Splash on ROS 2: A Runtime Software Framework for Autonomous Machines*

Hwancheol Kang, Cheonghwa Lee, Wooyoung Choi and Seongsoo Hong

Abstract— **ROS has been completely refactored and evolved into ROS 2 to address the ever-increasing software complexity of autonomous machines. While it has become a de facto standard software platform for autonomous machines, ROS 2 still has room for improvement. It lacks support for essential features such as real-time stream processing, mode change, sensor fusion and rate control for output shaping. Moreover, its programming model is at a lower level than what most programmers would expect.**

In this paper, we carefully analyze the shortcomings of ROS 2 and propose to augment it with the Splash programming framework. In doing so, we host Splash on top of the ROS 2 software stack by conducting model conversion between Splash and ROS 2. We refer to the end result as Splash on ROS 2. To show its viability, we conducted a case study with a robot arm controller performing DNN-based object detection and motion planning. The case study qualitatively confirms that Splash on ROS 2 relieves the programming burden on developers, increases the software development productivity and improves the quality of the software.

I. INTRODUCTION

Developing software for autonomous machines such as mobile robots, drones and self-driving cars is becoming more and more difficult, largely because autonomous machines are increasingly adopting deep neural network-based machine learning algorithms [1]. It is well-known that DNN-based applications tend to introduce serious complexities and operational complications into an autonomous machine for various reasons. First, they must coexist and seamlessly interact with conventional time-driven and/or event-driven real-time control software even though they are data-driven by nature [2]. Second, they need a complicated hardware platform consisting of heterogeneous multicore processors, graphical processing units (GPU) and neural network accelerators (NNA) to meet the ever-increasing computing power demand [3]. Third, DNN-based applications need a distributed computing platform that can deal with a wide variety of dispersed intelligent sensors and actuators. Lastly, they must correctly react to diverse sensor fusion scenarios [4].

This gives rise to a versatile runtime software framework supporting various programming abstractions, particularly suitable for DNN-based applications [5]. Such runtime software framework must be able to effectively separate application software developers from the already complex distributed hardware platform in order to hide implementation details from the developers and eventually improve software development productivity, robustness and reliability.

The robot operating system 2 (ROS 2) is one of the most successful runtime software frameworks for autonomous machines [6]. ROS 2 including its predecessor ROS has been widely used for thousands of robot designs in both industry and academia. It is still becoming more and more popular [7].

Despite such a great success, ROS 2 is still an on-going community project; in fact, it is the result of the complete refactoring of ROS. Now, ROS 2 can support real-time publish-subscribe communications and swarm operations among multiple collaborating robots via distributed join and leave. It also significantly improves the robot system reliability thru the decentralized node structure [8][9].

Although ROS 2 has many other enhancements over its predecessor, it still has room for improvement, especially from the DNN perspective. To address the four desired features that ROS 2 lacks, we propose to augment ROS 2 with the Splash programming framework we have developed at Seoul National University [2][4].

Splash is a graphical programming language that supports programmers in developing diverse applications for autonomous machines. The benefits of Splash are mainly four-fold. First, it provides an effective programming abstraction that supports the real-time stream processing and sensor fusion frequently appearing in DNN-based applications. Second, it enables programmers to specify genuine, end-to-end timing constraints and monitor the violation of such constraints at runtime. Third, it provides basic and yet crucial utilities such as exception handling and mode change to name a few. Finally, it can aid programmers with performance optimization and tuning during system implementation.

While Splash was originally up and running on a DDS-based middleware hosted on Linux [2][4], we retarget Splash to ROS 2 to take advantage of a plethora of features and packages that have already existed in ROS 2. In doing so, we enumerate programming model differences between Splash

* This research was supported by Hyundai Robotics. (No. 0668-20200206) and Industrial Strategic Technology Development Program (10079961, "Development of a deterministic DCU platform with less than 1μs synchronization for autonomous driving system control") funded By the Ministry of Trade, Industry & Energy(MOTIE, Korea).

Hwancheol Kang is with Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea (e-mail: hckang@redwood.snu.ac.kr).

Cheonghwa Lee is with Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea (e-mail: chlee@redwood.snu.ac.kr).

Wooyoung Choi is with Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea (e-mail: wychoi@redwood.snu.ac.kr).

