Inferred Interactive Controls Through Provenance Tracking of ROS Message Data

1st Thomas Witte
Inst. of Software Engineering and Programming Languages
Ulm University
Ulm, Germany
thomas.witte@uni-ulm.de

2nd Matthias Tichy
Inst. of Software Engineering and Programming Languages
Ulm University
Ulm, Germany
matthias.tichy@uni-ulm.de

***Abstract*—Interactive controls that enrich visualizations need domain knowledge to create a sensible visual representation, as well as access to parameters and data to manipulate. However, source data and the means to visualize them are often scattered across multiple components, making it hard to link a value change in the interface to the appropriate source data. Provenance, the documentation of the origin and history of message data, can be used to reverse the evaluation of a value and change it at its source. We present a communication pattern as well as a C++ support library for ROS to track the provenance of message data across multiple nodes and apply source changes, reversing any transformation on the tracked data. We demonstrate that it is possible to automatically infer interactive 3D user interfaces from standard, non-interactive ROS visualizations by leveraging this additional tracking information. Preliminary results from a prototypical implementation of multiple origin tracking enabled ROS nodes indicate, that this tracking introduces a significant but still practicable message size and serialization performance overhead. To apply this tracking to existing C++ codebases only small, syntactic changes are necessary: a wrapper type around tracked values hides all necessary bookkeeping.**

***Index Terms*—data provenance, ROS, source location tracking, bidirectional evaluation, interactive markers**

I. Introduction and Motivation

Since its inception in 2007, the Robot Operating System (ROS) [12] has become one of the most widely used robotics frameworks. One reason for its success is probably the standard visualization tool RViz [9]. It can display 3D data and marker messages from all nodes, integrated into one scene, thereby providing a consistent view into the system's state. The user profits from the mix of visualizations from all components and layers of the application, as results from one component can be easily compared with the others. Differences or inconsistencies are easily spotted.

Interactive markers [6] extend this concept and enable the creation of 3D interactive interfaces. They are harder to implement, as the visual interface must be manually linked to the appropriate internal data through a callback function. A node that is able to create a meaningful visual representation might not be able to access this internal state. If the parameters or source data were received through a message from another node, the callback function of the interactive marker cannot change them across the node boundary without significant additional coding effort. Conversely, the node with access to this state or data often lacks domain knowledge to create appropriate visualizations and controls.

Data provenance [1] is a record of the origin, change history and relation of a data item to its source. ROS applications can profit from additional provenance information in ROS messages: e.g. documenting the source and history of data for validation or security; making dataflow more transparent to the user to improve explainability and trust; attribution of data sources to provide error traces and recreate or reverse changes on data. Here, we concentrate on using provenance information to improve visualizations and providing interactive interfaces with low to no additional code.

A. Overview and Running Example

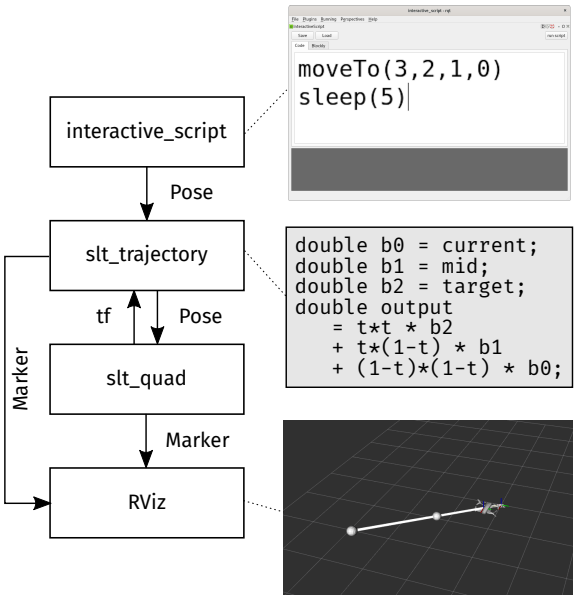

Fig. 1. An example application: the user inputs waypoints, a trajectory is calculated and executed by a simulated quadcopter. The quadcopter and trajectory are visualized (non-interactive) using RViz.

As our running example, we look at a small ROS application, as shown in Figure 1: the user can enter some waypoints into a script editor (interactive_script). These target poses are then sent to a minimal trajectory planner (slt_trajectory).

Here, a trajectory is planned from the simulated quadcopter's current pose to the received target pose. To interpolate between these poses, a quadratic bezier curve is used. The interpolated poses are then sent at a rate of 50Hz to a virtual quadcopter (slt_quad) that simulates the movement. The trajectory planner and quadcopter node both publish additional visualizations: the planned trajectory and its control points as well as a quadcopter model depicting the simulated quadcopter's current pose. The bottom picture in Figure 1 shows a screenshot from RViz with these non-interactive visualizations.

