of compression outweights its overhead. However, we assume that for bigger data sets, compression only improves the storage efficiency in the serialization process.

By making use of random operators in our data sets, we assume to measure workloads that are worse than what is used in practice. We assume that in tight loops of computer applications, values are usually processed using the same type of operator, which results in our operator chaining method to be more effective. However, we include the case that only one operator is being used for a given tracked value for comparision, which results in much lower memory usage and execution time on provenance tree copy operations, than what we assume for pratical workloads.

As the implementation of our library might have implementation bugs, we are using a range of unit tests to validate our resulting provenance trees, their inverted variants, the location trees of our generic value wrapper, and general semantics while using the library. However we can not automatically validate the results of our evaluation data by using these unit tests.

We expect the memory usage and execution time to be different if the feature set of the library changes. For example, additional data types in the provenance tree serialization might increase memory efficiency but also alter execution time. As such, our results are only valid for the library version that was existing at time of writing and might be different in the future.

7. Conclusion

Provenance tracking offers ways to inspect and analyze the flow of data during program runtime while offering abilities for code self-explainability, allowing programs modifying their own parameters to match a desired outcome. We present a Python library allowing for easy integration into existing program code, while utilizing a range of optimization techniques lowering the overhead of value tracking.

RQ 1: How can alternative approaches be employed for extracting nested attributes from tracked objects?
We implement a generic wrapper that can be used to encapsulate nested object structures. By overriding operator functions we can keep track of operations being applied to the object itself, and by making use of the dot operator to create temporary wrappers around its attributes we can enable provenance tracking for nested data structures such as ROS message types.

RQ 2: Which data structures are suitable for the provenance tree?
Storing the provenance tree in a variable type array allows for efficiently appending new data without type casting nor conversion. By implementing operators in an object based approach, we can store operand values and operator references directly without additional overhead.

RQ 3: How can the source location information be stored for nested attributes?
By constructing an object based location tree live at runtime, we can dynamically support custom nested data structures and keep track of each attribute and its provenance information. This allows us to wrap the message types necessary to support provenance information propagation over ROS node boundaries.

RQ 4: Which improvements can be made to reduce the message overhead of the provenance data?
As provenance trees of values can grow large on heavy operator usage, we chain together groups of operators to reduce the tree to a smaller equivalent to save on memory usage and parsing complexity. We achieve this by applying a set of rules based on the used operators when appending data to the tree, while introducing a moderate execution time overhead. We further minimize memory usage of provenance information stored within ROS messages by implementing a new binary serialization format and apply compression on its data arrays when a configured size threshold is reached.

RQ 5: How can live tracking data be decoded and displayed?
To inspect live provenance data, we supply a monitor utility integrating our library to display the operator history of each attribute in ROS messages. We showcase the functionality of forcing values by making use of the available provenance information to calculate the required program parameter automatically tracked with source location information by evaluating the inverted provenance tree with the desired target value and propagate result to the origin node so it will automatically be applied on reevaluation.

RQ 6: How do the library adjustments impact the performance and usability?
We analyzed the performance implications of provenance tracking using our Python library and identified the duplication of big provenance trees as a bottleneck for execution time. By using the binary serialization format, we found that it outperforms the string representation which the existing C++ rosslt library uses for random floating point numbers. We note, that to approach the memory efficiency of the string representation for small integers, new data types need to be implemented to the serialization format. Because of our adjusted wrapper solution, integrating provenance tracking does not require adjusting attribute access of tracked object structures anymore. As there is less changes required to existing source code and complexity is decreased for writing new code, the usability of the library is increased.

8. Future Work

Finally, we elaborate on potential future work regarding our approach to provenance tracking in Python.

As copying the provenance tree is inperformant, research can be done how to avoid duplicating it in whole when applying non-inline operators on tracked values. This could be achieved by replacing the postfix memory representation of the provenance tree by an object based tree structure that allows the wrapper to reference partial provenance trees without having to copy the content. This reduces memory and execution time overhead in cases where copy operations of provenance trees can be avoided.

