Programming Language Support for Multisensor Data Fusion: The Splash Approach*

Soonhyun Noh, Cheonghwa Lee, Myungsun Kim and Seongsoo Hong

***Abstract*— We present the Splash programming framework to support the effective implementation of multisensor data fusion. Multisensor data fusion has been widely exploited in autonomous machines since it outperforms algorithms using only a single sensor, in terms of accuracy, reliability and robustness. Knowing that developers have long been lacking programming language support for multisensor data fusion, we offer a dedicated Splash language construct along with formal semantics for multisensor data fusion. Specifically, we analyze the structural characteristics of multisensor data fusion algorithms and derive technical issues that the language construct must tackle. We then give a detailed account of the language construct along with its formal semantics. Finally, we validate its utility and effectiveness via its application to a lane keeping assist system.**

I. Introduction

Multisensor data fusion, or sensor fusion, is a technique that estimates information about the nearby situation using data from multiple sensors [1], [2]. Sensor fusion-based algorithms are widely used in autonomous machines since they are more accurate, reliable and robust than algorithms using only a single sensor. For example, an autonomous vehicle fuses various sensor inputs such as cameras, LiDARs and radars for localization, object detection and scene segmentation [3], [4], [5].

As the application area of sensor fusion is rapidly expanding, a wide variety of sensor fusion algorithms have been actively designed and implemented into real systems such as robots, drones and self-driving cars. Despite such popularity, developers still face many technical difficulties in realizing sensor fusion algorithms. Once programmers finish designing a sensor fusion algorithm, they must translate the algorithm into executable code, simultaneously considering various implementation issues such as synchronization among multiple sensor inputs, enforcing arbitrarily-formed, user-defined triggering conditions, estimating missing sensor inputs and handling timeout exceptions, to name a few.

Consequently, developers are forced to rely on ad hoc practices in implementing sensor fusion algorithms. They try to avoid the abovementioned issues by adopting unrealistic assumptions. For example, they often assume that all sensor inputs are perfectly synchronized. In reality, such a naive assumption requires a non-trivial amount of implementation efforts later on, due to the unsynchronized behavior of sensors in the final target system.

Such an ad hoc practice of programming sensor fusion often demands laborious and error-prone coding and tuning, and thus makes an underlying sensor fusion algorithm intertwined with the code that deals with the abovementioned issues. This will seriously jeopardize the correctness and functional safety of autonomous machines. In turn, this calls for a generalized programming framework with a well-defined programming abstraction that can aid programmers in implementing sensor fusion algorithms.

To address such an urgent demand, we present the Splash approach to implementing sensor fusion. Splash is a graphical programming framework we have developed to support programmers in developing diverse applications for autonomous machines [6]. The design goal behind Splash consists of four components. First, Splash must provide an effective programming abstraction that supports the real-time stream processing and sensor fusion of an autonomous machine. Second, it must enable programmers to specify genuine, end-to-end timing constraints and monitor the violation of such constraints at runtime. Third, it must provide basic and yet crucial utilities which are exception handling and mode change. Finally, during system implementation, it must aid programmers with performance optimization and tuning.

Splash primarily relies on three key language semantics: timing semantics, in-order delivery semantics and rate-controlled data-driven stream processing semantics. These semantics serve as a high-level programming abstraction that can hide low-level details from programmers. With these semantics, Splash can automatically generate code and set up interfaces between system components, exempting users from manually linking each part of the application. Splash also monitors the generated interfaces at runtime, allowing users to conveniently handle synchronization and exceptions.

More importantly, Splash provides a specific programming abstraction that lays a foundation for supporting multisensor data fusion in autonomous machines. It consists of a *fusion operator* and *sensor fusion semantics*. We argue that this abstraction can effectively simplify and thus expedite the

* This research was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean 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 Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea (e-mail: shnoh@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).

Myungsun Kim is with the IT Convergence Engineering Department, Hansung University, Seoul, Korea (e-mail: kmsjames@hansung.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).

software development of an autonomous machine by significantly reducing the implementation complexity

There are many graphical programming frameworks intended for real-time stream processing in both industry and academia. Representative examples include RTMaps [7], Simulink [8] and Ptolemy II [9]. Like Splash, these frameworks are based on a data-flow process network model and have extensions that satisfy engineering needs that arise during production-quality system development. Unfortunately, neither Simulink nor Ptolemy II provides support for multisensor data fusion. Only RTMaps allows programmers to specify the triggering condition of a sensor fusion algorithm in a limited manner such that the algorithm is always invoked whenever input is received from a designated sensor. To the best of our knowledge, Splash is the first and the only programming language that explicitly and extensively supports multisensor data fusion.