In order to create a more interactive interface, it is not sufficient to implement *interactive_marker* and replace the visualizations in the trajectory planner and the quadcopter node. When the control points of the trajectory or the quadcopter are dragged in RViz, the trajectory and the text in the editor should adapt accordingly. However, the trajectory planner has no access to the waypoint data or the text buffer as they were sent from the script node. The interactive markers can also not be created by the *interactive_script* node, as it lacks domain knowledge which planning algorithm is used. By calling a service on each other node to return the trajectory or set values in the text buffer, this problem can be solved at the cost of a more complicated node interface.

Making the simulated quadcopter interactive is a much more complicated problem: when the quadcopter is dragged around, the trajectory and editor content should change accordingly. The quadcopter node, however, has no information, how its received pose relates to the target pose in the script–i.e. if and how the data was changed or transformed–which makes a sensible change of the target pose in the editor impossible.

We propose a general mechanism to access and manipulate data across ROS nodes. When a value change on a received message is requested, the data is traced back to its node of origin and the data source is changed to a value that produces the desired value in the next received message.

B. Research Questions

Motivated by this problem of tracking the origin of data items and its application to visualization, we pose the following research questions:

- RQ1 What additional information is needed to preserve the origin and change history of data items from the data source across the ROS node graph?
- RQ2 How can we enable changing these data sources from any part of the ROS application?
- RQ3 How can we hide the necessary bookkeeping for this tracking from a ROS developer?
- RQ4 How can we use provenance-annotated messages to infer interactive user interfaces from non-interactive data?
- RQ5 What is the overhead in message size and performance of this provenance tracking?

C. Contributions

To answer the aforementioned questions, this paper makes the following contributions that are explained in more detail in the following sections:

RQ1: We wrapped the ROS Message types and added a header that contains provenance data and allows tracking the origin and history of ROS messages through the ROS graph. For each field and each list element of arbitrarily nested ROS messages, the origin node, an id and all previous operations on it are tracked.

RQ2: A *SourceChange* message to make data changes on the origin of tracked messages across ROS nodes is defined. The node of origin maintains a list of all known source locations, each of which is backed by a parameter to enable changes at runtime.

RQ3: We introduce a C++ support library that eases the transition to, and working with, provenance-annotated data. A generic wrapper type around values tracks all changes and manages provenance information. An extended *TrackedNode* class handles receiving and applying *SourceChange* requests.

RQ4: By using the provenance information of the non-interactive visualization messages, the callback of a similar interactive marker is connected to a source change on the visualization data's node of origin. We implemented such a node as part of our example: *slt_visualization* attaches interactive controls to visualization messages if provenance information exists.

RQ5: We measured the message size and performance overhead for our value wrapper. Results for our example show a $3\times$ to $6\times$ message size overhead and 1.8µs tracking overhead per arithmetic operation. While this is clearly too expensive for numeric applications or tracking all messages, it is practical to use for selected messages and data flows.

D. Outline

In the following sections, we will first discuss the foundations this work is based on as well as other related work. In Section IV, the provenance tracking across the ROS graph and the accompanying support library is presented in more detail, answering *RQ1-RQ3*. In Section V, we will then show a possible application of this tracking as outlined in our example: interactive interfaces are derived from non-interactive visualizations. This section will also discuss the limitations of our approach, answering *RQ4*. We evaluate the performance and size overhead in this example in Section VI (*RQ5*), before drawing a conclusion and outlining possible future improvements to our work.

II. Foundations

Our work applies provenance tracking to ROS applications. This section gives a quick overview and introduction to some of the frameworks and techniques used in the remainder of this paper.

A. ROS

The Robot Operating System (ROS) [12] is a robotics middleware and an ecosystem of standard interfaces, tools and third-party components that robotics applications can build upon. Its strongly component-based and decentralized architecture enables fast prototyping, higher resilience, simplified

testing and reconfiguration at runtime of applications. Each ROS component–called node–is often assigned to a separate process. Communication between nodes is done through two basic concepts: *topics* and *services*. While topics create a unidirectional, named and typed many-to-many communication channel, a service is an interface for a named, synchronous remote procedure call. Several libraries offer more sophisticated, often domain-specific communication patterns on top of topics and services: *tf2* [5] handles spacial and temporal coordinate transformations by collecting transformations from different sources and joining them into a transformation tree. The client library then offers transformations between any two coordinate frames in this tree. *Actions* [13] model long running asynchronous, preemptible commands that provide continuous feedback on their progress. *InteractiveMarkers* [6] create interactive 3D interfaces, that can assign callbacks to mouse events on these 3D objects. Each of these, more sophisticated, communication patterns needs their own server and client library that creates and connects necessary topics and services and provides a clean programming interface that hides internal complexity.