Supporting multiple locations in the same provenance tree requires a more complex approach in the implementation. As the inverted provenance tree can have multiple possible results, a heuristic needs to be implemented to get reasonable results which is non-trivial to solve.

By adding more data types to the provenance tree serialization format, such as `INT8` and `INT16`, it could be possible to approach the memory efficiency of strings, when only small integers that can be represented by one or two characters are involved. Even though there would need to be additional data range checks to detect what data type should be used to store a given value, we expect this to only slightly impact the execution time.

We currently support provenance information of boolean types by treating the `and` operator as multiplication, `or` as addition, and storing the result as number. Inversion of the provenance tree is still possible, however the result is a number that is allowed to be negative. To correctly support inverted provenance trees of booleans, additional operators need to be defined and we suspect that a solver for boolean equations may be necessary in order to find the required values in case multiple locations are involved.

Both Python and C++ are the main languages supported by ROS 2, as such many projects are making use of both languages for their variety of program nodes. To properly support provenance tracking in this constellation, there needs to be a C++ library counterpart that is compatible to our Python implementation. While the current rosslt C++ library can be adjusted to share the same message format, it may be worth looking into semantic differences, especially how objects are treated in regards to the object reference based approach in Python and the copy-by-value semantics in C++. Upcoming introspection features [ISO21] that are not yet part of mainline compiler branches may even allow for an approach that makes use of dynamic temporary wrappers that are similar to what we have designed in our Python implementation.

A. Appendix

A.1 Operator Chaining Code

```
def operator_chain(history, data):

  # recursion end
  if len(data) < 2: return history + data

  # check for new swap and operator
  new_swap = data[1] is Operator.SWAP
  op_new_index = 1 + int(new_swap)
  op_new = data[op_new_index]
  if type(op_new) is Operator:

    # ignore neutral element
    if not new_swap or op_new.commutative:
      if data[0] == op_new.neutral:
        return operator_chain(history, data[op_new_index+1:])

    # simplify tree
    if len(history) > 1:

      # check chain operator and swap
      op_chain = history[-1]
      if type(op_chain) is Operator and op_chain.group:
        chain_swap = history[-2] is Operator.SWAP
        if chain_swap or type(history[-2]) is not Operator:

          # check if both operators are in same group
          # guarantees new operator to be in a group as well
          if op_new.group == op_chain.group:
```

```
# check for chain swap
if chain_swap:
  # apply operator in right hand mode
  history.pop()
  history[-1] = data[0]
  if new_swap:
    history[-1], history[-2] = \
        history[-2], history[-1]
  op_new(history)
  history.append(Operator.SWAP)
else:
  # apply operator in left hand mode
  history[-1] = data[0]
  if new_swap:
    history[-1], history[-2] = \
        history[-2], history[-1]
  if op_chain.commutative:
    op_new(history)
  else:
    op_new.reversed(history)
  if new_swap:
    history.append(Operator.SWAP)

# continue chain
if new_swap:
  if chain_swap and op_chain.negate:
    history.append(op_new.reversed)
  else:
    history.append(op_new)
else:
  history.append(op_chain)

# simplify neutral element
if chain_swap or new_swap:
  # left hand side for commutative operators
  if history[-1].commutative:
    if history[-1].neutral == history[-3]:
      history.pop()
      history.pop()
      history.pop()
else:
  # right hand side neutral element
  if history[-1].neutral == history[-2]:
    history.pop()
    history.pop()
```

```
          # process next operator
          if op_new_index + 1 < len(data):
            return operator_chain(history,
                data[op_new_index + 1:])
          else:
            return history

  # create expression with new history
  history.extend(data)
  return history
```