We have implemented a whole development toolset and runtime system to realize the Splash programming framework. We have also written a lane keeping assist system (LKAS) application using Splash to illustrate how the toolset and runtime work. We have conducted experiments with the LKAS application and measured several performance metrics to validate the utility and effectiveness of the Splash programming framework.

The remainder of this paper is organized as follows. Section II gives s brief description of the Splash programming language to help the readers understand the fusion operator. Section III analyzes the existing sensor fusion algorithms and motivates our approach. Section IV presents the fusion operator by way of its formal semantics and runtime mechanism. Sections V and VI show the Splash toolset and experimental results. Finally, a conclusion follows in Section VII.

II. The Splash Programming Language

To help readers in understanding our programming abstraction, we give a brief overview of the Splash programming language, starting with its underlying timing semantics. We refer the interested readers to [6] for more details of the Splash language constructs.

A. *Timing Semantics and End-to-End Timing Constraints*

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 [10].

(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.

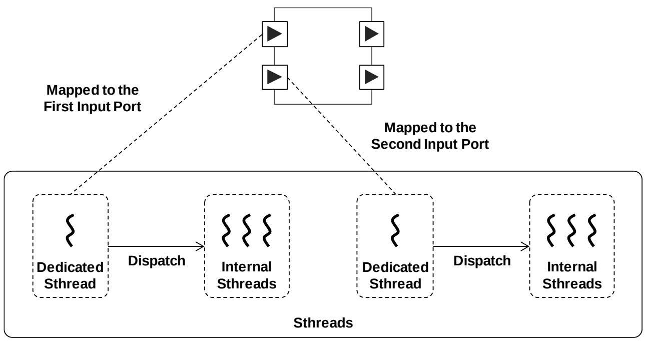

Figure 1. Splash's processsing component and associated sthreads.

(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 these three types of 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. *Key Language Constructs of Splash*

A Splash program is in essence a directed graph that 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.

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 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 for a processing component. In this model, a processing component consists of a group of Splash threads we call *sthreads*. A sthread is an independent logical entity that executes inside a processing

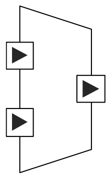

Figure 2. Graphical representation of a fusion operator.

component. Figure 1 shows a processing component that resembles a concurrent server design pattern [11]. It has a dedicated sthread for each port and several other internal sthreads that serve as the worker threads.

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. 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.

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 guarantees 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, a FIFO queue is used. In Splash, a 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 and enqueued into each of the FIFO queues at the input ports of downstream components.

III. Limitations of Current Practice of Programming Sensor Fusion

In this section, we analyze existing sensor fusion algorithms and derive issues that developers must tackle to implement a sensor fusion algorithm. We then motivate the Splash approach to sensor fusion.

The research into sensor fusion can be classified into two categories: measurement fusion and situation fusion. Measurement fusion takes raw measurement data directly from multiple sensors and uses them together to make an estimation [3], [12], [13], [14]. In contrast, situation fusion independently estimates each portion of the nearby situation using a corresponding sensor and then integrates such portions to get the final estimated situation [4], [5], [15].

For the lack of programming language support for sensor fusion, the developers of the existing approaches had to deal with the following critical issues on their own inside their algorithms and even manually embed their solutions into their sensor fusion code.

(1) A *triggering condition* that specifies which subset of sensor inputs must be present to form a legitimate combination of sensor values for fusion.

(2) A *correlation constraint* on multiple sensor values as defined in Section 2.

(3) A *triggering mechanism* that dictates how to produces a sensor value combination that satisfies the imposed correlation constraint as well as the triggering condition.

(4) A *data selection mechanism* that specifies which one should be selected among multiple legitimate combinations of sensor values.

(5) A *data prediction scheme* that computes an estimation for one or more missing sensor inputs.

IV. Splash's Fusion Operator

In this section, we address the abovementioned five issues with a language construct and associated formal semantics. We name the language construct a *fusion operator*.

