Splash: A Graphical Programming Framework for an Autonomous Machine

Soonhyun Noh and Seongsoo Hong

Abstract— **Autonomous machines have begun to be widely used in various application domains due to recent remarkable advances in machine intelligence. As these autonomous machines are equipped with diverse sensors, multicore processors and distributed computing nodes, their software architecture has become more and more complex. This leads to a demand for a new programming framework that has an easy-to-use programming abstraction. In addition, such framework requires support for genuine end-to-end timing constraints and run-time detection of their violation. In this paper, we present a graphical programming framework named Splash that explicitly addresses the programming challenges that arise during the development of an autonomous machine. We set four design goals to solve these challenges. First, Splash must provide an effective programming abstraction that supports the stream processing of an autonomous machine. Second, it must enable programmers to specify genuine, end-to-end timing constraints and monitor the violation of such constraints. Third, it must support exception handling, mode change and sensor fusion. Finally, it must support performance optimization and tuning during system implementation. We present the syntax and semantics of the key language constructs of Splash and show how we achieve our design goals. To show the utility of our programming framework, we have written an adaptive cruise control (ACC) application in Splash as an example. We also present the findings that we have obtained during the development process of the ACC application using Splash.**

I. Introduction

With recent remarkable advances in machine intelligence, autonomous machines have been actively developed and begun to be widely used in various application domains. Representative examples of such machines include drones, robots and self-driving cars. Often times, they are equipped with diverse sensors for perception, localization and positioning [1]. They also include high performance multicore processors for intelligence and microcontrollers for real-time control [2].

These hardware components are interconnected via onboard networks inside autonomous machines [3]. Due to the heterogeneous, distributed and multicore nature of the underlying computing platform, the software architecture of an autonomous machine has become more and more complex. Its complexity has reached a point where programmers must resort to a versatile programming framework that has an easy-to-use programming abstraction that can hide implementation details, and supports a model-based code generation capability. Additionally, such a framework needs to support genuine, end-to-end timing constraints such as a freshness, correlation and rate constraint, which means that it must support the detection of the aforementioned timing constraints as well as handling of its exceptions.

Quite a few graphical programming frameworks have been widely used in practice, particularly for automatic control and signal processing domains. Such frameworks include Simulink and RTMaps [4][5]. Also, several academic programming frameworks such as Ptolemy II exist for research purposes [6]. Except for RTMaps, most of the existing frameworks were designed and developed for a broad range of reactive embedded systems.

Simulink is one of the most representative commercial programming frameworks. It can support both time-driven and event-driven data processing. Unfortunately, it does not fulfil our design goals; it does not support end-to-end timing constraints that must be considered when implementing an autonomous machine; it does not offer language constructs for exception handling and sensor fusion; and it provides little or no support for the performance optimization and tuning of a resultant system to run on a distributed multicore computing platform.

RTMaps is well suited for the development of a system that has to deal with multiple sensors and actuators like an autonomous machine. It has many features in common with our approach. RTMaps supports time as a first-class entity and records a timestamp on each data item. As result, it can offer a method for specifying and handling freshness and correlation constraints. It allows programmers to write applications in both data and time-driven programming styles. However, it has several limitations that makes it unfit for our design goals. First, RTMaps does not consider a rate constraint in an explicit manner. Thus, programmers must independently develop their own rate control mechanism, creating spaces for error. Second, it does not support concurrency models explicitly, leaving programmers with the responsibility of thread creation and synchronization. Third, RTMaps does not offer a language construct for asynchronous event notification and handling. Finally, RTMaps lacks support for imperative programming such as mode change and exception handling.

This research was supported by Samsung Electronics Co., Ltd., Korea (No. 0115-20180001) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R7117-16-0164, Development of wide area driving environment awareness and cooperative driving technology which are based on V2X wireless communication).

Soonhyun Noh is with the Electrical and Computer Engineering Department, Seoul National University, Seoul, Korea (e-mail: shnoh@redwood.snu.ac.kr).

Seongsoo Hong is with the Electrical and Computer Engineering Department, 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).