B. Interactive Markers

Non-interactive visualizations can be easily created in ROS by sending a *visualization_msgs/Marker* message that is then interpreted and visualized in RViz or other compatible tools [10][16]. A marker mainly consists of a basic shape, pose, color and scale. The possibility to seamlessly integrate multiple visualizations from different sources make RViz a very powerful visualization and debugging tool. Early on in the development of ROS the need to create interfaces for interactive manipulation arose, leading to the development of interactive markers [6]. An *InteractiveMarker* consists of several controls, each of which can use one or more Markers as their representation, an interaction mode to select its possible interactions and behavior and a callback to execute a custom reaction to this interaction.

C. Provenance Tracking

Provenance is a century old concept of noting and preserving the ownership and creation history of works of art. Provenance is also an integral part in many areas of scientific research, noting and publishing documentation of the process of empirical research to enable reproduction. More recently, provenance found its application in computer science [7] mainly in two areas: machine learning, where provenance is used to track the origin and data used for pre-trained models and security, where the concept of data provenance [1] is used to formally describe the disclosure of confidential data through functions. Provenance tracking at runtime can also be used as an alternative to static data flow analysis, sacrificing completeness for easier implementation.

Runtime value tracking can be implemented either directly as a feature of the language runtime, by wrapping the value or by observing the system and watching for data changes. However, implementation as a feature of the language runtime requires changing the language and observation of the system requires runtime introspection capabilities provided by the language. A value wrapper does not have specific language requirements but can come at the cost of major syntactic changes to the program to implement it.

The value type is extended–in the interpreter or through a wrapper–by the origin of the data, e.g. file and offset of a number literal.As this provenance data is assigned to the *value* it is automatically carried along when bound to a variable or used as a function argument. If a tracked value is used in an expression, the resulting value has the same origin. Additionally, the relation between the operand and the result of the expression can be tracked, which makes it possible to reverse its evaluation and link a change of the result to a corresponding change of the origin [3].

Reversing the evaluation of binary operators such as + leads to obvious problems as the relation between the argument tuple and the result is not bijective and therefore not invertible. It is, however, possible to create a heuristic that injectively maps the result to the arguments, e.g. by changing only one operand and keeping the other operand constant.

III. Related Work

Value tracking at runtime to record provenance information–similar to our approach–was previously proposed and applied to various domains:

By instrumenting *toString* methods in Java code, Tiny Structure Editors [8] can automatically generate graphical interfaces to change complex data structures. In contrast to our work, it is limited to textual string representations and unable to track the origin across multiple processes.

The code portal editor *Inline* [2] integrates similar tracking of values into the live evaluation of expressions in the code to create a portal for the result that is editable and can be applied to its source. The editor supports live evaluation for its own functional language, that integrates the value tracking in the interpreter itself to enable value tracking for any value without needing any annotations or changes by the user. Using C++ instead, we cannot introspect the program at runtime to hide the tracking completely, so we rely on our value wrapper implementation.

Similarly, the code editor used in our example–*interactive_script* [15]–uses a custom implementation of the Lua language to automatically annotate value literals with their location in the text buffer and can draw an interactive live preview in RViz. This paper improves it by tracking values across multiple nodes and implements the interactive marker interface in a more generic way.

Sketch-n-sketch [4][11] combines provenance tracking with an interactive graphical preview for different use cases. Bidirectional editing is implemented for HTML pages, generated from a functional, declarative language and for a SVG editor amongst other things. The value tracking differs in tracking multiple sources for a value and deferring application of the source change heuristic to let the user select where and how these changes should be applied in the editor.