Seongsoo Hong is with Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea. (corresponding author to provide phone: 82-2-880-8357; fax: 82-2-871-5974; e-mail: sshong@redwood.snu.ac.kr).

and ROS 2 and then conduct model conversions to reconcile the differences. We also identify the shortcomings of ROS 2 in supporting DNN-based applications and augment ROS 2 with Splash to rectify them. We call the end result *Splash on ROS 2*.

In this paper, we present the design of Splash on ROS 2 and show a case study that we perform with a robot arm controller performing DNN-based object detection and motion planning. We intend to qualitatively assess the utility and viability of the proposed approach. Our case study clearly shows that Splash on ROS 2 is more effective than ROS 2 alone, in terms of software development productivity and reliability. In fact, Splash drastically helps reduce development time and effort due to the following reasons. First, it offers for developers an intuitive and easy-to-understand programming model based on a data flow graph known as the Kahn process network [10]. Second, it supports model-based software development where the code generator automatically produces skeleton code and meta-data files from a given graphical Splash program with textual annotations [2][4]. Third, it provides developers with many built-in modules dealing with essential mechanisms such as mode change, sensor fusion and control output shaping [2][4]. Such modules, given in the form of the Splash client and runtime libraries, fundamentally block the possibility of errors that might occur if programmers repeatedly wrote code for such mechanisms. Finally, it provides a feature called the *build unit* that can help developers automate the software build and deployment process while focusing on optimizing the concurrency and parallelism of the resultant system.

II. BACKGROUND

We propose to augment the programming capability of ROS 2 with the Splash programming language. This requires the model conversion from the Splash language constructs into the ROS 2 programming and execution features. To help readers understand the model conversion, we give a brief but essential account of both Splash and ROS 2.

A. Splash Programming Language

This subsection gives a quick overview of the Splash programming language that is a summary from [2][4]. We refer the interested readers to [2][4] for more details of the Splash's diverse language constructs.

A Splash program is in essence a directed graph that consists of nodes and edges. Figure 1 shows a sample Splash program drawn via the Splash schematic capture tool. This program consists of the perception and planning subsystems used in our case study of this paper.

In the Splash terminology, a node and an edge of a directed graph are called a *component* and a *channel*, respectively. A component is either an *atomic* component or a *composite* component. A composite component is also called a *factory*. A factory encapsulates a subgraph has external ports. Figure 1 demonstrates two factories interconnected with each other. Atomic components are further classified into four different types: (1) a processing component, (2) a source component, (3) a sink component and (4) a fusion operator.

A component has stream input ports and stream output ports with the exception of the source and the sink component. The stream output port of an upstream component is connected to the stream input port of a downstream component and such connection creates a channel.

The most important language construct in Splash is the *processing component* since it performs computation on input data items and produces transformed data items as an output. Moreover, a processing component serves as a building block for constructing a Splash program.