A fusion operator is an atomic Splash component that merges multiple input data streams into a single output data stream. Its most peculiar benefit is that programmers can graphically specify a triggering condition, a correlation constraint and a data prediction scheme via a well-defined graphical and textual interface while the triggering mechanism and data selection mechanism are automatically synthesized by the Splash toolset, transparently from programmers.

Figure 2 shows the graphical representation of a fusion operator. It has two stream input ports and one stream output port along with a pop-up data template for specifying a *fusion rule*, which consists of a triggering condition and a correlation constraint. The fusion rule can significantly improve the correctness of a sensor fusion algorithm since it forces programmers to clearly enumerate all the elements of the triggering condition.

Once a programmer specifies a fusion rule for a fusion operator, the Splash runtime can automatically check if the fusion rule is satisfied, generates a sensor input combination that satisfies the fusion rule and finally outputs it. This way, Splash can hide low-level details that arise during the system-wide implementation of sensor fusion algorithms.

In this section, we formally present the semantics of the fusion operator and explain its runtime mechanism that the Splash code generator relies on.

A. The Semantics

A fusion operator with a set of m stream input ports $P = \{p_1, p_2, \dots, p_m\}$ has a fusion rule R. A fusion rule R is defined as follows.

Definition 1. A fusion rule $R = (M, O, \theta, c)$ is a tuple where $M \subseteq P$ is a set of mandatory ports and $O \subseteq P$ is a set of

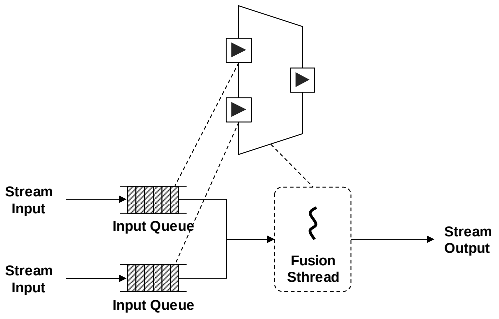

Figure 3. Runtime mechanism and entities for a fusion operator.

optional ports where $M \cap O = \emptyset$ and $M \cup O = P$. Also, θ is a threshold on the number of optional ports and c is a correlation constraint. Let $(d_1, d_2, \ldots, d_m)$ be an input tuple where d_i is one of data items present in the port p_i's input queue. If p_i is empty, d_i has an empty data item, which is denoted by $d_i = \perp$. We define $n_O(d_1, d_2, \ldots, d_m)$ and $n_M(d_1, d_2, \ldots, d_m)$ as the numbers of non-empty optional ports and non-empty mandatory ports, respectively. For R to be satisfied, there must exist at least one tuple $(d_1, d_2, \ldots, d_m)$ that meets the following three conditions.

(C1) For all ports $p_i \in M$, d_i is not $\perp$.

(C2) $n_O(d_1, d_2, \ldots, d_m) \geq \theta$.

(C3) For any two non-empty data items d_i and d_j, $|b(d_i) - b(d_j)| \leq c_i$ where $b(d)$ is the birthmark of a data item d.

We refer to a tuple $(d_1, d_2, \ldots, d_m)$ that satisfies all of the above conditions as a *firing tuple*.

A fusion operator generates a firing tuple whenever the fusion rule is satisfied. If multiple tuples are eligible, it chooses one based on the following rules.

(R1) A fusion operator prefers a firing tuple that has the smallest birthmark. By processing older data items first, the Splash runtime can reduce freshness constraint violations.

(R2) A fusion operator tries to select the firing tuple with the largest $n_O(d_1, d_2, \ldots, d_m)$. This is to provide the fusion algorithm with information from as many sensors as possible.

The fusion operator prioritizes (R1) over (R2).

The fusion operator additionally provides a timeout mechanism for extrapolating missing sensor inputs. A fusion operator gets timed out when no fusion rule has been satisfied for the entire timeout interval. At the timeout, a fusion operator performs the following steps with the extrapolation command e.

(1) It generates a *partially* firing tuple $(d_1, d_2, \ldots, d_m)$ that satisfies only condition (C3). If multiple tuples are eligible, it chooses the one with the largest $n_M(d_1, d_2, \ldots, d_m)$; and then ties are broken in favor of larger $n_O(d_1, d_2, \ldots, d_m)$.

