



## 저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Ph.D. Dissertation

Design of the Splash  
Programming Language to  
Support Real-Time Stream  
Processing and Sensor Fusion for  
an Autonomous Machine

Autonomous Machine을 위한  
실시간 스트림 처리와 센서 퓨전을 지원하는  
Splash 프로그래밍 언어의 설계

February 2020

Department of Electrical Engineering and  
Computer Science  
College of Engineering  
Seoul National University

Soonhyun Noh

Ph.D. Dissertation

Design of the Splash  
Programming Language to  
Support Real-Time Stream  
Processing and Sensor Fusion for  
an Autonomous Machine

Autonomous Machine을 위한  
실시간 스트림 처리와 센서 퓨전을 지원하는  
Splash 프로그래밍 언어의 설계

February 2020

Department of Electrical Engineering and  
Computer Science  
College of Engineering  
Seoul National University

Soonhyun Noh

# 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, the complexity of the underlying software platform is increasing at a rapid pace, overwhelming the developers with implementation details. This leads to a demand for a new programming framework that has an easy-to-use programming abstraction.

In this thesis, 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 the challenges. First, Splash should provide an easy-to-use, effective programming abstraction. Second, it must support real-time stream processing for deep-learning based machine learning intelligence. Third, it must provide programming support for real-time control system of autonomous machines such as sensor fusion and mode change. Finally, it should support performance optimization of software system running on a heterogeneous multicore distributed computing platform.

Splash allows programmers to specify genuine, end-to-end timing constraints. Also, it provides a best-effort runtime system that tries to meet the annotated timing constraints and exception handling mechanisms to monitor the violation of such constraints. To implement these runtime mechanisms, Splash provides underlying timing semantics: (1) it provides an abstract global clock that is shared by machines in the distributed system and (2) it supports

programmers to write birthmark on every stream data item.

Splash offers a multithreaded process model to support concurrent programming. In the multithreaded process model, a programmer can write a multithreaded program using Splash threads we call sthreads. An sthread is a logical entity of independent execution. In addition, Splash provides a language construct named build unit that allows programmers to allocate sthreads to processes and threads of an underlying operating system.

Splash provides three additional language semantics to support real-time stream processing and real-time control systems. First, it provides rate control semantics to solve uncontrolled jitter and an unbounded FIFO queue problem due to the variability in communication delay and execution time. Second, it supports fusion semantics to handle timing issues caused by asynchronous sensors in the system. Finally, it provides mode change semantics to meet varying requirements in the real-time control systems. In this paper, we describe each language semantics and runtime mechanism that realizes such semantics in detail.

To show the utility of our framework, we have written a lane keeping assist system (LKAS) in Splash as an example. We evaluated rate control, sensor fusion, mode change and build unit-based allocation. First, using rate controller, the jitter was reduced from 30.61 milliseconds to 1.66 milliseconds. Also, average lateral deviation and heading angle is reduced from 0.180 meters to 0.016 meters and 0.043 rad to 0.008 rad, respectively. Second, we showed that the fusion operator works normally as intended, with a run-time overhead of only 7 microseconds on average. Third, the mode change mechanism operated correctly and incurred a run-time overhead of only 0.53 milliseconds. Finally, as we increased the number of build

units from 1 to 8, the average end-to-end latency was increased from 75.79 microseconds to 2022.96 microseconds. These results show that the language semantics and runtime mechanisms proposed in this thesis are designed and implemented correctly, and Splash can be used to effectively develop applications for an autonomous machine.

**Keywords :** Autonomous Machine, Real-time Stream Processing, Rate Control, Sensor Fusion, Mode Change

**Student Number :** 2013-20785

# Table of Contents

Abstract .....	i
Table of Contents .....	iv
List of Tables.....	vii
List of Figures .....	viii
<b>Chapter 1 Introduction.....</b>	<b>1</b>
1.1 Motivation .....	2
1.2 Splash Overview.....	5
1.3 Organization of This Dissertation.....	9
<b>Chapter 2 Related Work .....</b>	<b>10</b>
2.1 Kahn Process Network .....	10
2.2 Firing Rule Applied to a Process.....	13
2.3 Programming Framework for an Autonomous Machine..	14
2.4 Runtime Software for an Autonomous Machine .....	16
2.5 Rate Control.....	18
2.5.1 Traffic Shaping.....	20
2.5.2 Traffic Policing.....	22
2.6 Sensor Fusion.....	23
2.6.1 Measurement Fusion.....	24
2.6.2 Situation Fusion.....	27
2.7 Mode Change .....	30
2.7.1 Synchronous Mode Change.....	32
2.7.2 Asynchronous Mode Change .....	32