Listing A.1: Operator Chaining Code

A.2 Provenance Tree Data Size with Chaining

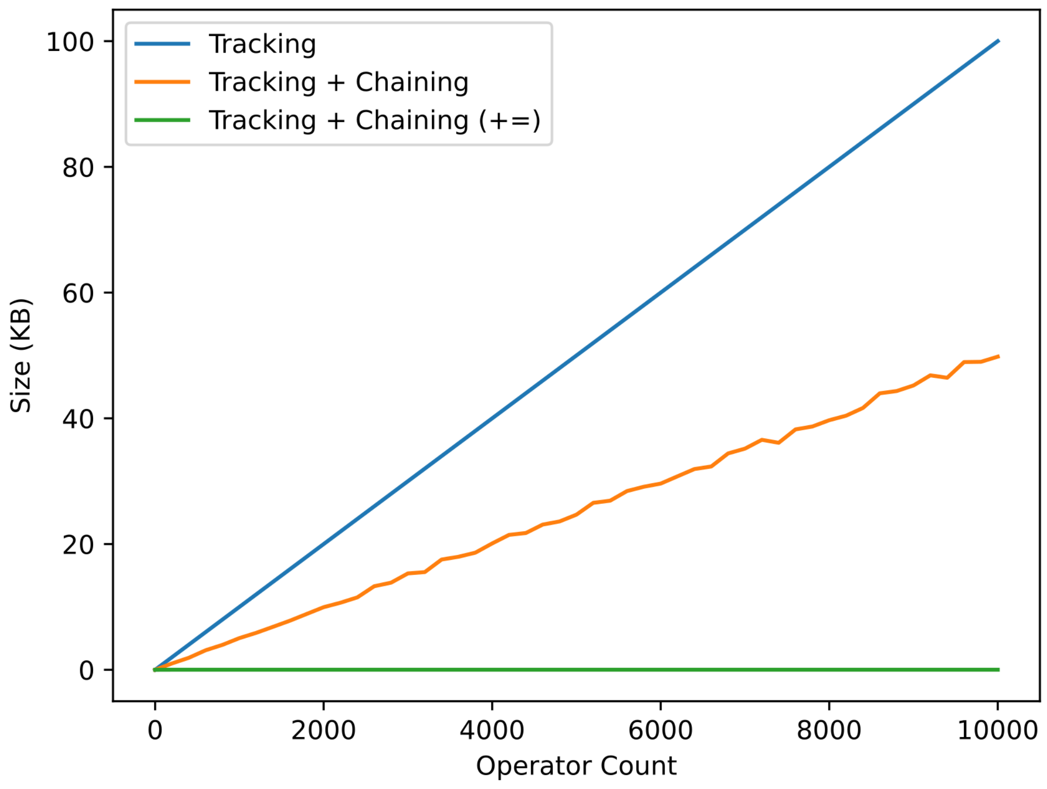


Figure A.1: Provenance Tree Size with Chaining

A.3 Calculating Parameters for Unknown Functions

```
def blackbox(x):
    from random import Random
    rng = Random(1337)
    while rng.random() < 0.9:
        choice = rng.randint(0, 8)
        val = rng.random() * 10
        if choice == 0:
            x = x + val
        elif choice == 1:
            x = x - val
        elif choice == 2:
            x = x * (val + 1)
        elif choice == 3:
            x = x / (val + 1)
        elif choice == 4:
            x += val
        elif choice == 5:
            x -= val
        elif choice == 6:
            x *= (val + 1)
        elif choice == 7:
            x /= (val + 1)
        elif choice == 8:
            x = x ** 0.75
    return x

from rosslt import Tracked
param = blackbox(Tracked(0.0)).get_expression().reverse()(42.0)
print(f"blackbox({param}) = {blackbox(param)}")

# program output
blackbox(13.231650832969262) = 42.0
```

Listing A.2: Blackbox Example

Bibliography

[AACP13] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 21:919–969, 2013. 6. (cited on Page 1, 3, and 4)

[BGCG$^+$19] Mathias Blumreiter, Joel Greenyer, Francisco Javier Chiyah Garcia, Verena Klös, Maike Schwammberger, Christoph Sommer, Andreas Vogelsang, and Andreas Wortmann. Towards self-explainable cyber-physical systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 543–548, 2019. (cited on Page 1)