(2) For all d_i such that $d_i = \perp$ and $p_i \in M$, it places e for d_i.

```
ALGORITHM    FINDVALIDINPUTTUPLE

Input:  A fusion rules $R = (M, O, \theta, c)$
        A set of lists of data items $S = \{s_1, s_2, \ldots, s_m\}$

FINDVALIDINPUTTUPLE($R$, $S$)
1:  let index[1 ... $m$] be a new array
2:  for $i = 1$ to $m$
3:      if $p_i \in M \cup O$ and $|s_i| > 0$
4:          index[$i$] ← 1
5:      else
6:          index[$i$] ← NIL
7:  while index[$i$] = NIL for $1 \leq i \leq m$
8:      if ISVALIDTUPLE($R$, $S$, index)
9:          return BUILDTUPLE($S$, index)
10:     $k$ ← GETEARLISTINDEX($S$, index)
11:     if index[$k$] < $|s_k|$
12:         index[$k$] ← index[$k$] + 1
13:     else
14:         index[$k$] ← NIL
15: return $(\perp, \perp, \ldots, \perp)$
```

Figure 4. Pseudocode for **FINDVALIDINPUTTUPLE**.

(3) While $n_O(d_1, d_2, \ldots, d_m) < \theta$, for each d_i such that $d_i = \perp$ and $p_i \in O$, it repeatedly places e for d_i.

(4) It finally outputs $(d_1, d_2, \ldots, d_m)$.

It is the programmers' responsibility to handle the extrapolation command. For example, they may choose to either ignore e or arrange an extrapolation handler inside the processing component directly connected to the output of the fusion operator.

B. The Runtime Mechanism

The runtime mechanism of a fusion operator is depicted in Figure 3. Each stream input port of a fusion operator has an input queue that stores data items in ascending order of their birthmarks. A fusion operator is associated with a dedicated sthread named a *fusion sthread*. When a data item is inserted into any of input queues, a fusion sthread is invoked. It checks if the fusion rule is satisfied. If so, it retrieves data items and constructs a firing tuple and outputs it. It repeats until no firing tuple is found. Then it gets blocked until a new data item comes in.

The fusion sthread runs the **FINDVALIDINPUTTUPLE** algorithm whose pseudocode is given in Figure 4. The inputs of the algorithm are a fusion rule $R = (M, O, \theta, c)$ and a set $S = \{s_1, s_2, \ldots, s_m\}$ where s_i is a list of data items stored in p_i's input queue. Each list is sorted in ascending order of its data items' birthmarks. The algorithm returns a firing tuple $(d_1, d_2, \ldots, d_m)$ on success or $(\perp, \perp, \ldots, \perp)$, otherwise.

The **FINDVALIDINPUTTUPLE** algorithm always returns the firing tuple with the smallest birthmark when there are multiple eligible tuples. The runtime complexity of the algorithm is $O(m^2 \cdot l)$ where m be the number of input ports of the fusion operator and l be the maximum input queue size. This is clear because the while loop repeats at most $m \cdot l$ times and it takes $O(m)$ to run lines 7-9 and $O(1)$ to run lines 10 and 11.