<b>Chapter 3</b>	<b>Motivation and Contributions .....</b>	<b>34</b>
3.1	Problem Description.....	34
3.2	Limitations of Kahn Process Network .....	36
3.3	Contributions of this Dissertation .....	38
<b>Chapter 4</b>	<b>Underlying Timing Semantics of Splash.....</b>	<b>41</b>
4.1	End-to-End Timing Constraints .....	41
4.2	Global Time Base and In-order Delivery .....	42
4.3	Integrating Three Distinct Computing Models.....	43
<b>Chapter 5</b>	<b>Splash Language Constructs.....</b>	<b>45</b>
5.1	Processing Component.....	46
5.2	Port .....	49
5.3	Channel and Clink.....	52
5.4	Fusion Operator.....	54
5.5	Factory and Mode Change .....	60
5.6	Build Unit .....	65
5.7	Exception Handling .....	67
<b>Chapter 6</b>	<b>Splash Runtime Mechanisms.....</b>	<b>69</b>
6.1	Rate Control Mechanism .....	69
6.2	Sensor Fusion Mechanism .....	70
6.3	Mode Change Mechanism .....	77
<b>Chapter 7</b>	<b>Code Generation and Runtime System .....</b>	<b>80</b>
7.1	Build Unit-based Allocation.....	80
7.2	Code Generation Template .....	82
7.3	Splash Runtime System .....	84



<b>Chapter 8</b>	<b>Experimental Evaluation.....</b>	<b>86</b>
8.1	LKAS Program .....	86
8.2	Experimental Environment .....	91
8.3	Evaluating Rate Control .....	92
8.4	Evaluating Sensor Fusion .....	96
8.5	Evaluating Mode Change.....	97
8.6	Evaluating Build Unit-based Allocation.....	99
<b>Chapter 9</b>	<b>Conclusion.....</b>	<b>102</b>
<b>Bibliography</b> .....		<b>104</b>
<b>Abstract in Korean</b> .....		<b>113</b>

# List of Tables

Table 1	Graphical symbols for ports.....	49
Table 2	Example of mode change table.....	63
Table 3	Experimental environment .....	91

# List of Figures

Figure 1	Example of Kahn Process Network graph.....	11
Figure 2	Traffic shaping.....	19
Figure 3	Traffic policing.....	20
Figure 4	Measurement fusion.....	25
Figure 5	Situation fusion .....	28
Figure 6	Hierarchy of Splash components .....	45
Figure 7	Splash program example: 2D object detection .....	46
Figure 8	Processing component .....	46
Figure 9	Processing component and its sthreads.....	47
Figure 10	Source component .....	48
Figure 11	Sink component.....	48
Figure 12	Input and output ports as subtype of port .....	50
Figure 13	Hierarchy of port interfaces .....	50
Figure 14	Rate-controlled stream output port .....	51
Figure 15	Behavior of a rate controller.....	52
Figure 16	Channel .....	53
Figure 17	Channel with three fan-outs .....	53
Figure 18	Clinks .....	54
Figure 19	Fusion operator.....	55
Figure 20	A multimode factory with two modes .....	61
Figure 21	Example of mode factory.....	62

Figure 22	Internal data items of a multimode factory .....	64
Figure 23	Example of component–build unit mapping .....	67
Figure 24	Hierarchy of exception class .....	68
Figure 25	Runtime mechanism of a rate controller .....	69
Figure 26	Runtime mechanism of a fusion operator.....	71
Figure 27	Pseudocode of <b>FINDVALIDINPUTTUPLE</b> algorithm	72
Figure 28	Example of $I_{\text{older}}$ , $I_{\text{answer}}$ , $I_{\text{newer}}$ .....	74
Figure 29	The relationship between $d'_k$ , $d''_k$ , $d'_j$ and $d''_j$ .....	76
Figure 30	Runtime mechanism of mode change .....	78
Figure 31	Pseudocode of <b>CHANGEMODE</b> algorithm.....	79
Figure 32	Example of template source code.....	83
Figure 33	Splash runtime architecture.....	85
Figure 34	LKAS factory .....	86
Figure 35	Lane departure detection factory .....	87
Figure 36	Lane center estimation factory .....	88
Figure 37	Lane keeping control factory .....	89
Figure 38	Steering angle selection factory .....	89
Figure 39	Timing constraints of LKAS factory.....	90
Figure 40	Software components of the platform.....	91
Figure 41	Comparison of the number of the accumulated output items.....	93
Figure 42	Comparison of the end–to–end latency .....	94
Figure 43	The number of data items in the output queue	