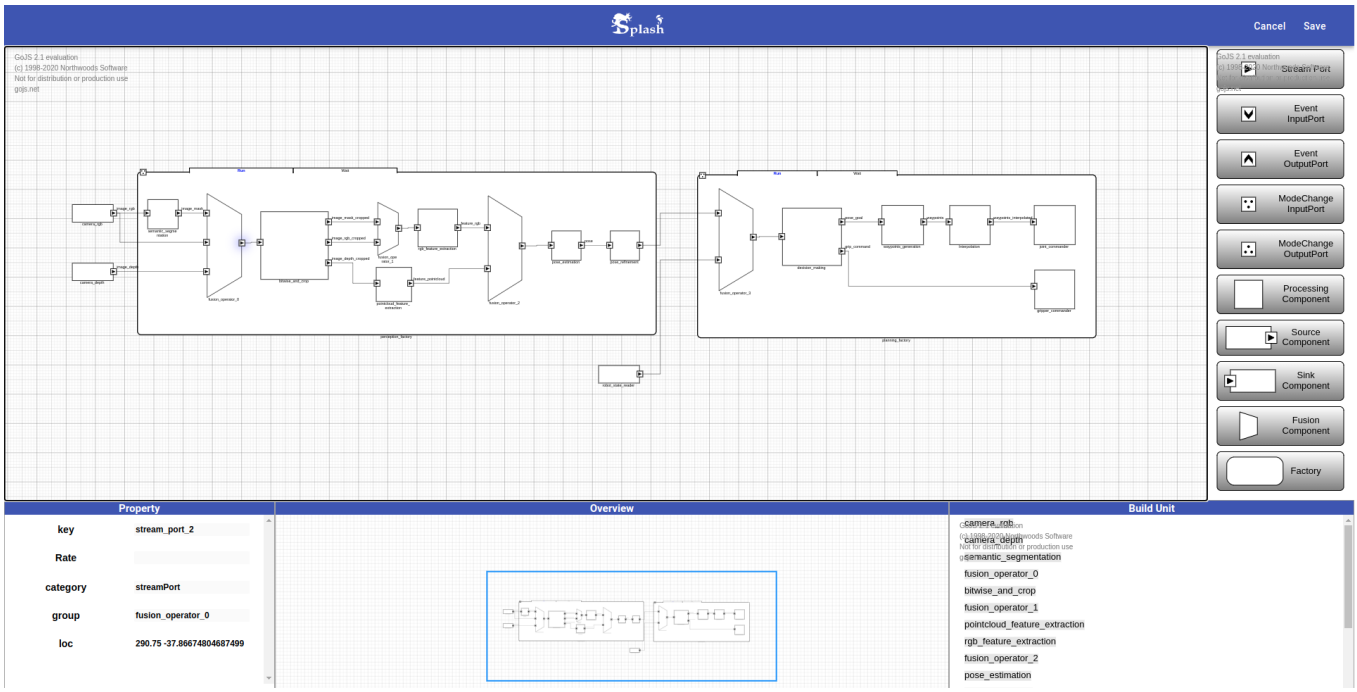

Figure 1. A Splash program for object detection and motion planning.

Splash supports three types of *ports*: (1) stream input/output ports for sending and receiving stream data, (2) event input/output ports for delivering events and (3) mode change input/output port for passing mode change signals.

Input and output port types are the subtypes of the port type. Each port type is associated with one of three port interfaces: stream, event and mode change port interface. The output port and the input port connected by a channel must share the same port interface. Each port interface has a data type for data items it sends or receives. A data type can be a primitive data type or a composite data type.

A *channel* is a delivery path for steam data. It is represented with a solid line from a stream output port to a stream input port. The Splash language mandates that data items always go through a channel in the order of their birthmarks. Such *in-order delivery semantics* significantly reduces the amount of work done at a downstream component. In order to store data items on a channel until they are consumed by a downstream component, an imaginary FIFO queue is assumed. In Splash, an imaginary FIFO queue is on the stream input port of the downstream component, instead of the stream output port of the upstream component. The fan-in of a channel is restricted to one, but the fan-out of a channel can be greater than one. When a channel is connected to multiple input ports, all data items generated from an output port are replicated to all the input ports of downstream components. When a data item is delivered to an input port, the arriving data item is handled by a specified callback function.

A *clink* is a delivery path for events and mode change signals. It is represented by a dotted line from an output port to an input port. Unlike the channel, both fan-in and fan-out of a clink can be greater than one.

Time is a first-class entity in Splash in the sense that the creation time of a live data item is always preserved in its timestamp, allowing it to be monitored in comparison with an abstract global clock. We refer to the timestamp that carries the creation time as a *birthmark*. If an intermediate process in a Splash program generates a data item, it inherits the birthmark from its oldest ancestor. To enforce and monitor various timing constraints, Splash compares the birthmark of a data item with the current system time.

Splash supports three types of end-to-end timing constraints [11].

(1) A *freshness constraint* on a single sensor value: It bounds the time it takes for a sensor value to flow through the system. A sensor value will become useless if it exceeds the freshness constraint since sensor values get stale with time.

(2) A *correlation constraint* on multiple sensor values: It limits the maximum time difference among a group of distinct sensor values used for sensor fusion.

(3) A *rate constraint* on an output port of a process: It limits the number of output data items produced per second. A rate constraint is a soft real-time constraint in the sense that the Splash runtime tries its best to minimize the jitter between consecutive data items on the same channel, but does not guarantee that the stream output port is jitter-free.

Programmers can explicitly annotate such timing constraints at application development time via Splash's language constructs. The Splash runtime will raise an exception if it detects the violation of an annotated timing constraint at runtime.

B. ROS 2 Programming and Execution Model

A ROS program is also represented with a directed graph called a *ROS graph*. A node in a ROS graph is a building block of a ROS program and an edge is a communication link between two nodes. Some links are called *topics* when they represent publish-subscribe communications; others are called *services* when they denote client-server communications. A ROS node can receive three types of messages: (1) subscription, (2) client and (3) service messages. A node reacts to an incoming message by activating a callback corresponding to the type of the message [7][8][9]. Figure 2 shows a sample ROS graph [12].