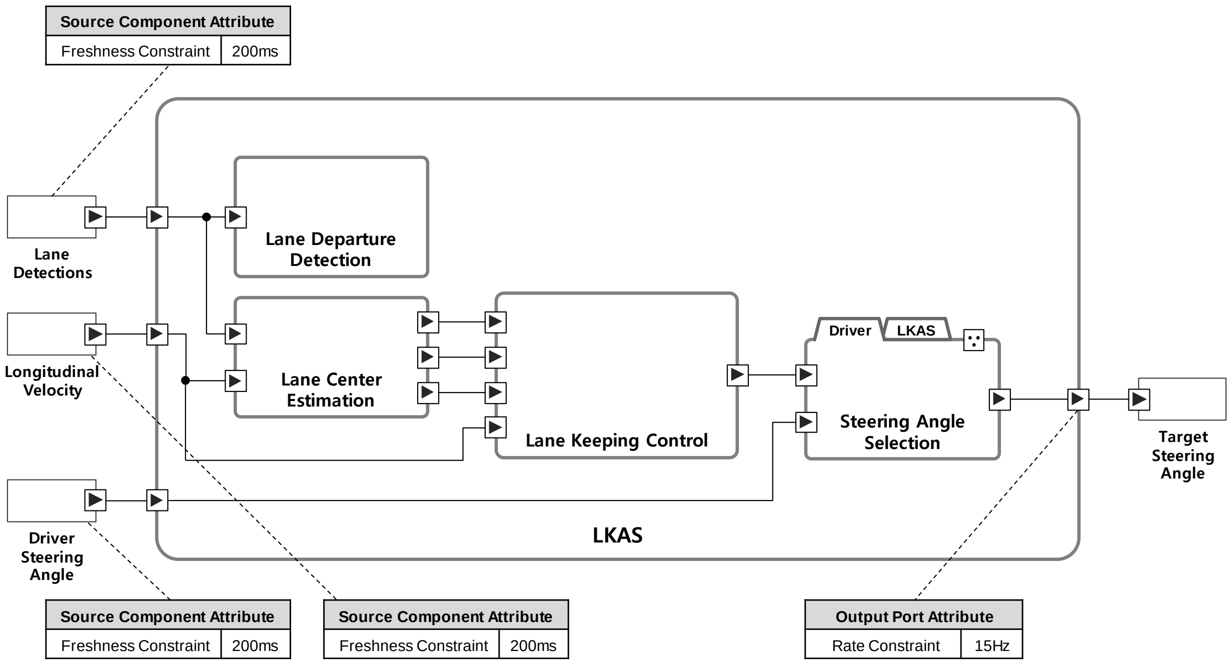

Figure 5. `Lane Keeping Assist System` factory.

V. Splash Toolset and Runtime

In this section, we describe how the Splash toolset and runtime work by using a lane keeping assist system (LKAS) application as a work-through example. We also demonstrate the utility of the fusion operator.

A. *LKAS using Splash*

We have designed LKAS based on an algorithm in [16]. It automatically adjusts the steering angle to keep the ego vehicle inside the detected lane. Figure 5 shows the top-level `LKAS` factory. Its inputs include the detected lane, the longitudinal velocity of the ego vehicle and the driver steering angle. Its output is the target steering angle of the vehicle. The `LKAS` factory consists of four sub-factories: (1) `Lane Departure Detection`, (2) `Lane Center Estimation`, (3) `Lane Keeping Control` and (4) `Steering Angle Selection`.

To illustrate the internals of a factory, we choose the `Lane Center Estimation` factory from Figure 5 and draw it in Figure 6. The factory computes the curvature of the lane, the lateral offset between the ego vehicle and the center of the lane and the heading angle of the ego vehicle. To generate such outputs, it first checks if the left and the right boundary of the detected lane are clear enough to be used for estimation and selects one of the boundaries as a reference. It also estimates the curvature of the forward lane on which the ego vehicle runs over the next three seconds using the curvature computed through the currently detected lane.

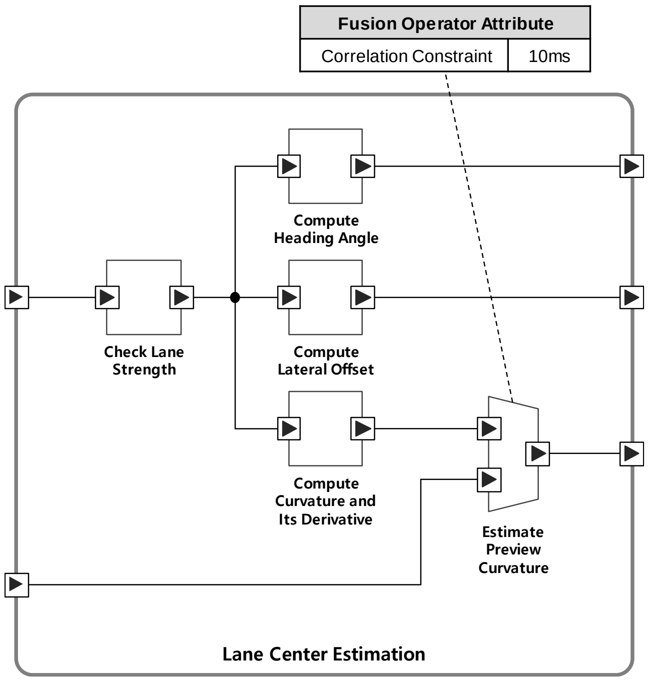

Figure 6. `Lane Center Estimation` factory.