IV. Provenance Tracking in ROS

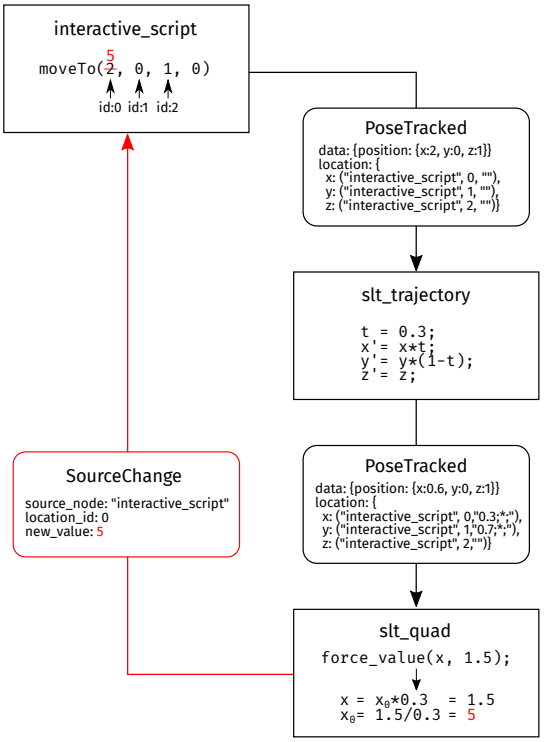

Fig. 2. *interactive_script* sends a provenance-annotated *PoseTracked* message to *slt_trajectory*. It sends a new message to *slt_quad*, based on the received data. *slt_quad* can change the source in *interactive_script* by requesting a *SourceChange*.

To solve this problem of communicating changes to the appropriate node without cluttering the nodes' interfaces, we adapt the concepts of source location tracking and bidirectional evaluation to the ROS communication primitives. Implementing this system requires additional provenance information in the received messages:

- The origin (node and source location therein) of the received data.
- The relation between the received value and its source.
- A way to change the data at its source.

Under the assumption that the control flow remains unchanged and no aliasing occurs, the evaluation can then be reversed with a new value and the original value can be set to produce the desired new value after reevaluation, e.g. in the next message that is received.

We use the *Pose* messages from the example as shown in Figure 2 to introduce the pattern that can then be applied to other messages in the quadcopter example.

A. Communication Pattern and Tracking Operations

The script node (*interactive_script*) creates a message from the waypoint entered by the user. The ranges in the text buffer containing the values are identified by an id that is added to the message as additional provenance information. For each data field, the name of the node, the location id and a serialized expression is recorded. Then, *slt_trajectory* does some calculations on the received data–e.g. to interpolate between waypoints–creating a new *PoseTracked* message. Note, that the origin node for each field is still *interactive_script* but the expression changed to reflect the calculations on these values by *slt_trajectory*. In *slt_quad*, the received data should be changed, e.g. to react to user interaction: the user dragged the simulated quadcopter from $x = 0.6$ to 1.5 and wants to change the waypoint in the editor accordingly. Using the recorded history of the pose, the relation between the simulated quadcopter's position and the initial waypoint is known. By first applying the inverse expression history of the old value–$2 * 0.3 = 0.6$–to the new value–$1.5/0.3 = 5$–and then sending a *SourceChange* message to the value's node of origin, the correct literal in the editor is replaced. Resending the waypoint and recalculating the trajectory now places the quadcopter in the desired position.

To implement this communication pattern, the following operations on values are necessary:

create_location: This operation creates a changeable code location for a value. As the compiled C++ code–and encoded data, e.g. constants–of the node cannot be changed at runtime, a parameter is created, initialized to the argument value and given an id. Alternatively, the value can originate from an external source, e.g. a file. In this case, instead of a parameter, a getter/setter pair of functions is used to read and write the location. The source location of the call to *create_location*–file, line and column–is used to identify subsequent calls from the same location that then return the current value of the same parameter or the result of the corresponding getter function.

force_value: By using the *force_value* operation, the origin of a tracked value can be changed. The inverse of all previous arithmetic operations on the old value is applied to a target value which evaluates to a new value for the origin. A *SourceChange* message is then created and sent to the old values node of origin. Assuming that an inverse of all previous arithmetic operations exists, that the control flow remains unchanged for the new value and there are no aliasing effects due to the origin appearing multiple times in the evaluation history, the next value received from the same origin evaluates to the target value at the *force_value* call.

apply_source_change: A node must apply all received source changes to the appropriate locations. The location id is used to determine the appropriate parameter or setter function for external locations.

reevaluate: In some cases, it is necessary to poll for changes to the source of a value. Using the reevaluate operation, the origin of the value–or each of its fields recursively, in case of a structured data type–is queried for its current value. Then all recorded operations on it are reapplied. This can be used if the value is updated infrequently or not at all but it should still reflect the current state of its origin.

arithmetic operators: If an arithmetic operator is applied to a tracked value, in addition to its normal semantics, the operation must be logged to enable its inversion. In case of binary operators, we track only one source, even if both operands

provide provenance information. Some simple heuristics are used in this case: we prefer the origin of the left operand except if this would lead to problems inverting the operation e.g. a multiplication with 0. We deliberately decided against tracking all sources, which would allow deferring this decision to the application of source changes: tracking only a single source reduces the size of the provenance header of a message and simplifies its serialization.

selection of fields and list elements: ROS messages are structured data that can contain other messages or lists. Each field or list element needs to be tracked separately as they might stem from different sources. Selecting a field of a tracked message or a list element of a tracked list should again return a tracked value with the appropriate slice of the provenance information.