For execution, one or more nodes in a ROS graph can be grouped into an operating system process. Such a process is associated with a special thread called an *executor*. The executor implements the ROS execution model in that a callback is invoked to process an incoming message. ROS 2 provides two built-in executors: (1) a sequential executor that executes callbacks in a single thread and (2) a parallel executor that distributes the pending callbacks across multiple worker threads obtained from the thread pool [7]. ROS 2 also supports arbitrary user-defined executors. Recently, the callback-group-executor has been introduced and is gaining popularity since it allows programmers to prioritize incoming messages via real-time profiles [13].

Despite many improvements made for ROS 2, it still has shortcomings when it comes to real-time DNN inference in an autonomous machine.

(1) It provides only little support for specifying end-to-end timing constraints or monitoring dynamic timing violation. In ROS 2, configurable timing constraints can appear only at the communication level.

(2) The ROS 2 programming model is still at a lower level than what most programmers would expect. For instance, a ROS graph directly exposes the details of publish-subscribe communications. In addition, ROS 2 lacks support for frequently appearing features such as real-time stream processing, mode change, sensor fusion and rate control for output shaping.

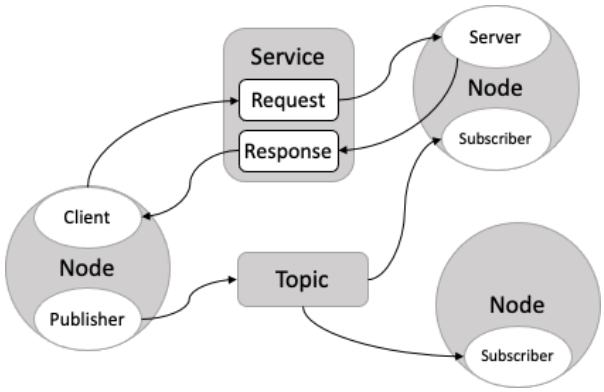

Figure 2. A ROS graph.

TABLE I. SPLASH - ROS 2 MAPPING TABLE

Splash Entities	ROS 2 Entities
Processing Component	Node
Source Component	Node
Sink Component	Node
Fusion Operator	Node with built-in callback
Stream Input Port	Subscriber
Stream Output Port	Publisher
Event Input Port	Subscriber
Event Output Port	Publisher
Mode Change Input Port	Subscriber
Mode Change Output Port	Publisher
Channel	Topic
Clink	Topic

(3) ROS 2 offers little support for automated deployment that enables users to remotely install, start, stop and monitor applications. As a result, users need to manually deploy applications to distributed machines.

III. MODEL CONVERSION BETWEEN SPLASH AND ROS 2

In this section, we explain how Splash on ROS 2 addresses the aforementioned shortcomings. In our approach, Splash sits on top of the ROS 2 software stack as shown in Figure 3. The Splash client and runtime libraries collectively realize the Splash framework and provide application programming interfaces (APIs) for Splash applications. They play the role of mapping the Splash programming entities to those of ROS 2 via the ROS APIs.

A. Splash to ROS 2 Mapping

Each of Splash language constructs is mapped to an entity of ROS 2 as specified in TABLE I. First, an atomic component is mapped to a ROS 2 node. A processing component can have a user-defined callback for each of its input ports. A fusion operator has a built-in callback whose role is to check if a fusion rule is satisfied and to trigger a fusion function whenever needed.

An output port and an input port are mapped to a publisher and a subscriber of ROS 2, respectively. A channel passes data items of a known data type from an output port to an input port.

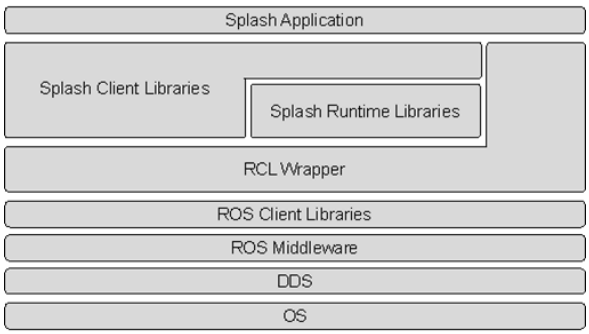

Figure 3. Layered architecture of Splash on ROS2.

A clink delivers events of a known data type. Both a channel and a clink are mapped to ROS 2 topics. They both possess a name and a data type. A channel has a user-defined data type and a clink is associated with a built-in data type.