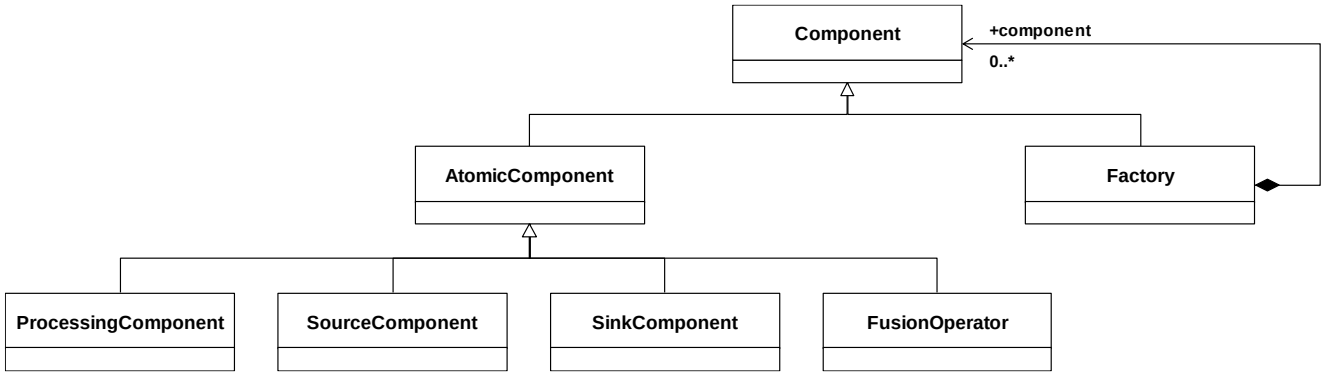

Figure 1. Hierarchy of Splash components.

Ptolemy II is an academic programming framework capable of supporting a wide variety of process network models. Thus, programmers can write an application utilizing several different models at the same time. Ptolemy II offers rich support for imperative programing such as mode change and exception handling. However, Ptolemy II lacks support for real-time stream processing except Ptide. It is an experimental model and allows a freshness constraint to be specified on a sensor value [7]. But it does not support a rate constraint or a correlation constraint. Like RTMaps, Ptolemy II lacks a concurrency model or a thread-to-core allocation mechanism inside a process. Simply, it maps each process to a Java thread and delegates thread scheduling to the underlying operating system.

In this paper, we present a graphical programming framework named *Splash* to explicitly address such programming challenges that arise during the development of an autonomous machine. For Splash, we have the following design goals in mind. First, Splash must provide an effective programming abstraction that supports the stream processing of an autonomous machine. Second, it must be able to specify genuine, end-to-end timing constraints and monitor the violation of such constraints. Third, it must support exception handling, mode change and sensor fusion that make the most critical engineering features of an autonomous machine. Lastly, Splash must support performance optimization and tuning during system implementation.

We present the syntax and semantics of the key language constructs of Splash and show how we achieve our goals. To do so, we organize this paper as follows. In Section II, we present the underlying timing semantics of Splash and three end-to-end timing constraints. In Section III, we explain in detail the core language constructs of Splash. Section IV shows an example program written in Splash to show the utility of the Splash language constructs, along with lessons learned. Section V concludes this paper.

II. TIMING SEMANTICS AND END-TO-END TIMING CONSTRAINTS

Time is a first-class entity in Splash. Reading the time in a Splash program is supported by an abstract global clock that is possibly implemented via distributed local clock synchronization [8]. In Splash, a data item that flows through the system carries the timestamps of noticeable event occurrences associated with it. The primary timestamp required for a data item is its own creation time. Often, this time stamp is created through a sensor. We call this the *birthmark* of a data item.

In Splash, every live data item is assigned with its own birthmark. The birthmark can also be inherited from its oldest ancestor if the data item is generated by an intermediate process. Enforcing time constraints involves comparing the birthmark of a data item with the current time.

Splash supports three types of genuine, end-to-end timing constraints [9].

(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 a sensor value gets 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 defines the number of output data items produced per second. A rate constraint is a soft real-time constraint in a sense that the Splash runtime tries its best to minimize the jitter between consecutive data items on a channel, but cannot guarantee that the stream output port is jitter-free.