.....	95
Figure 44 Comparison of the lateral deviation of the ego vehicle .....	95
Figure 45 Comparison of the heading angle of the ego vehicle .....	96
Figure 46 Maximum birthmark difference of input tuples chosen by the fusion operator.....	97
Figure 47 The steering angle selected using mode change .....	98
Figure 48 A Splash program and its build unit configurations .....	100
Figure 49 End-to-end latencies of the distinct build unit configurations.....	101

# Chapter 1. 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 [1–3]. Often times, they are equipped with diverse sensors for perception, localization and positioning [4,5]. They also include high performance multicore processors for intelligence and microcontrollers for real-time control [6,7].

These hardware components are interconnected via onboard networks inside autonomous machines [8–10]. 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.

The programming framework for autonomous machine should achieve four key design goals. First, it should provide an easy-to-use, effective programming abstraction that can hide implementation details and supports a model-based code generation capability. Second, it must support real-time stream processing for deep-learning based machine learning intelligence. Third, it must provide

programming support for real-time control system of autonomous machine such as sensor fusion and mode change. Finally, it needs to support performance optimization of software system running on a heterogeneous multicore distributed computing platform.

In this thesis, we present a graphical programming framework named Splash that achieves four design goals. We present the syntax and semantics of the key language constructs of Splash and show how we achieve our design goals. Furthermore, we present the internal workings of the proposed programming framework and validate its effectiveness via a lane keeping assist system (LKAS). Section 1.1 describes the motivation. Then, Section 1.2 gives overview of our work. Finally, Section 1.3 explains how this dissertation is organized.

## **1.1 Motivation**

Quite a few graphical programming frameworks have been widely used for developing autonomous machines, particularly for automatic control and signal processing domains. Such frameworks include Simulink and RTMaps [11,12]. Also, several academic programming frameworks such as Ptolemy II exist for research purposes [13]. 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. It also offers a wide range of plug-ins such as Stateflow, SimEvents and Deep Learning Toolbox to support programmers to develop embedded applications [14]. 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 must 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 control systems such as mode change and exception handling.



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 programming such as mode change and exception handling. However, Ptolemy II lacks support for real-time stream processing. 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.

Due to the limitations of these programming frameworks, many programmers choose to develop autonomous machines without using these programming frameworks. ROS is a representative open-source runtime software system that is commonly used to develop an autonomous machine [15]. However, existing runtime software systems including ROS does not support any method for specifying and handling end-to-end timing constraints for real-time stream processing. In order to overcome the limitation of the ROS, ROS 2 is currently under development based on data distribution service (DDS), a communication standard that supports real-time publish-subscribe communication. However, each function in ROS 2 is not yet fully implemented and verified since it is still in the early stage of development [15,16].

Therefore, we need a new programming framework that

overcomes the limitations of existing programming frameworks and runtime software. Unlike existing approaches, the proposed programming framework must be able to achieve all four design goals.

## 1.2 Splash Overview

In this thesis, we present a graphical programming framework named Splash that achieves all design goals. The name Splash is named after the first letter of the first three words in the *stream processing language for an autonomous machine*.

Splash is designed based on the Kahn process network (KPN), which offers a programming model in such a way that developers can write an application in a parallel way such that constituent processes are independently written [17,18]. KPN provides graphical programming abstraction to programmers and helps them avoid error-prone issues such as data races and non-determinism. However, KPN cannot be directly used to develop an autonomous machine since it fails to achieve our four design goals. To overcome this limitation, Splash offers six additional language semantics: (1) timing semantics, (2) exception handling semantics, (3) multi-threaded processing model and build unit-based allocation, (4) rate control semantics, (5) sensor fusion semantics and (6) mode change semantics.

One of the most important design goal of Splash is to support real-time processing. To achieve this goal, Splash allows

programmers to specify three essential end-to-end timing constraints: freshness constraint, correlation constraint and rate constraint. It provides a best-effort runtime system to satisfy the timing constraints annotated in the program. Splash also provides exception handling mechanism to monitor and to handle violations of such constraints at runtime.

Splash provides timing semantics which is the basis for all other language semantics of Splash. Splash supports an abstract global clock that is possibly implemented via distributed local clock synchronization. It also enables programmers to write birthmark on every stream data item, and guarantees that data items always go through a communication channel in the order of their birthmarks. This is called in-order delivery semantics.