B. Splash Client Libraries

The Splash client libraries (scl) provide a programming abstraction for Splash applications via APIs written in Python. These APIs are a set of wrappers for both the ROS client libraries Python wrapper (rclpy) and object-oriented classes for Splash entities.

Coding a Splash application is a two-step process: (1) skeleton code generation and (2) algorithm specification. The scl provide APIs in both steps. Skeleton code generation is done by the Splash code generator using schematic data produced by the schematic capture tool. In this step, port objects are created with channel information and callback functions are generated and registered to input ports. Rate constraints are enforced at rate-controlled ports. Component objects are created. Freshness constraints are enforced at source components. Fusion rules are implemented at fusion operators. The port objects are attached to the component objects. Build unit objects are created and the component objects are added to the build unit objects. The algorithm specification step is performed by programmers. They fill in their own logic inside callback functions associated with input ports.

In order to implement the mapping rules specified in TABLE I, each Splash entity class inherits from the matching ROS 2 entity class as shown in Figure 4. `Component` is a subclass of a ROS 2 node class. The ROS 2 node class contains properties and methods for managing publishers, subscribers and callback functions. Using the properties and methods, the scl implement methods for attaching and managing ports inside the component class.

`ProcessingComponent`, `SourceComponent` and `SinkComponent` inherit `Component`. These classes have a method for registering callback functions. When an input port object is attached to it, the component object registers the input port's callback function. `SourceComponent` has a method for enforcing a freshness constraint. `FusionOperator` also inherits `Component`. It has a method for implementing a fusion rule with an extrapolation handler. It then registers a built-in callback function for sensor fusion.

`InputPort` is a subclass of the ROS 2 subscriber class that is defined by a message type, a topic name and a callback function. Information on a channel or a clink is passed as an argument to the constructor of the `InputPort` object. `StreamInputPort` and `EventInputPort` inherit `InputPort`. They have a method for registering a callback function. `ModeChangeInputPort` lacks a callback registration method even though it inherits `InputPort`. Instead, a built-in callback function for mode change is automatically registered when the object is constructed.

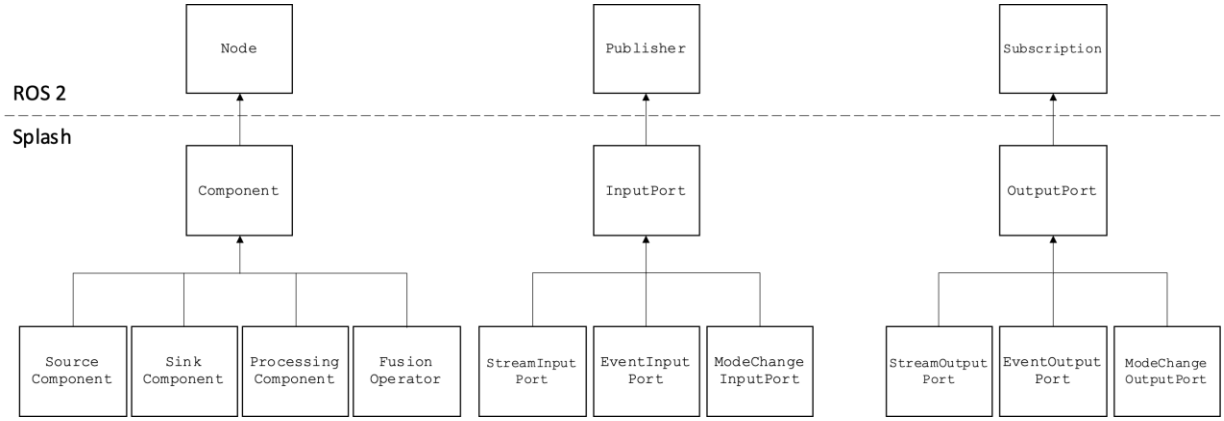

Figure 4. Class hierarchy among ROS and Splash classes

`OutputPort` is a subclass of the ROS 2 publisher class that is defined by a message type and a topic name. The publisher class has a method for publishing a message. `StreamOutputPort` inherits `OutputPort`. By extending the method for publishing, the scl implement a method for publishing a message simultaneously. The method also checks for the violation of a freshness constraint inside `StreamOutputPort`. When a `StreamOutputPort` object, which is attached to a `SourceComponent` object, publishes a message, a freshness constraint and a birthmark are added to the message. `StreamOutputPort` class has a method for enforcing a rate constraint. `EventOutputPort` and `ModeChangeOutputPort` inherit `OutputPort`. The former triggers an event with an event name. The latter obtains an event name and a name of the target factory for mode change from the graphical Splash program.