Developers are allowed to explicitly annotate these three types of timing constraints via language constructs in a Splash program. The Splash runtime will raise an exception if it detects the violation of an annotated timing constraint at runtime.

The Splash programming framework is designed to support real-time stream processing on a distributed, and possibly multicore computing platform for an autonomous machine. The timing semantics explained in this section clearly lays foundation for the semantics of the language constructs of Splash. In the next section, we elaborate upon them.

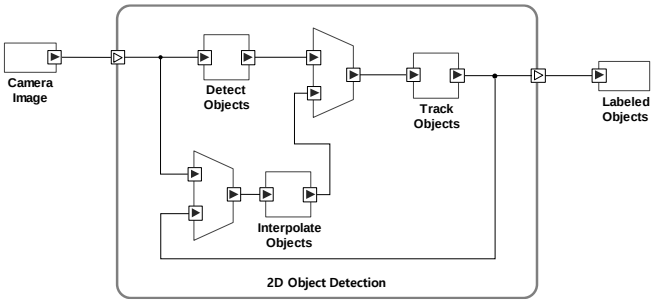

Figure 2. Sample Splash program: 2D object detection.

III. Language Constructs of Splash

A Splash program consists of processing nodes and edges between two processing nodes. In the Splash terminology, a node and an edge are called a *component* and a *channel*, respectively. A component in a Splash program is either an *atomic* component or a *composite* component. A composite component is also called a *factory*. 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. Figure 1 shows the hierarchical relationships among the diverse Splash components in the UML diagram format.

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. Figure 2 shows a sample Splash program that consists of various components, channels and ports.

A. Processing Component

The most essential language construct in Splash is a processing component since it actually performs computation on input data items and produces output data items. Surely, a processing component serves as a building block for constructing a Splash program. Figure 3 shows the graphical representation of a processing component with two stream input ports and two stream output ports.

In order to exploit parallelism explicitly from the underlying operating system and computing platform, Splash offers a multithreaded process model. It also provides a container model to aid developers in performing thread-to-processor allocation. In the multithreaded process model, a processing component consists of a group of Splash threads we call *sthreads*. An sthread is a logical entity of independent execution inside a processing component. Figure

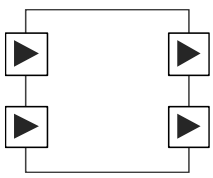

Figure 3. Graphical representation of a precessing component.

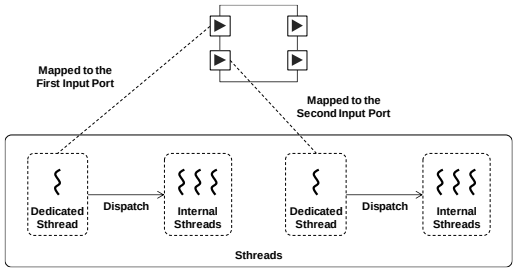

Figure 4. Process and its sthreads.

4 shows a processing component example where a dedicated sthread is attached to each port and internal sthreads serve as worker threads as in the concurrent server design pattern [10].

As a sthread is an abstract entity, it needs to be mapped to a thread of an underlying operating system during the system implementation process. Since the thread is an execution entity, it must eventually run on a specific core of a specific processor on a specific computing node. Such mapping involves thread-to-core allocation. To facilitate this process, Splash offers an allocation entity called a *container*.

B. Port

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. Each port type has a unique graphical symbol as shown TABLE I.

A stream output port is connected to a stream input port via a *channel*. We differentiate from a channel a connection between event ports or a connection between mode change ports. Such connections carry control signals or discrete data items, instead of a data stream. We refer to them as *control links* or *clinks* for short.