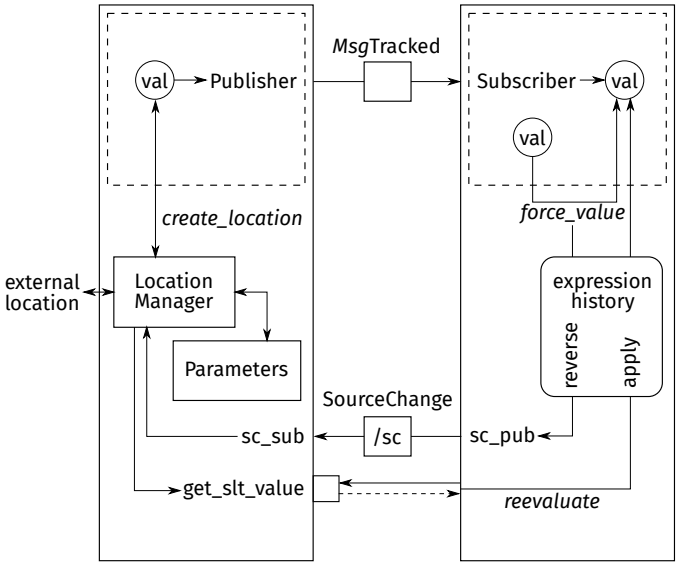

Fig. 3. The full *rosslt* communication pattern. Additional communication and operations to support value changes across nodes for any data received via a tracked message.

We use a wrapper around messages and values to intercept the arithmetic operators and field selection in particular. This way, all bookkeeping on the provenance information is hidden from the user and changes to the nodes are kept minimal. The management of node locations, applying and creating source changes is done in a wrapper derived from ROS2's node class. Simply by changing a node's base class and wrapping relevant values that should be tracked, a node can be made provenance aware; tracked data can be changed across node borders without cluttering the node's interface or additional node complexity.

Figure 3 shows the full interface of two provenance-aware nodes. The user code resides in the dashed boxes at the top, while the *rosslt* library hides additional topics, services and management of changeable source locations. This pattern is applicable to any two nodes that exchange a tracked message, i.e. a message containing the additional provenance header to preserve provenance data for each of its message fields.

B. The rosslt Support Library

To help this transition towards provenance tracking enabled nodes and to reduce the necessary code changes to a minimum, we created the rosslt library for ROS2. Currently, the library supports only nodes written in C++; as C++ is more restrictive in terms of operator overloading and runtime type introspection, we assume a Python port of the library is straightforward and feasible. Using aspect-oriented programming techniques, the library could also omit the value wrapper, eliminating nearly all code changes.

The library consists of two parts: a templated wrapper class for values that tracks and updates provenance information and an extension of ROS2's *rclcpp::Node* class that manages changeable locations, applies source changes and provides a method to change tracked values.

Tracked value wrapper: By overloading the arithmetic operators in templated value wrapper and then delegating it to the wrapped type, the wrapper can intercept the evaluation of expression. It serializes argument values and operators to a string so that they can be reapplied (*reevaluate* operation) or inverted (*force_value* operation).

A tracked value can be cast to an untracked value to allow its usage as an argument to not yet converted functions. It can also be cast to an appropriate tracked ROS message type to minimize the overhead when transferring it to other ROS nodes.

Listing 1
ACCESSING ELEMENTS OF COMPLEX TRACKED TYPES.

```
1 Tracked<vector<int>> vec = ...
2 Tracked<int> i = vec[0];
3 vec.push_back(5);
4
5 struct IntPair {int a,b;};
6 Tracked<IntPair> ip = ...
7 Tracked<int> x = GET_FIELD(ip, a);  // x = ip.a;
8 SET_FIELD(ip, b, i);                // ip.b = i;
```

Tracked structured data–e.g. ROS messages–and tracked lists need special handling: accessing a field of the data structure or a list element should inherit a corresponding slice of tracking information from the parent struct or list. Listing 1 shows some of these operations. The tracking header of the parent struct or list manages provenance data for each field or list item separately. The *GET_FIELD* and *SET_FIELD* macros cannot be avoided as C++ does not allow overloading the '.' operator.