C. Splash Runtime Libraries

The Splash runtime libraries (srl) offer essential runtime functionalities such as (1) callback execution, (2) rate control, (3) sensor fusion and (4) mode change.

Callback execution is a functionality to coordinate the execution of callbacks registered to components. The srl implement the Splash executor that is similar to the callback-group-executor [13]. The Splash executor puts an emphasis on asynchronous event handling, message prioritization and in-order message delivery. It has two queues, one for data and the other for events. When a callback is registered to an input port, it is associated with a specific queue depending on the type of the input port. The Splash executor prioritizes the event queue over the data queue. Events are queued in the FIFO order while data items are stored according to the nondecreasing order of their birthmarks.

Rate control is a mechanism that prevents bursty data traffic by limiting the number of output data items that are generated per unit time [2]. When a `StreamOutputPort` object publishes a message with a rate constraint, it stores the message into a queue of the associated rate controller, instead of immediately calling the ROS 2 publishing method. The rate controller has a queue and a timer for each rate-controlled output port. On each periodic invocation, the rate controller looks up the output queue to find the oldest message. If there is at least on fresh message in the queue, the rate controller calls the ROS 2 publishing method.

Sensor fusion is a mechanism that estimates information about nearby situation by processing data from multiple sensors. To support sensor fusion, Splash offers a fusion operator with which programmers specify a fusion rule, a correlation constraint and an extrapolation handler via a well-defined graphical and textual interface. The fusion operator internally runs the optimal sensor fusion algorithm that we came up with in our previous work [4]. The algorithm is guaranteed to generate a tuple of multiple sensor data that satisfies the user-specified fusion rule if one exits. A `FusionOperator` object has a method that implements the fusion rule. When the method is called, it first creates a queue for each stream input port attached to the object. Whenever a message arrives at any of the input ports, a built-in callback function enqueues the message into the corresponding queue and runs the sensor fusion algorithm to get a valid tuple. This tuple is published through the corresponding output port. If the algorithm cannot find any valid tuple, a user-defined extrapolation handler is called instead.

Mode change is a mechanism that activates or deactivates a selected subgroup of components in a factory that has multiple execution modes [2].The srl offer a module called the mode manager to support the mechanism. In response to a mode change request, an event triggering method is called with a target factory name and an event name through a `ModeChangeOutputPort` object. Then the mode manager publishes a mode change message to components in the target factory. The components that received the message are activated or deactivated through the execution of a built-in callback function.

IV. CASE STUDY

In order to demonstrate the utility of Splash on ROS 2 as a versatile programming framework for autonomous machines, we have conducted a case study in which a robot arm controller performs DNN-based object detection and motion planning. Specifically, our case study is composed of a robot arm controller and a simulator. The simulator includes a dynamics engine that reflects the physical characteristics of the robot and a visualizer that expresses robots, objects and sensors in three dimensional space. As shown inFigure 5, the robot arm picks

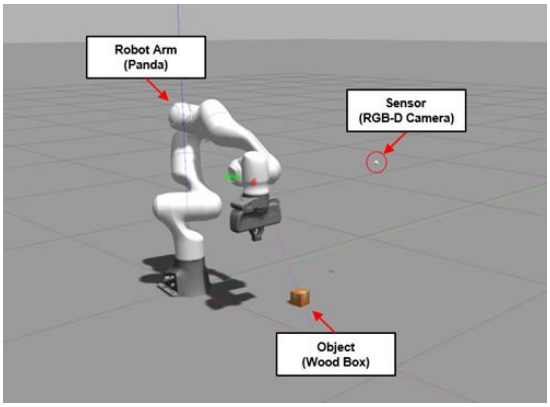

Figure 5. Overview of the case study.

up the dropped object at a random location and moves it to the designated location.

The robot controller program we had drawn using the Splash schematic capture tool already appeared in Figure 1 in Section II. Apparently, the graphical program looks quite straightforward: It is composed of a perception subsystem and a planning subsystem. As stated previously, the Splash code generator automatically produces skeleton code from the schematic data that the schematic capture tool generates. Unlike our expectation, the resultant code forms a very complex source tree. To make the source code management easier, the Splash code generator packages the entire source tree into a ROS 2 package as shown in Figure 6. At the top directory is a ROS 2 package. It contains a Python package, which in turn contains a Splash package. It is a subpackage that contains the hierarchy of the skeleton code.