Input and output port types are the subtypes of the port type as described in Figure 5. Each port type is associated with one of three port interfaces: stream, event and mode change port interfaces. Clearly, an output port and an input port connected by a channel or a clink must share the same port interface. Figure 6 shows the three port interfaces. As in the figure, each

TABLE I. Graphical Symbols for Ports

Port Type	Input	Output
Stream Port		
Event Port		
Mode Change Port		

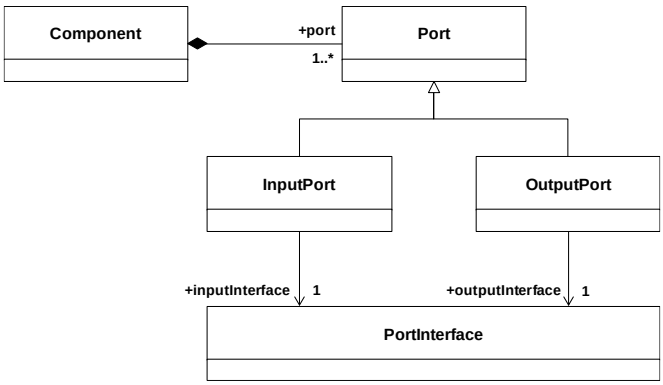

Figure 5. Input and output ports as subtype of port.

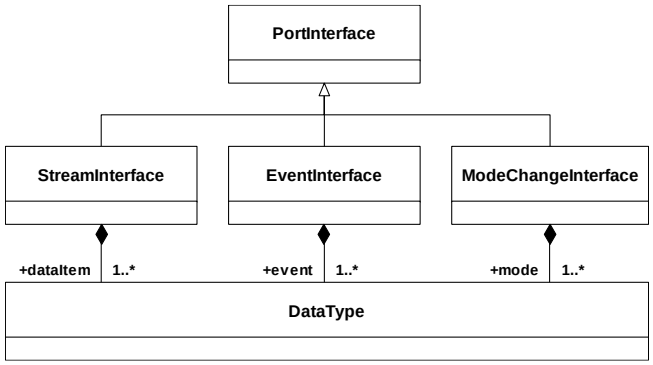

Figure 6. Hierarch of port interfaces.

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. Splash supports five primitive data types: (1) a Boolean type, (2) an integer type, (3) a real type, (4) a character type and (5) a string type. Splash supports two composite data types: (1) arrays and (2) records.

Splash developers can annotate a rate constraint on a stream output port. As mentioned in Section II, a rate constraint is regarded as a soft real-time constraint; the Splash runtime tries its best to minimize the jitter between consecutive data items on a channel, but cannot guarantee that the stream output port is jitter-free.

C. Channel

A channel is a delivery path for steam data. It is represented by a solid line from a stream output port to a stream input port. Figure 7 (A) shows the graphical representation of a channel.

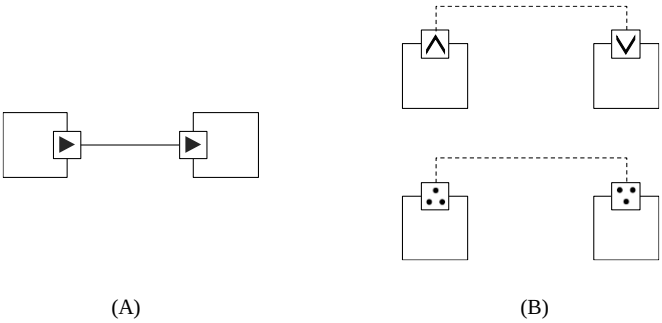

Figure 7. Channel and clinks.

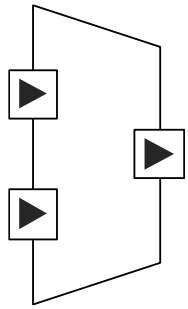

Figure 8. Fusion operator.

In order to store data items on a channel until they are consumed by a downstream component, a FIFO queue is used. In Splash, a FIFO queue is considered to be 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. Where a channel is connected to multiple input ports, all data items generated from an output port are replicated and enqueued into each of the FIFO queues on the input ports of downstream components.

D. Clink

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. Figure 7 (B) shows the graphical representation of a clink between event ports and a clink between mode change ports.

Like a channel, a clink uses a FIFO queue to store events or mode change signals. This queue is considered to be on an event input port or a mode change input port. Unlike a channel, both the fan-in and fan-out of a clink are restricted to one.

E. Fusion Operator

A fusion operator is a component that merges multiple stream data into a single stream data. It has multiple stream input ports and one stream output port. The graphical representation of a fusion operator is shown in Figure 8.