TrackedNode ROS interface: As shown in Figure 3, the ROS interface extends the *rclcpp::Node* class, subscribes to the global source change topic and offers a service to request values for reevaluation. To support value changes at runtime, changeable source locations must be explicitly defined and are backed by either a parameter or a pair of functions to set and get the value of an external location. When the control flow reaches the source location for the first time, the parameter or external location is set to the value of the given literal. Each

subsequent evaluation of the source location uses the current value of the parameter or external location instead, making the source location virtually changeable at runtime.

V. Generic, Interactive Interfaces

The provenance information in tracked values is similar to the information encoded in callback functions to events of interactive markers: the event callback connects a change in the visualization to a change in internal state. The programmer is responsible to provide the correct relation between the visualization and the internal state as well as means to access it, through calling setter methods, services etc. This code is often complex, highly error prone, and hard to maintain if the changed data is not directly accessible – e.g. if it is managed by another node – or the relation is unknown – e.g. due to other nodes processing or transforming the data in between. The provenance header and the support library provide these capabilities: the origin of and relation between visualization and its source is contained in the provenance header of a *MarkerTracked* message and can be changed by calling the *force_value* method with the marker's changed position. Therefore, the provenance header and *rosslt* library can be seen as a structured and generic access mechanism that can replace specialized callback code to access and change values in many cases. While the generic callback that uses our library can obviously express less than a hand coded custom callback function, it avoids the aforementioned problems: it requires no additional user code and automatically creates a correct relation to the source data by tracking the data flow at runtime.

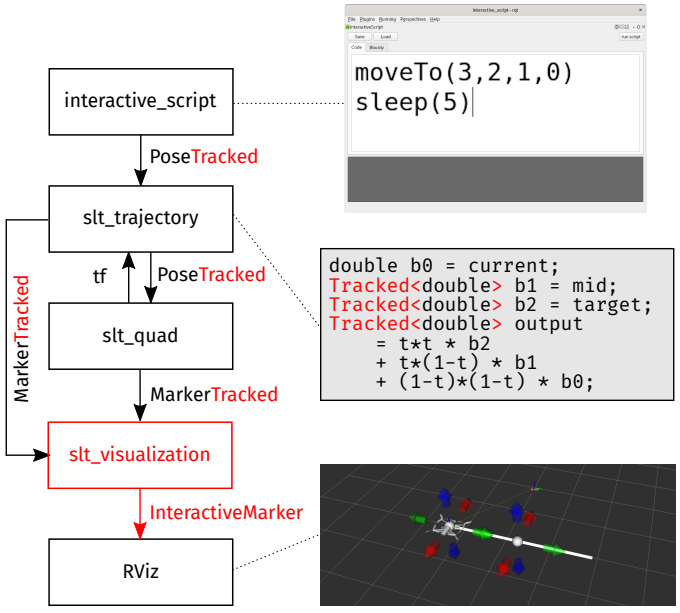

Fig. 4. Changes (highlighted in red) to the example to enable provenance tracking and interactive visualizations. An additional node (slt_visualization) creates interactive markers from tracked non-interactive markers.

Using provenance tracking and our support library, we can transform our example from before as shown in Figure 4. The message types of all nodes are extended to include provenance information and all relevant code paths use the *Tracked* value wrapper. Apart from these changes, the code stucture and and node interfaces remain unchanged. A new node *slt_visualization* is introduced. It automatically translates non-interactive visualizations with provenance information (*MarkerTracked*) to interactive markers, displayed in RViz. If the x, y or z coordinate of the position of the marker contain provenance information, a control for the respective axis is created. Accordingly, markers with partial or incomplete information have reduced modes of interaction and can not be moved along all three axes.

A. Generality and Limitations

The effect of moving the generated interactive marker is determined only by its provenance information. While the generated interfaces correctly manipulate the source data, the resulting behavior of the markers can be very unintuitive: multiple source locations might influence the marker position but the change is mapped to only one of these. The implemented left bias on arithmetic operators–the origin of the left operand is used for the result preferentially–can be exploited to influence and hint the generated markers towards a preferred semantic by rearranging calculations. Similarly, by explicitly discarding provenance information a data source can be excluded from changes to the result. An intermediate node, like *slt_trajectory* in the example, can cast a value that should not be considered for a change to its untracked base type, to exclude data sources based on local domain knowledge. Implementing such hints contradicts our goal to create interfaces without additional code changes and is–in our experience–seldom necessary for applications of similar complexity to the presented example's.

There are no restrictions on the ROS message type: messages can be arbitrarily nested and may contain scalars or lists of any basic data type. However, messages with many values–such as large lists or matrices of values–will suffer from a high size overhead as explained in Section VI. Often, message data is transformed into other container types i.e. lists are transformed into vector types of a linear algebra library. In these cases the underlying scalar value type can be changed to its tracked counterpart i.e. to track changes through matrix operations. Due to the high number of basic arithmetic operations, the additional bookkeeping and tracking will reduce the performance significantly and might not be feasible.