A factory in a Splash program is converted to a subpackage. Each component contained in a factory becomes the factory's subpackage. A component subpackage has a module for creating a `Component` object and attaching `Port` objects to the `Component` object. Its code is given in Figure 8.

All input ports attached to a component become the component's subpackage that contains a callback module for each input port. The callback module is named after the channel to which the input port is connected. The callback function inside the module is the place where programmers insert their own code as shown in Figure 9.

Figure 6. Hierarchical source tree for generated skeleton code.

A build unit is an individual software entity for build, deployment and execution on a distributed system. To create a `BuildUnit` object and add related `Component` objects to it, Splash offers the build unit module as shown in Figure 10. The main module, an entry point of a Splash program, imports the build unit module and calls the `run` method of the `BuildUnit` object to start off a Splash program, as in Figure 11.

In addition to constructing a hierarchical code structure as explained above, the Splash code generator produces code for the connection logic between components in a Splash program. Figure 7 shows the ROS graph representing the various ROS nodes and connections between them.

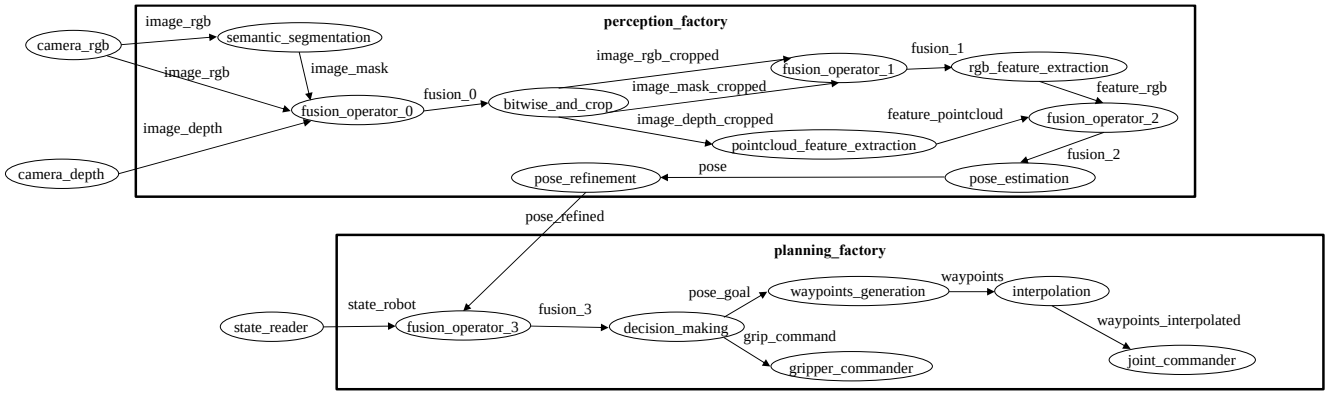

Figure 7. ROS graph in execution.

```
#perception_factory/bitwise_and_crop/bitwise_and_crop.py
from scl.components import ProcessingComponent
....
from sensor_msgs.msg import Image
from .input_ports import fusion_0
def build():
    component = ProcessingComponent()
    component.attach(StreamInputPort("fusion_0", fusion_0.callback))
    component.attach(StreamOutputPort("image_rgb_cropped", Image))
    component.attach(StreamOutputPort("image_depth_cropped", Image))
    return component
```

Figure 8. Source code for component module.

```
# perception_factory/bitwise_and_crop/input_ports/fusion_0.py
from sensor_msgs.msg import Image
def callback(component, msg):
    output = Image()
     .... # user logic
    component.write("image_mask_cropped", output)
```

Figure 9. Source code for input callback module.

```
# build_units/default_build_unit.py
from scl.build_unit import BuildUnit
....
def run():
    build_unit = BuildUnit("default_build_unit")
    build_unit.add(camera_rgb.build())
    build_unit.add(camera_depth.build())
    build_unit.add(semantic_segmentation.build())
    build_unit.add(fusion_operator_0.build())
    ....
    build_unit.run()
```

Figure 10. Source code for build unit module.

```
# default_build_unit_exec.py
from .splash.build_units import default_build_unit
def main():
    default_build_unit.run()
```

Figure 11. Source code for the main module.