A fusion operator can be effectively used for sensor fusion in an autonomous machine. Programmers can annotate a correlation constraint on a fusion operator. When the fusion operator is triggered, it extracts a data item from each stream input port and build an output tuple in such a way that the correlation constraint is satisfied. If a fusion operator can

Figure 9. Source and sink component.

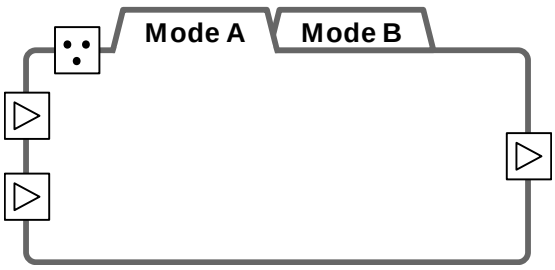

Figure 10. Factory.

generate multiple output tuples, it actually produces a tuple with the oldest data items. If such an output tuple cannot be created, the Splash runtime raises an exception.

F. Source Component

A source component is an atomic component that produces stream data items from a sensor. It has a single stream output port. Figure 9 (A) shows the graphical representation of a source component.

All data items produced from a source component must have its own birthmarks. The programmer of a source component is responsible for recording a birthmark. An exception is raised whenever a data item without a birthmark is found at runtime.

Programmers can annotate a freshness constraint on a source component. Such freshness constraint is automatically recorded on all data items generated by the source component. The Splash runtime checks whether a data item violates its freshness constraint each time it is enqueued into or dequeued from a FIFO queue on a channel. If a freshness constraint is violated, the data item is discarded immediately. Programmers may regard it as an exception and execute a handler.

G. Sink Component

A sink component is an atomic component that consumes stream data items and delivers each of them to an actuator. It has a single stream input port and no stream output port. The graphical representation of a sink component is shown in Figure 9 (B).

H. Factory

A factory is the largest building block of a Splash program. It contains a piece of a Splash program that serves as a subprogram in a procedural language. Splash distinguishes the stream port of a factory from the stream port of an atomic component by using a different symbol.

In Splash, a factory may have multiple modes of operations. In such case, a factory consists of as many alternative factories as the mode. Each alternative factory corresponds to a certain mode. Figure 10 shows a factory with two operation modes. Mode change is triggered by a mode change signal that arrives on the mode change input port of a factory. On each mode change, the Splash runtime processes all the current data items and empties all the FIFO queues inside the factory while blocking incoming data items and then starts a new mode.

IV. Example Program in Splash

To better illustrate the utility of Splash, we have written an adaptive cruise control (ACC) application in Splash. This application automatically adjusts the vehicle speed to maintain a safe distance from a front vehicle. We explain its overall application logic along with its timing constraints annotation.

A. Application Logic

Figure 11 shows the top-level factory of the application, labeled as `ACC`. Its inputs include a 2D image stream from a camera sensor, a set of 3D points from a LiDAR sensor and the current steering angle and speed of the ego vehicle. Its output is the target acceleration of the vehicle. The top-level factory consists of two sub-factories: (1) `3D object detection` and (2) `vehicle speed adjustment`.

The `3D object detection factory` is shown in Figure 12. We design this factory based on the algorithm in