Splash offers a multithreaded process model to exploit parallelism explicitly from the underlying operating system and computing platform. In the multithreaded process model, a programmer can write a multithreaded program using sthread that is a logical entity of execution. Splash also supports a language construct named build unit to allocate sthreads to processes and threads on the underlying operating system.

Splash takes the data-driven processing as the default style, unless specified otherwise in a program. However, data-driven triggering is not the most suitable programming abstraction for an autonomous machine since it may have serious side effects such as uncontrolled jitter and an unbounded queue. Variability in

communication delay and execution time in a physical system can easily cause bursty data traffic on communication channels and eventually deteriorate the resultant control quality to a significant degree. To solve these problems, Splash provides rate control semantics.

Splash also offers a language construct named fusion operator that handles complex implementation issues caused by asynchronous sensor inputs during the development of sensor fusion algorithms [19,20]. Using the fusion operator, a programmer can clearly specify temporal requirements of a fusion algorithm. Then, Splash provides a runtime system that handles these issues automatically.

Finally, Splash provides mode change semantics that is often used in real-time control systems. The Splash provide a language construct named multimode factory to support multiple modes of operations. A programmer can describe the behavior of each mode and the specification of mode change. The Splash runtime system then performs mode changes according to the programmer's specification. During the mode change, the consistency of data used by sthreads is preserved.

The proposed language semantics of Splash achieves our key design goals as follow:

- (1) The Splash's language semantics is designed to provide programmers with an easy-to-use programming abstraction. It allows programmers to focus on developing their business

logic without worrying about the specific implementation issues that arise during the development of an autonomous machine.

- (2) The Splash enables programmers to specify three essential end-to-end timing constraints and provides timing semantics, rate control semantics and sensor fusion semantics to satisfy such constraints. Also, it supports exception handling semantics to monitor and to handle the timing constraint violation.
- (3) In order to provide programming supports for the development of real-time control systems, Splash provides fusion semantics, mode change semantics and exception handling semantics.
- (4) To support development and performance optimization in the multicore distributed computing platform, The Splash provides multithread process model based on sthread and build unit-based allocation

To show the effectiveness of our framework, we wrote a lane keeping assist system (LKAS) as a Splash program example. We then evaluated rate control mechanism, sensor fusion mechanism, mode change mechanism and build unit-based allocation. First, the jitter was reduced from 30.61 milliseconds to 1.66 milliseconds when using a rate controller. As a result, average lateral deviation and heading angle is also reduced from 0.180 meters to 0.016 meters

and 0.043 rad to 0.008 rad, respectively. Second, we showed that the fusion operator successfully satisfies temporal requirements that is annotated in the program with 7 microseconds of run-time overhead. Third, we showed that the mode change mechanism works as intended, with a run-time overhead of only 0.53 milliseconds. Finally, as we increased the number of build units from 1 to 8, the average end-to-end latency was increased from 75.79 microseconds to 2022.96 microseconds. These results show that the language semantics and runtime mechanisms proposed in this paper are designed and implemented correctly.

### **1.3 Organization of This Dissertation**

This dissertation is organized as follows. Chapter 2 explains the background and related work of our work. Chapter 3 describes the motivation and contributions of this dissertation. Chapter 4 explains the underlying timing semantics of Splash. Chapter 5 then presents language constructs of Splash. Chapter 6 explains runtime mechanisms of Splash. Chapter 7 presents code generation and Splash runtime system. Chapter 8 reports on the experimental evaluation. Finally, Chapter 8 concludes the dissertation.

## Chapter 2. Related Work

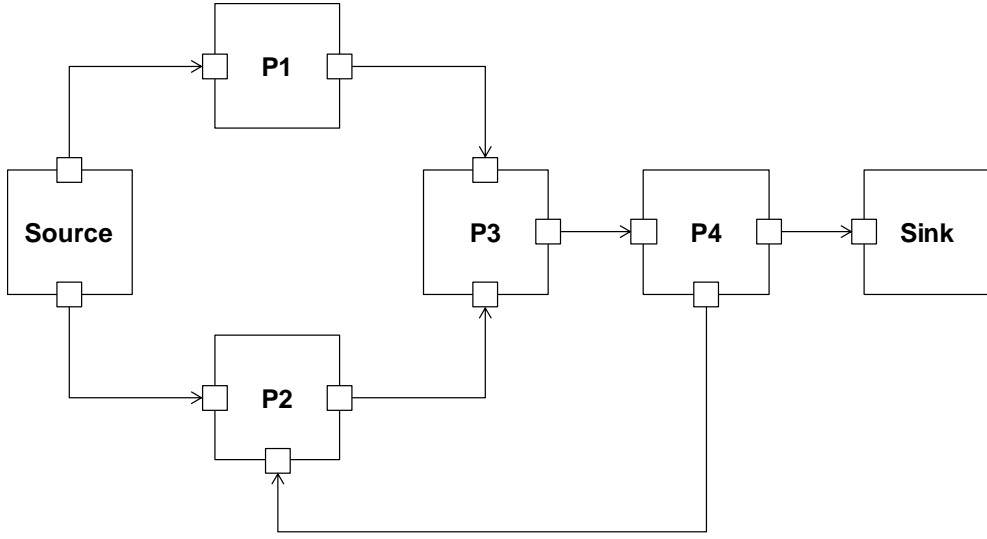
This chapter presents background underlying Splash and related work on (1) programming frameworks for an autonomous machine, (2) runtime systems for an autonomous machine, (3) rate control, (4) sensor fusion and (5) mode change.