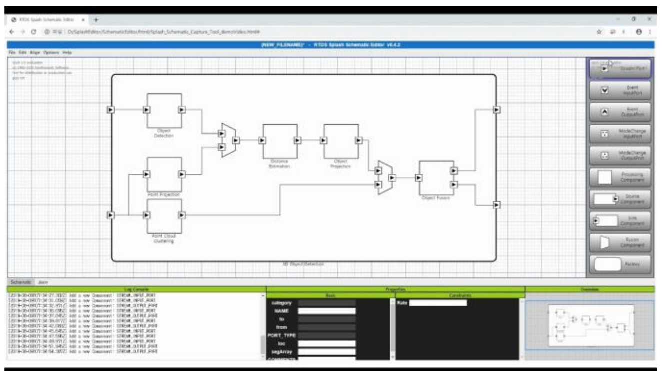

Figure 7. Splash schematic editor.

```
{
   "name": "LaneCenterEstimation",
   "ProcessingComponent": [
      {
         "name": "CheckLaneStrength",
         "stream_input_port": {"source_channel": "//@top_level_factory/@factory.0/@channel.0"},
         "stream_output_port": {"target_channel": //@top_level_factory/@factory.0/@channel.1"}
      },
      …
   ],
   "FusionOperator": {
      …
   },
   "Channel": [
      …
   ],
   …
}
```

Figure 8. JSON output for `Lane Center Estimation` factory generated by the schematic editor.

In Splash, programmers can annotate timing constraints with a processing component. In Figure 5, three source components have 200ms as their freshness constraint. The stream output port has 15HZ as its rate constraint. A fusion operator appears in Figure 6. It has 10ms as a correlation constraint.

B. *Splash Toolset*

We provide Splash programmers with two development tools: a schematic editor and a code generator.

Splash is a coordination language that is used to specify the interactions between components. A host language such as C++ is used to define subprograms inside a component [17]. Programmers use the schematic editor shown in Figure 7 to write a coordination program in Splash. Once a programmer finishes writing a coordination program, the schematic editor can produce a JSON file, which is used by the code generator. The JSON file for the `lane center estimation` factory is shown in Figure 8. It contains information about a factory and internal language constructs such as processing components, fusion operators, channels and stream input/output ports.

The code generator takes a JSON file produced by the schematic editor and outputs two types of files: an IDL (interface definition language) file [18] and template source code files written in C++. An IDL file describes the data type of each port interface. Figure 9 shows the IDL output for the `Lane Center Estimation` factory.

```
module LaneCenterEstimation
{
    struct data0
    {
        double curvature;
        double curvature_derivative;
        // …
    };
    struct data1
    {
        bool detected;
    };
    // …
};
```

Figure 9. IDL output for `Lane Center Estimation` factory.

We explain the template code using the `Check Lane Strength` processing component in the `Lane Center Estimation` factory as an example. As shown in Figure 10, the template code contains two segments: configuration (lines 3-12) and execution (line 13). In the configuration segment, the processing component and its internal stream input/output port objects are declared (lines 3-7) and initialized (lines 8-10). The stream input/output port objects are attached to the processing component object (lines 11-12). In the execution segment, it waits for a data item to come in on the input port (line 13). When a data item arrives, the user function for the `Check Lane Strength` processing component is called (lines 16-22). Programmers should fill in their own logic inside the user function (line 20).

After filling in the template source code, programmers compile source code files and build an executable image using

```
1:  int main(void)
2:  {
3:      ProcessingComponent pc;
4:      StreamInputPort
5:      <LaneCenterEstimation::data0> sin0;
6:      StreamOutputPort
7:      <LaneCenterEstimation n::data1> sout0;
8:      pc.initialize("CheckLaneStrength");
9:      sin0.initialize();
10:     sout0.initialize(30);
11:     sin0.attach(&pc, "topic1");
12:     sout0.attach(&pc, "topic2");
13:     pc.run();
14: }
15:
16: template<typename t> void
17: ProcessingComponent<t>::usr_func(t input)
18: {
19:     LaneCenterEstimation::data1 output
20:     // Write user logic here
21:     this->write(&output_data, "topic1");
22: }
```

Figure 10. Template code for `Check Lane Strength` processing component.