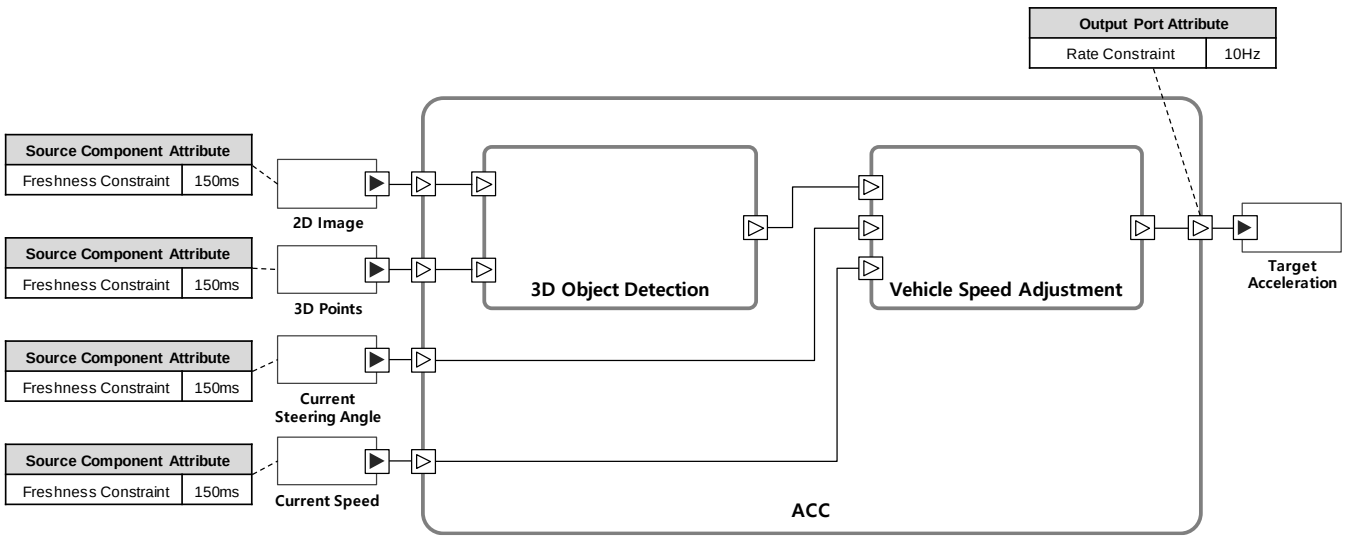

Figure 11. ACC factory.