The implemented value tracking approach can easily capture dataflow but is oblivious to control flow. All changes to a values source happen under the assumption that this does not change control flow. This might not be the case, if any value that is dependent on the changed value is used in a condition of a control flow statement. In most cases the assumption holds true: even if the condition depends on the value it detects edge cases most of the time which our–often small–output changes hardly trigger.

Aliasing is another problem that can cause incorrect changes–that is, changes that do not reevaluate to the desired output–at the source of a value. The value change is tracked back to a single source to change. If the value depends more than once on this source, reversing the evaluation accounts for only one of these occurrences. This problem can be often avoided by using specialized functions, e.g. by using the *power* function to square a value instead of a multiplication.

Similarly, changing the source of a value often causes side effects. The changed value might be used in entirely different contexts, which change as well. This is an intended effect; the change should not create special cases and keep the structure of the program intact and consistent.

VI. Evaluation

All examples used and presented in this paper are available in our public repository [14].

In order to estimate the overhead of our value tracking implementation, we measured the message size overhead and the time to deserialize and apply a values expression history, as these directly limit the applicability in ROS applications. Additionally, we show and compare the code of parts of the minimal trajectory planner used in the example. This demonstrates the practicability of introducing and using provenance tracking in new or existing codebases.

A. Message Size and Expression Deserialization Overhead

The size overhead of tracked messages varies depending on how many fields have associated provenance data, the length of its expression history and the name of its origin. The minimal size overhead is 8 bytes per message, if no provenance data is available. In our example we observed a size increase of around 60 to 100 bytes per tracked field. For a *PoseTracked* message this led to a 6 times larger peak message size (60 to 359 bytes, 17.8kB/s at 50Hz). For the *MarkerTracked* messages sent to visualize the trajectory, we observed an around 5 times larger message size (786 to 4034 bytes).

TABLE I
Benchmark results for tracked values.

	50 random operations	time per operation
Tracking	90.5µs	1.81µs
Reapply	21.5µs	0.43µs
Reverse	81.2µs	1.62µs

To measure the time needed to create, reverse and apply the serialized expression history of a tracked value, we created a simple benchmark that tracks and times applying and reversing 50 random binary arithmetic operations. Results are shown in Table I (Core i5, 2.3GHz). *Tracking* includes applying the operation to the value and recording the operation in its expression history. *Reapply* measures the time needed to apply the history to another value, e.g. as a part of the *reevaluation* operation. *Reverse* measures the time needed to reverse the expression history and then applying it, e.g. as a part of the *force_value* operation.

While the current increase in message size limits the adoption of provenance tracking to selected topics and low message rates, we expect that a more sophisticated implementation–transmitting only changes in the provenance header and compressing it–can reduce this overhead significantly. Tracking the expression history can create a high overhead for numeric tasks due to string operations when serializing these arithmetic expressions. For less computation intensive tasks–like in the example–the overhead is negligible compared to message transport between nodes.

B. Code Comparison

Listing 2
Excerpt from slt_trajectory; the necessary changes to implement provenance tracking are highlighted in red.

```
struct Bezier {
  Tracked<geometry_msgs::msg::Point> b0, b1, b2;
  double tau;

  Tracked<geometry_msgs::msg::Point>
    get_trajectory_point(double t) const
  {
    if (t <= 0)   return b0;
    if (t >= tau) return b2;
    t /= tau;

    auto x = t * t * GET_FIELD(b2,x)
      + 2 * t * (1-t) * GET_FIELD(b1,x)
      + (1-t) * (1-t) * GET_FIELD(b0,x);
    auto y = t * t * GET_FIELD(b2,y)
      + 2 * t * (1-t) * GET_FIELD(b1,y)
      + (1-t) * (1-t) * GET_FIELD(b0,y);
    auto z = t * t * GET_FIELD(b2,z)
      + 2 * t * (1-t) * GET_FIELD(b1,z)
      + (1-t) * (1-t) * GET_FIELD(b0,z);

    Tracked<geometry_msgs::msg::Point> result;
    SET_FIELD(result,x,x);
    SET_FIELD(result,y,y);
    SET_FIELD(result,z,z);
    return result;
  }
};

...

auto p = current_trajectory.get_trajectory_point(t);
Tracked<geometry_msgs::msg::Pose> pose;
SET_FIELD(pose, position, p);
publisher_->publish(
  static_cast<rosslt_msgs::msg::PoseTracked>(pose));
```

The introduction of provenance-annotated data flows into an existing codebase should be as simple and unintrusive as possible. Listing 2 highlights the necessary changes in a representative excerpt of the *slt_trajectory* node that is used throughout the running example. The changes can be grouped into three categories:

- Type changes to use the *Tracked* value wrapper to hold additional provenance information.