### 2.1 Kahn Process Network

Kahn process network is a fundamental process network model underlying the Splash programming language [17,18]. A KPN offers a programming model in that developers are allowed to write an application in a parallel way such that constituent processes are independently written, allocated and executed on a multiprocessor system such as a modern multicore system. Unlike thread programming on a shared-memory machine, the KPN model helps developers avoid error-prone issues such as data races and non-determinism.

A KPN is rendered in a directed graph format, where for a given graph  $G(V, E)$ :

- $V$  is a set of processes. A process  $v_i \in V$  computes on data items coming from its incoming edges to produce data items on its outgoing edges.
- $E \subseteq (V \times V)$  is a set of unbounded unidirectional edges. An



**Figure 1. Example of Kahn Process Network graph.**

edge  $(v_i, v_j) \in E$  denotes a FIFO channel that is able to carry data items of a given data type from the process  $v_i$  to the process  $v_j$ . There is no restriction on the number of incoming edges or the number of outgoing edges for a process in a KPN. A KPN may be either cyclic or acyclic.

For syntactic clarity, a process in a KPN has an input port connected to an incoming edge and an output port connected to an outgoing edge as described in [18]. Figure 1 shows an example KPN graph that has one source, one sink and a cycle.

The KPN model provides two predefined functions for processes: `get()` and `put()`. A process uses `get()` to obtain a data item from a channel connected to an input port. If the channel is empty, the process stays blocked on `get()` until some data item is being sent into the input port. A process calls `put()` to send a data item along a



channel connected to an output port. In contrast to `get()`, nothing can block a process from performing `put()`.

The behavior of a process is specified with a sequential program written in an imperative programming language. Typically, a process sequentially reads in data items from inputs ports, computes on the data items and writes generated data items into output ports. A process may skip reading data items from certain input ports, but this leads to a problem of infinitely stored data on a channel. A KPN is referred to as an effective process network if it is free of such problem [21].

Each process in a KPN is viewed as a function that maps the complete history of data items received on its input channels to the complete history of data items emitted on its output channels. The most intriguing property of a KPN is that the network is determinate. A process network is determinate if and only if it defines a unique history of data items on each channel between processes. Whereas a process may have a private state, no shared state between any two processes is allowed. This is required for maintaining the determinacy of a KPN.

It is also shown that any two fair and maximal executions of a KPN produce the same history of data items on each channel. A fair execution is one that ensures that if any process is able to produce an output data item or read an input data item, then it will eventually be allowed to do so. A maximal execution is an execution that either does not halt, or if it halts, has produced exactly every sequence

defined by the network. This property is known as the Kahn principle [22,23].

## 2.2 Firing Rule Applied to a Process

Lee et al. proposed a dataflow process network [24–26] that extends the KPN by incorporating the notion of firing which was first introduced by Dennis [27]. In a dataflow process network, the behavior of a process is specified with a set of firings instead of a sequential program with the functions `get()` and `put()`. A firing is an atomic computation that consumes a finite number of input data items and produce a finite number of output data items. A firing is invoked if and only if its associated firing rule is satisfied.

A dataflow process with  $m$  input ports and  $n$  output ports has a set  $U = \{R_1, R_2, \dots, R_k\}$  of firing rules. A firing rule  $R_i = (r_{i,1}, r_{i,2}, \dots, r_{i,m})$  is a tuple that consists of finite sequences  $r_{i,j}$  of data items that will be consumed from the  $j$ th input port when the process fires. A firing rule  $R_i$  is satisfied if and only if each sequence  $r_{i,j}$  in  $R_i$  forms a prefix of the sequence of unconsumed data items on the channel connected to the  $j$ th input port.