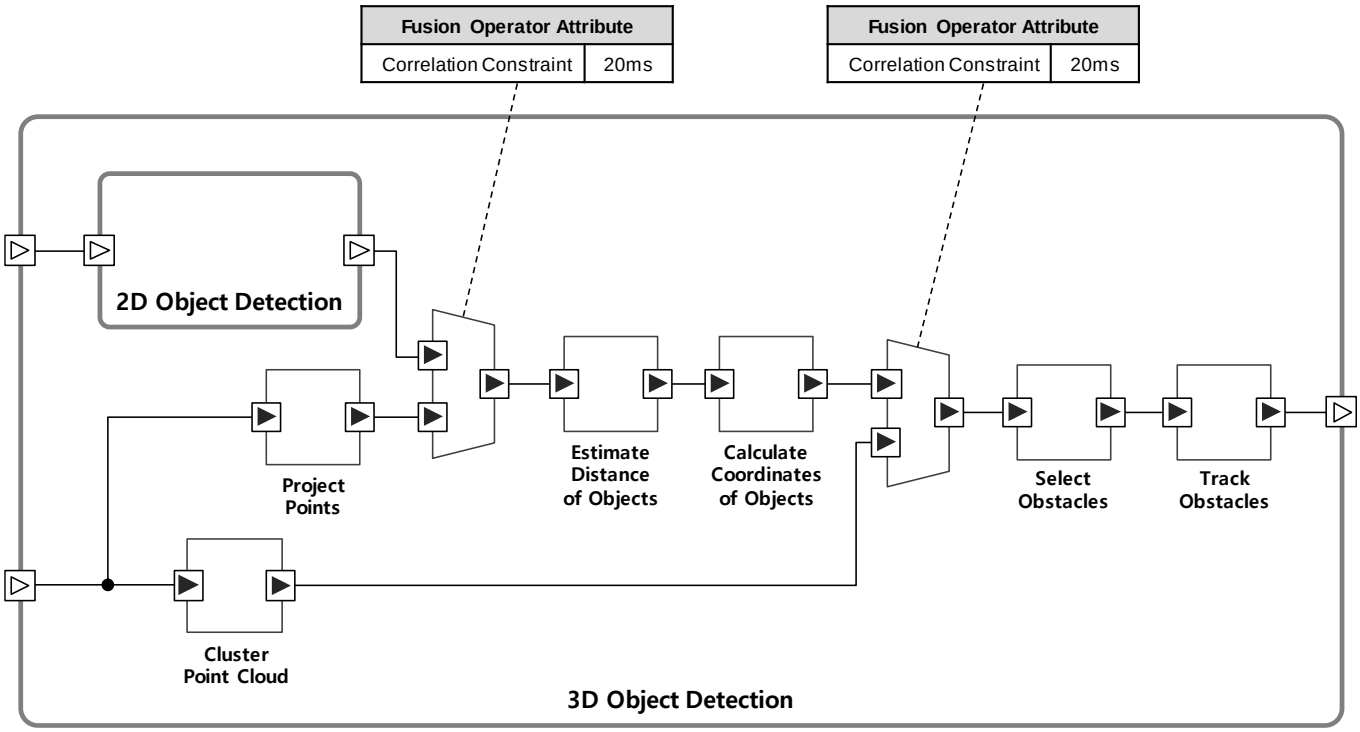

Figure 12. 3D object detection factory.

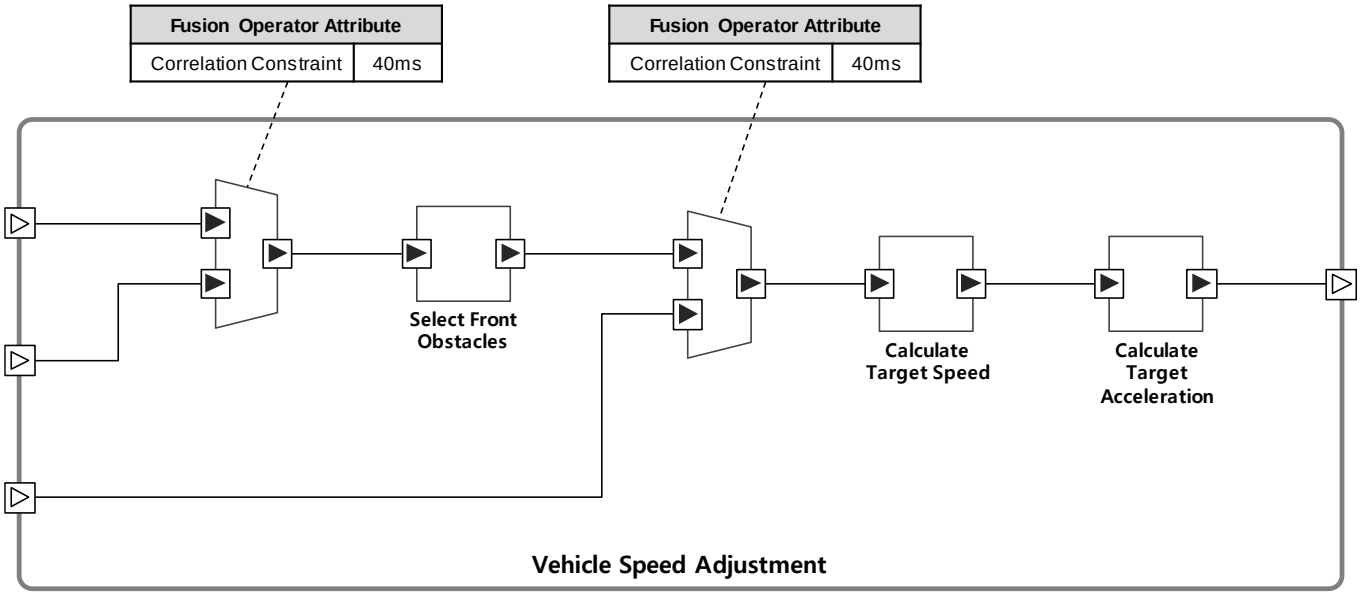

Figure 13. Vehicle speed adjustment factory.

[11]. The primary task of the factory is to detect all objects surrounding the ego vehicle, such as other vehicles, pedestrians and traffic lights. In doing so, it uses the 2D image stream and the 3D point cloud. As an output, it generates a set of surrounding objects with associated meta-data: class, position and velocity.

The `vehicle speed adjustment` factory is shown in Figure 13. It merges a group of surrounding objects with the current steering angle using a fusion operator. It then derives obstacles in front of the ego vehicle using the processing component labeled as "`Select front obstacles`." The factory goes on merging the front obstacles with the current vehicle speed. Finally, it generates the target speed and the target acceleration.