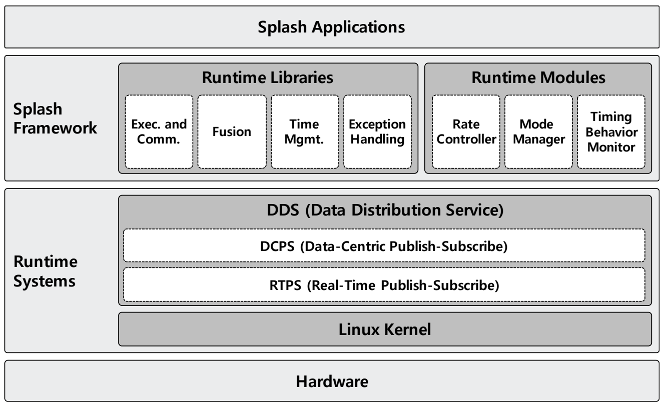

Figure 11. Splash runtime architecture.

the compiler of the host language and the IDL preprocessor [18].

C. *Splash Runtime*

The Splash runtime consists of two layers of software as shown in Figure 11. At the top layer is the Splash framework that consists of runtime libraries and modules written in the host language. The user-augmented template code uses the library provided by the Splash framework as shown in Figure 10. The runtime libraries are divided into four subtypes according to their functions: (1) execution and communication, (2) fusion, (3) time management and (4) exception handling. The Splash framework also comes with three runtime modules: (1) the rate controller, (2) the mode manager and (3) the timing behavior monitor.

At the bottom layer lies a runtime system based on DDS (data distribution services) and Linux. DDS is a well-known specification for real-time publish-subscribe communication [19]. We chose OpenSplice DDS because it is open source and implements the specification efficiently [20].

VI. Validating Fusion Operator

In this section, we validate the effectiveness of the fusion operator by experimentally assessing the performance of the LKAS. We first describe the execution environment of our experiments and performance metrics. We then present the experimental results.

A. *Execution Environment and Metrics for Validation*

We simulated a driving environment with MathWorks' Simulink for our experiments [16]. Specifically, we created a closed-loop simulator on top of two machines: one running the Splash implementation of LKAS and the other executing the driving simulator. The LKAS receives sensor values from the simulator and outputs a target steering wheel angle. The

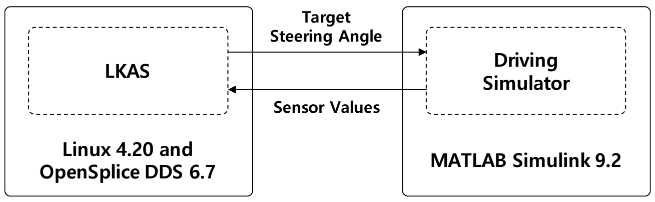

Figure 12. Runtime platform for LKAS and driving simulator.

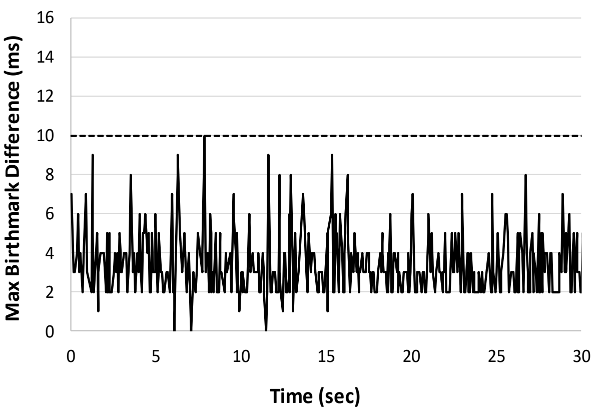

Figure 13. Maximum birthmark differences between data items in firing tuples.

simulator in turn receives the steering wheel angle as its input. The software organization is shown in Figure 12.

We chose two metrics to evaluate the performance of the LKAS: the maximum birthmark difference between data items in each firing tuple and the average runtime overhead of the fusion operator. The **FindValidInputTuple** algorithm is responsible for most of the overhead.

B. *Experimental Results*

In our first experiment, we ran the LKAS program for 30 seconds and measured the maximum birthmark differences between data items in firing tuples generated by the `Estimate Preview Curvature` fusion operator inside the `Lane Center Estimation` factory. Figure 13 shows the result. The maximum birthmark difference between data items in every firing tuple was always less than the fusion operator's correlation constraint, 10 milliseconds. This result confirms that the fusion operator satisfied the annotated correlation constraint.

In our second experiment, we measured the average running time of the **FindValidInputTuple** algorithm. As expected, the runtime overhead was only 7 microseconds on average, which is negligible.