Lee et al. also showed that a sufficient condition for a dataflow process network to be determinate is that all processes in the network are functional and a set of firing rules of each process is sequential [24]. A process is functional if it is free from side effects, i.e., the outputs of the process firing are purely a function of the

inputs. A set of firing rules is sequential if the outputs are independent of how a choice between firing rules is made when two or more firing rules are satisfied at the same time.

## **2.3 Programming Framework for an Autonomous Machine**

There is a plethora of graphical programming frameworks that provides programming abstraction to programmers in the development process of an autonomous machine. Representative examples include RTMaps, Simulink and Ptolemy II [11–13]. Like Splash, these frameworks are more or less based on the KPN model and have some extensions to satisfy engineering needs that arise during production–quality system development.

We set four design goals for these programming frameworks: (1) they should provide easy–to–use, effective programming abstraction, (2) they must support real–time stream processing for machine intelligence, (3) they must provide programming supports for a real–time control system and (4) they should support performance optimization on distributed multicore computing platform. We analyze the pros and cons of the above frameworks with respect to these design goals.

Simulink is a commercial modeling framework which is widely used particularly in automatic control and signal processing systems. It offers as a primary programming abstraction a time–driven process network model in that a process is triggered at periodic

sampling time points specified by programmers. Simulink supports the event-driven programming style as well, via an event port. However, Simulink has several limitations to be used in the development of an autonomous machine. First, it does not provide programming abstraction for data-driven programming style that is commonly used for stream processing. Second, it does not support timing constraints annotation and handling which is essential for supporting real-time stream processing. Third, it does not provide language constructs for mode switch and exception handling that is needed for the development of real-time control systems. Finally, it does not have an explicit concurrency model inside a process.

RTMaps is well suited for the development of an application that must deal with multiple sensors and actuators like an autonomous machine. 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 also has several limitations to be used in the development of an autonomous machine. First, RTMaps does not offer a language construct for asynchronous event notification and handling. Second, it does not consider a rate constraint in an explicit manner. Thus, programmers are left with a burden to implement a rate control mechanism in user-level code by themselves; or they need to rely on time-driven, periodic task invocation to maintain a desired output rate. Either way, lower-level

implementation details are exposed to users in the programming abstraction of RTMaps. Third, it lacks support for real-time control systems such as mode switch and exception handling. Finally, RTMaps does not make a concurrency model explicit inside a process, leaving programmers responsible for thread creation and synchronization.

Ptolemy II is an academic programming framework that can support a wide variety of process network models. It even allows programmers to create an application using several different models at the same time. Ptolemy II offers rich support for imperative programming such as mode switch and exception handling. However, most of the process network models of Ptolemy II lack support for real-time stream processing. Only Ptide which is an experimental model for academic research allows a freshness constraint to be specified for a sensor value [28]. But it does not support a rate constraint or a correlation constraint. Like RTMaps, Ptolemy II does not specify a concurrency model or thread-to-core allocation inside a process. Simply, it maps each process to a Java thread and delegates thread scheduling to the underlying operating system.

## **2.4 Runtime Software for an Autonomous Machine**

Since existing programming frameworks have limitations in the development of an autonomous machine, many companies and laboratories choose to develop autonomous machines without using

these programming frameworks. ROS is a representative open-source runtime software system that is commonly used to develop an autonomous machine [29]. ROS comes with a publish-subscribe communication mechanism for transferring data items between processes in distributed systems and provides an interface for programmers to easily use it. Also, it supports ROS package that allows programmers to provide their own programs to other developers as libraries. ROS has become a representative software framework for the development of autonomous machines due to its easy-to-use communication interface and vast developers' community based on the ROS package.

However, when developing an autonomous machine using ROS, most implementation issues except for inter-process communication must be dealt with by the programmer. For example, ROS provides no support for specifying and handling end-to-end timing constraints. As a result, tuning and exception handling to satisfy timing constraints should be carried out by the programmer himself. Also, functions such as sensor fusion, mode change, and exception processing are not supported by ROS.