B. Timing Constraints Annotation

We annotate three types of timing constraints in the ACC program. First, we set freshness constraints to the same value of 150ms for the four source components since freshness constraints must consider the maximum vehicle speed. Second, we set a rate constraint to 10Hz for the stream output port of the `ACC` factory. In Splash, freshness and rate constraints are specified as meta-data for related language constructs, as depicted in Figure 11. Finally, we annotate correlation constraints with the fusion operators as specified in Figure 12 and Figure 13. The correlation constraints of two fusion operators in the `3D object detection` factory are set to 20ms. Those of the two fusion operators in the `vehicle speed adjustment` factory are set to 40ms.

C. Lessons Learned

We discuss the lessons that we have learned from writing the ACC application with the Splash programming language.

- Among the various language constructs offered by Splash, what we have benefited the most is surely the fusion operator. We were able to write cleaner code with Splash since we could avoid manually handling time synchronization that arises in specifying sensor fusion. Without the fusion operator, a programmer would have to insert temporal correlation code into the logic of the corresponding processing component. This could easily lead to hard-to-understand and hard-to-maintain code.
- From the perspective of language semantics, we took the greatest advantage of its timing semantics that provides a global time base, the birthmark of a live data item and end-to-end timing constraints. We could validate, both statically and dynamically, the code produced by the Splash code generator, with respect to end-to-end timing constraints. This is because all timing constraints were made explicit in our program and were monitored for their violation at runtime.
- As of writing this paper, Splash is currently evolving. It surely has room for improvement. Programmers would benefit even more if Splash could handle traffic shaping on the stream output ports of a processing component. Programmers must gracefully manage bursty data traffic caused by the variability of communication delay and execution time inside an autonomous machine. If the Splash code generator can automatically attach a traffic shaper to the stream output port, programmers will be free from uncontrolled jitter and queue overflow.

V. Conclusion

In this paper, we proposed the Splash framework for programming an autonomous machine. We first presented our goals we had in mind during the design of Splash and its underlying timing semantics. We then explained in detail the core language constructs of Splash. We have verified that our programming framework achieves our design goals: (1) it provides an effective programming abstraction that supports the stream processing of an autonomous machine; (2) it enables programmers to specify genuine, end-to-end timing constraints and monitor the violation of such constraints; (3) it supports exception handling, mode change and sensor fusion; and (4) it supports performance optimization and tuning during system implementation. To better illustrate the utility of Splash, we developed an adaptive cruise control (ACC) application using Splash.

There are several future research directions along which our programming framework can be extended. First, we are planning to include traffic shaping mechanisms on Splash to better control jitter and bound the size of FIFO queues. Second, we plan on adding triggering rules that can specify various triggering conditions for processing components having multiple stream input ports. Finally, we will attempt to evaluate the performance and run-time overhead of realistic Splash programs with extensive experiments. The result looks promising.

References

[1] W. Shi, M. B. Alawieha, X. Li and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," Integration, the VLSI Journal, no. 59, pp. 148-156, 2017.

[2] NVIDIA, "Jetson AGX Xavier Developer Kit," [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit. [Accessed 29 1 2019].

[3] L. Reger, "The EE architecture for autonomous driving: a domain-based approach," ATZelektronik worldwide, 2017.

[4] "Simulink," [Online]. Available: https://www.mathworks.com/help/simulink/index.html.

[5] N. d. Lac, C. Delaunay and G. Michel, "RTMaps: real time, multisensor, advanced prototyping software," in First National Workshop on Control Architectures of Robots, 2008.

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," Proceedings of the IEEE, vol. 91, no. 1, pp. 127 - 144, 2003.

[7] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zhao and J. Zou, "PTIDES: a programming model for distributed real-time embedded systems," Technical Report No. UCB/EECS-2008-72, 2008.

[8] H. Kopetz and G. Grunsteidl, "TTP - A time-triggered protocol for fault-tolerant real-time systems," in The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993.

[9] 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.

[10] C. Breshears, The art of concurrency, O'Reilly, 2009.

[11] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii and T. Azumi, "Autoware on board: enabling autonomous vehicles with embedded systems," in 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), 2018.