[Car23] Carbide Alpha — alpha.trycarbide.com. https://alpha.trycarbide.com/, 2023. (cited on Page 11)

[CGRL15] Huaxiong Cao, Naijie Gu, Kaixin Ren, and Yi Li. Performance research and optimization on cpython's interpreter. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 435–441, 2015. (cited on Page 2 and 14)

[Cza12] Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 30, 2012. (cited on Page 12)

[DlG96] L. Peter Deutsch and Jean loup Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950, May 1996. (cited on Page 28)

[Fit85] John Fitch. Solving algebraic problems with reduce. *Journal of Symbolic Computation*, 1(2):211–227, 1985. (cited on Page 12)

[GTA21] Pablo Galindo, Batuhan Taskaya, and Ammar Askar. Include fine grained error locations in tracebacks. PEP 657, 2021. (cited on Page 18)

[Ham62] C. L. Hamblin. Translation to and from Polish Notation. *The Computer Journal*, 5(3):210–213, 11 1962. (cited on Page 10)

[HGF22] Dave Hershberger, David Gossow, and Josh Faust. Rviz. https://wiki.ros.org/rviz, 2022. (cited on Page 7)

[HLC19] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-sketch: Output-directed programming for svg. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST ’19, page 281–292, New York, NY, USA, 2019. Association for Computing Machinery. (cited on Page 12)

[HP76] Eldon Hansen and Merrell Patrick. A family of root finding methods. *Numerische Mathematik*, 27(3):257–269, Sep 1976. (cited on Page 11)

[ISO20] Interface definition language (IDL) 4.2. Standard, International Organization for Standardization, Geneva, CH, February 2020. (cited on Page 6)

[ISO21] C++ extensions for reflection. Standard, International Organization for Standardization, Geneva, CH, October 2021. (cited on Page 23 and 42)

[LN89] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, Aug 1989. (cited on Page 11)

[Loi23] Sébastien Loisel. GitHub - sloisel/numeric: Numerical analysis in Javascript. https://github.com/sloisel/numeric, 2023. (cited on Page 11)

[MFG+22] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. (cited on Page 6)

[MSCG23] Steven Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. Impact of ros 2 node composition in robotic systems. *IEEE Robotics and Autonomous Letters (RA-L)*, 2023. (cited on Page 6)

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009. (cited on Page 1 and 6)

[Roj96] Raúl Rojas. *The Backpropagation Algorithm*, pages 149–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. (cited on Page 5)

[Ros23] rostopic - ROS Wiki — wiki.ros.org. http://wiki.ros.org/rostopic, 2023. (cited on Page 7)

[rqt23] rqt - ROS Wiki — wiki.ros.org. http://wiki.ros.org/rqt, 2023. (cited on Page 7)

[Ske23] GitHub - ravichugh/sketch-n-sketch: Direct Manipulation Programming for HTML/SVG — github.com. https://github.com/ravichugh/sketch-n-sketch, 2023. (cited on Page 12)

[Tho20] Dirk Thomas. Idl. https://design.ros2.org/articles/idl_interface_definition.html, July 2020. (cited on Page 6)

[Wit21] Thomas Witte. rosslt. https://github.com/sp-uulm/rosslt, 2021. (cited on Page 1)

[WT19] Thomas Witte and Matthias Tichy. A hybrid editor for fast robot mission prototyping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 41–44, 2019. (cited on Page 1, 3, and 5)

[WT21] Thomas Witte and Matthias Tichy. Inferred interactive controls through provenance tracking of ros message data. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pages 67–74, 2021. (cited on Page 9)

[ZKK$^+$12] Olive Qing Zhang, Ryan K.L. Ko, Markus Kirchberg, Chun Hui Suen, Peter Jagadpramana, and Bu Sung Lee. How to track your data: Rule-based data provenance tracing algorithms. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1429–1437, 2012. (cited on Page 3)