In order to overcome the weaknesses of ROS, ROS 2 is currently under development based on data distribution service (DDS), a communication standard that supports real-time publish-subscribe communication [15,30]. ROS 2 utilizes quality of service (QoS) policies of DDS to provide features to reduce communication latency or increase reliability of the transmission. However, each function is

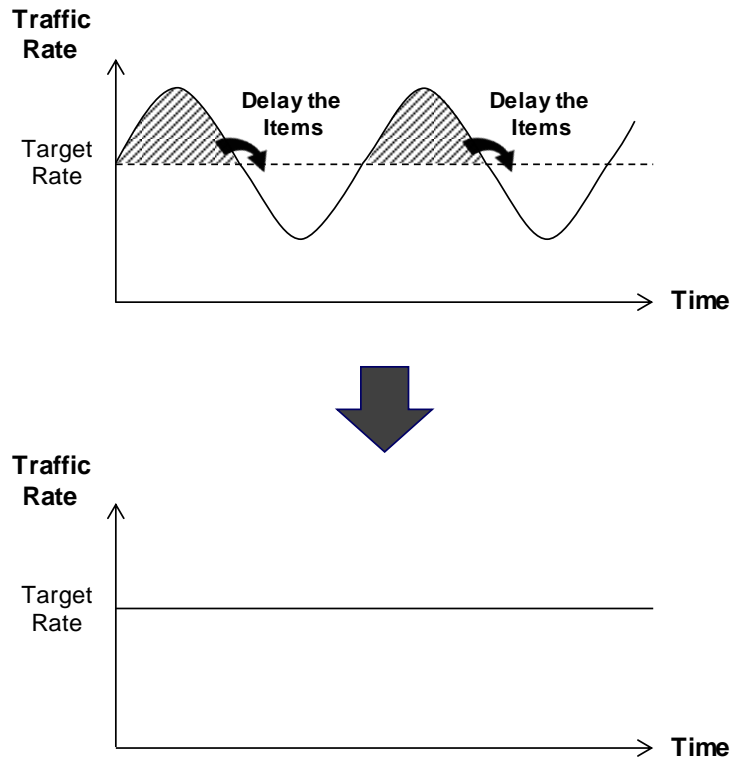
not yet fully implemented and verified since it is still in the early stage of development [31].

## **2.5 Rate Control**

In the processing of stream data items, serious problems such as uncontrolled jitter and an unbounded queue can occur. Variability in communication delay and execution time in a physical system can easily cause bursty data traffic on communication channels and eventually deteriorate the resultant control quality to a significant degree.

Rate control is a technique that prevents bursty data traffic by limiting the number of output data items that are generated per unit time. Existing approaches on rate control can be classified as traffic shaping and traffic policing, depending on how the bursty data traffic is handled. Traffic shaping is a technique that delays the output data items that are generated above the target rate, as shown in Figure 2. In order to implement traffic shaping mechanism, we need a buffer to temporarily store delayed data items. Traffic shaping has the advantage of low data item loss because it stores the data items instead of discarding them, but there is the disadvantage that additional delays occur while storing the data items in the buffer.

Traffic policing is a technique that drops the data items that are generated above the target rate, as shown in Figure 3. Traffic policing incurs less additional delays compared to traffic shaping because data



**Figure 2. Traffic shaping.**

items are not stored in the buffer. However, the data item is always lost if the data items are generated above the target rate.

The loss of data items on an autonomous machine is undesirable since it causes a performance degradation of the control system. Therefore, Splash chooses traffic shaping among the two rate control methods to reduce the loss of data items. In addition, Splash includes a mechanism to limit delays that occur during traffic shaping. The Splash's traffic shaper checks whether there will be a violation of a freshness constraint before putting the data item into the buffer. It only puts the data item in the buffer if there will be no violation of the freshness constraint. If the violation might occur, it immediately