VII. Conclusion

In this paper, we presented the fusion operator of Splash along with its formal semantics. Due to the lack of proper linguistic support, programmers had to desperately resort to laborious and error-prone coding and tuning during the implementation of sensor fusion algorithms. Such an ad hoc practice leads to an undesirable code structure where sensor fusion code is deeply intertwined with managerial code that deals with tricky implementation details. This will seriously deteriorate code quality and jeopardize the correctness and functional safety of autonomous machines

The greatest benefit of the Splash approach is that programmers can graphically specify a triggering condition, a correlation constraint and a data prediction scheme via a well-defined interface while the triggering mechanism and data selection mechanism are automatically synthesized by the Splash code generator, transparently from programmers. As a result, programmers can be safely exempted from manual code development.

We validated the utility and effectiveness of the fusion operator via its application to a lane keeping assist system. The result looks promising.

REFERENCES

[1] B. Khaleghi, A. Khamis, F. O. Karray, and S. N. Razavi, “Multisensor data fusion: A review of the state-of-the-art,” *Inf. Fusion*, vol. 14, no. 1, pp. 28–44, Jan. 2013.

[2] Hugh Durrant-Whyte, “Multi-Sensor Data Fusion,” The University of Sydney, 2001.

[3] H. Cho, Y.-W. Seo, B. V. K. V. Kumar, and R. R. Rajkumar, “A multi-sensor fusion system for moving object detection and tracking in urban driving environments,” in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 1836–1843.

[4] R. Zhang, S. A. Candra, K. Vetter, and A. Zakhor, “Sensor fusion for semantic segmentation of urban scenes,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, vol. 2015-June, no. June, pp. 1850–1857.

[5] R. O. Chavez-Garcia and O. Aycard, “Multiple Sensor Fusion and Classification for Moving Object Detection and Tracking,*” IEEE Trans. Intell. Transp. Syst.*, vol. 17, no. 2, pp. 525–534, Feb. 2016.

[6] S. Noh and S. Hong, “Splash: a graphical programming framework for an autonomous machine,” *International Conference on Ubiquitous Robots (UR),* submitted for publication.

[7] N. d. Lac, C. Delaunay and G. Michel, “RTMaps: real time, multisensor, advanced prototyping software,” *National Workshop on Control Architectures of Robots,* 2008.

[8] “Simulink,” [Online]. Available: https://www.mathworks.com/help/simulink/index.html.

[9] 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,* 2003.

[10] R. Gerber, S. Hong and M. Saksena, “Guaranteeing real-time requirements with resource-based calibration of periodic processes,” *IEEE Transactions on Software Engineering*, 1995.

[11] C. Breshears, “The art of concurrency,” O'Reilly, 2009.

[12] L. Drolet, F. Michaud, and J. Cote, “Adaptable sensor fusion using multiple Kalman filters,” in Proceedings. *2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)*, 2000, vol. 2, pp. 1434–1439.

[13] Feng Liu, J. Sparbert, and C. Stiller, “IMMPDA vehicle tracking system using asynchronous sensor fusion of radar and vision,” in *2008 IEEE Intelligent Vehicles Symposium*, 2008, pp. 168–173.

[14] P. Geneva, K. Eckenhoff, and G. Huang, “Asynchronous Multi-Sensor Fusion for 3D Mapping and Localization,” in *2018 IEEE International Conference on Robotics and Automation (ICRA*), 2018, pp. 1–6.

[15] N. Floudas, A. Polychronopoulos, O. Aycard, J. Burlet, and M. Ahrholdt, “High Level Sensor Data Fusion Approaches For Object Recognition In Road Environment,” in *2007 IEEE Intelligent Vehicles Symposium*, 2007, pp. 136–141.

[16] “Lane keeping assist with lane detection,” [Online]. Available: https://www.mathworks.com/help/mpc/ug/lane-keeping-assist-with-lane-detection.html

[17] E. A. Lee and T. M. Parks, “Dataflow process networks,” *Proceedings of the IEEE,* 1995.

[18] Adlink, “Vortex OpenSplice IDL preprocessor guide,” 2018.

[19] OMG, “Data distribution service (DDS) version 1.4,” 2015.

[20] “Vortex OpenSplice,” [Online]. Available: https://github.com/ADLINK-IST/opensplice.