- Usage of the *GET_FIELD* and *SET_FIELD* macros to access fields of messages, as C++ forbids overloading the '.' operator.
- Type casts to automatically translate tracked values into appropriate ROS messages.

All these changes are syntactic and do not require restructuring or refactoring of the code, reducing the risk of introducing unwanted semantic changes or bugs. As the structure of the code is unchanged, the code complexity does not increase according to most popular metrics.

VII. Conclusion

We applied the concept of provenance tracking to ROS applications to link values to their origin across multiple nodes. The necessary provenance information–the node of origin, the location and history of applied operations–is transmitted using a provenance message header, which introduces a significant, but still practicable, increase in message size and processing performance. The proposed communication pattern then uses an additional *SourceChange* message and an inversion of the value expression history to communicate changes to a value back to its origin. The *rosslt* hides this additional communication and bookkeeping where possible and exposes methods to create changeable source locations, trigger changes and reevaluate values. While *Marker* messages are used to display non-interactive visualizations in RViz, the additional provenance header of *MarkerTracked* messages can be used to automatically generate interactive visualizations. This is demonstrated by creating an interface from visualizations of a planned trajectory and a simulated quadcopter: the trajectory and quadcopter can be dragged around to change the waypoints for the quadcopter in an independent *interactive_script* node that publishes tracked positions.

While our implementation strictly limits itself to the ROS ecosystem, the concepts and communication pattern can be applied to other component based architectures. The ROS implementation demonstrates the feasibility of our approach and showcases how existing patterns and tools can benefit from additional provenance information.

VIII. Future Work

We plan to support Python nodes in the future. The far superior introspection capabilities compared to C++ will allow creating a seamless tracking of values without requiring code changes. Other limitations of our current implementation, like lacking support for the vast amount of nodes still using ROS1, missing library support for transmitting tracked values using services and actions and reevaluation of structured data being limited to ROS message types can be tackled as well.

Our long term vision is a complete provenance tracking throughout the ROS application: from static launch configuration files or mission descriptions as data source to human-robot interaction interfaces–e.g. hand guiding of a robot–to generate source changes.

References

[1] Umut A Acar et al. "A core calculus for provenance". In: *Journal of Computer Security* 21.6 (2013), pp. 919–969.

[2] Alexander Breckel and Matthias Tichy. "Live Programming with Code Portals". In: *Workshop on Live Programming Systems. Workshop on Live Programming Systems - LIVE'16*. 2016.

[3] James Cheney, Umut Acar, and Amal Ahmed. "Provenance traces". In: *arXiv preprint arXiv:0812.0564* (2008).

[4] Ravi Chugh et al. "Programmatic and direct manipulation, together at last". In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 341–354.

[5] Tully Foote. "tf: The transform library". In: *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. IEEE. 2013, pp. 1–6.

[6] David Gossow et al. "Interactive markers: 3-d user interfaces for ros applications". In: *IEEE Robotics & Automation Magazine* 18.4 (2011), pp. 14–15.

[7] Paul Groth et al. *An architecture for provenance systems*. Tech. rep. University of Southampton, 2006.

[8] B. Hempel and R. Chugh. "Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–5.

[9] Dave Hershberger, David Gossow, and Josh Faust. *RViz, 3D visualization tool for ROS*. [19-01-2021]. URL: http://wiki.ros.org/rviz.

[10] Burkhard Hoppenstedt et al. "Debugging Quadrocopter Trajectories in Mixed Reality". In: *6th International Conference on Augmented Reality, Virtual Reality and Computer Graphics (SALENTO AVR 2019)*. Lecture Notes in Computer Science. Springer, Apr. 2019.

[11] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. "Bidirectional evaluation with direct manipulation". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–28.

[12] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[13] Higor Barbosa Santos et al. "Control of Mobile Robots Using ActionLib". In: *Robot Operating System (ROS)*. Springer, 2017, pp. 161–189.

[14] Thomas Witte. *rosslt*. [19-01-2021]. URL: https://github.com/sp-uulm/rosslt.

[15] Thomas Witte and Matthias Tichy. "A Hybrid Editor for Fast Robot Mission Prototyping". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE. 2019, pp. 41–44.

[16] Antonio Zea and Uwe D Hanebeck. "iviz: A ROS Visualization App for Mobile Devices". In: *arXiv preprint arXiv:2008.12725* (2020).