The case study clearly reveals that Splash on ROS 2 provides developers with a useful and effective programming abstraction that can relieve the programming burden on developers, increase the software development productivity and improve the quality of the software. Such benefits are due to the three features of Splash on ROS 2: (1) The Splash progrmming language offers the appropriate level of abstraction so that programmers can avoid a low-level programming details such as communicating nodes and publish-subscribe communications. (2) The Splash code generator produces the skeleton code and the connection logic between communicating nodes so that developers can get away with writing tedious and error-prone house-keeping code in their programs. (3) The Splash client and runtime libraries can help developers avoid coding complex but frequently appearing mechasnisms such as mode change, sensor fusion and rate control.

V. CONCLUSION

We proposed Splash on ROS 2 as a versatile runtime software framework for autonomous machines. It offers support for essential features such as real-time stream processing, mode change, sensor fusion and rate control for output shaping. These features are exposed to programmers as language constructs that they use in Splash applications. The Splash toolset automatically generates high-level language code according to the semantics of those language constructs. The generated code makes use of APIs provided by the Splash client libraries (scl) and the Splash runtime libraries (srl). These libraries are implemented on top of the ROS 2 software stack and collectively perform model conversion between Splash and ROS 2.

We conducted a case study with a robot arm controller performing DNN-based object detection and motion planning. The case study confirmed that Splash on ROS 2 relieves the programming burden on developers, increases the software development productivity and improves the quality of the software.

There are several future research directions along which our programming framework can be extended. First, we are planning to include service orientation thru which new or updated services are dynamically deployed to autonomous machines and become immediately available to users [14][15]. Second, we plan on conducting in-depth case studies in the autonomous vehicle domain where the benefits of our programming framework stand out. Finally, we will evaluate the performance and run-time overhead of Splash-based systems with extensive experiments. The results look promising.

REFERENCES

[1] S. Wang, A. Pathania and T. Mitra, “Neural network inference on mobile SoCs,” *IEEE Design & Test*, vol. 37, issue. 5, pp. 50–57, Oct. 2020.

[2] S. Noh and S. Hong, “A graphical programming framework for an autonomous machine,” in *proc. The 16th International Conference on Ubiquitous Robot (UR)*, 2019, pp. 660-666.

[3] J. Kim, P. Shin, M. Kim and S. Hong, "Memory-aware fair-share scheduling for improved performance isolation in the Linux kernel," *IEEE Access*, vol. 8, pp. 98874-98886, Jun. 2020

[4] S. Noh and S. Hong, “Programming language support for multisensor data fusion: the Splash approach,” in *proc. The 17th International Conference on Ubiquitous Robot (UR)*, 2020, pp. 429-436.

[5] A. Hellmund, S. Wirges, Ö. Ş. Taş, C. Bandera and N. O. Salscheider, "Robot operating system: A modular software framework for automated driving," in *proc. IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016, pp. 1564-1570.

[6] Open Source Robotics Foundation (OSRF), “ROS 2,” [online] Available: https://github.com/ros2.

[7] D. Casini, T. Blaß, I. Lütkebohle and B. B. Brandenburg, “Response-time analysis of ROS 2 processing chains under reservation-based scheduling,” in *proc. 31st* Euromicro Conference on Real-Time Systems (ECRTS), 2019, pp. 6:1-6:23.

[8] J. Kim, J. M. Smereka, C. Cheung, S. Nepal and M. Grobler, “Security and performance considerations in ROS 2: a balancing act,” Computing Research Repository, arXiv preprint arXiv:1809.09566, Sep. 2018.

[9] Y. Yang and T. Azumi, "Exploring real-time executor on ROS 2," in *proc. IEEE International Conference on Embedded Software and Systems (ICESS)*, 2020, pp. 1-8.

[10] G. Kahn, “The semantics of a simple language for parallel programming,” in *proc. IFIP Congress*, 1974, pp. 471-475.

[11] R. Gerber, S. Hong and M. Saksena, “Guaranteeing real-time requirements with resource-based calibration of periodic processes,” *IEEE Transactions on Software Engineering*, vol. 21, no. 7, pp. 579-592, 1995.

[12] A ROS graph, [online] Available: https://docs.ros.org/en/foxy/_images/Nodes-TopicandService.gif

[13] R. Lange, “Mixed real-time criticality with ROS 2 - the callback group-level executor,” in *ROSCon 2018, Lightning Talk*, 2018.

[14] S. Fürst and M. Bechter, "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform," in *proc. 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, pp. 215-217.

[15] M. Stepanović, J. Jovičić, G. Stupar and M. Kovačević, "Application lifecycle management in automotive: Adaptive AUTOSAR example," in *proc. IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 2018, pp. 1-4.