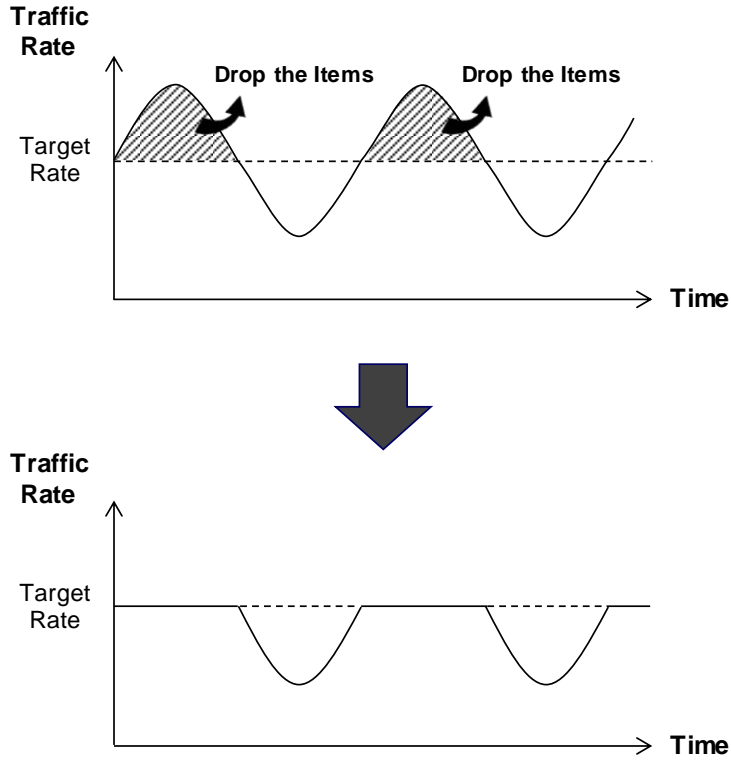


Figure 3. Traffic policing.

throws away the data item.

In the following sections, we introduce existing studies about traffic shaping and traffic policing.

### 2.5.1 Traffic Shaping

Many stream processing frameworks support traffic shaping that limits output stream rate at predefined target rate [32-34]. Tolosana–Calasanz et al. proposed a traffic shaper using token bucket algorithm that is one of the most fundamental algorithm used for rate control [32]. A token bucket–based traffic shaper has three

parameters:  $R$ ,  $C$  and  $b$ . Among three parameters, token generation rate  $R$  and buffer capacity  $C$  are predefined constants. On the other hand, bucket size  $b$  is a variable that changes during the execution of the algorithm.  $b$  is initialized to zero at the beginning of the algorithm and increases at a rate of  $R$  over time until  $b$  reaches  $C$ . When the token bucket-based traffic shaper receives a request to output a data item, it checks whether  $b$  is larger than the size of the data item to be produced. If  $b$  is larger, the traffic shaper subtracts  $b$  by the size of the data item and outputs the data item. If  $b$  is not larger, it puts the data item into the buffer and wait until  $b$  becomes large enough to be produced.

Amini et al. proposed a resource management and traffic shaping technique to maximize throughput on distributed stream processing systems [33]. In the first step, it determines on which processing node the processing elements are to be executed. A processing element the basic unit of stream processing. In the second step, the target input and output rate of each processing element is derived based on the placement of the processing elements. It then uses token bucket-based traffic shapers to meet the target rates. However, [32,33] have no guarantees about the delay incurred while storing data items in a buffer, which is an inherent problem of traffic shaping, and thus there is a limitation to use them in the real-time stream processing applications.

In order to overcome this limitation, Ernesto conducted a study to bound the maximum delay and the maximum buffer size of traffic

shaper [34]. They used real-time calculus that extends network calculus during the analysis of traffic shaper [35,36]. The proposed approach takes two inputs: (1) upper and lower arrival curve that specifies the maximum and minimum number of incoming data items of each stream input and (2) upper and lower service curve that specifies the maximum and minimum capacity of each resource. These inputs are then used to calculate the maximum latency and maximum buffer size of the traffic shaper. However, this approach also has limitations since it is difficult to specify tight upper and lower arrival and service curves in many practical applications.

All the aforementioned approaches perform traffic shaping using a predetermined target rate. There are traffic shaping mechanisms that vary the target rate at runtime. One of the representative mechanisms is RADAR [37] which is designed for distributed stream processing systems. This approach monitors application delays and system loads, and dynamically determines the target rate of the traffic shaper based on the monitored information. While doing so, it utilizes Lagrange Multiplier technique to maximize the system utilization of the target system.

### 2.5.2 Traffic Policing

[38,39] introduced traffic policing mechanisms that drop data items when a predefined target rate exceeds. However, these approaches are rarely used recently since they cause excessive data item loss

every time bursty data traffic occurs.

In contrast, many traffic policing approaches have been proposed to detect bursty data traffic without a predefined constant target rate. This type of techniques is also called load shedding. Aurora is one of the representative database management systems (DBMS) for stream data that supports load shedding [40]. It provides programmers with continuous queries for manipulating stream data, and a runtime system that processes the requested queries efficiently. Aurora lets programmers specify a QoS (quality of service) function that takes output delays, data item loss rate, output values as input. When bursty data traffic occurs, the runtime system drops data items in such a way that the QoS is maximized.

Simmhan et al. proposed a traffic policing mechanism that adjust target rate based on the application's context [41]. A programmer sets the minimum and maximum threshold of the target rate and describes a policy that adjusts the target rate. Then, the proposed approach automatically updates the target rate according to the policy at runtime.

## **2.6 Sensor Fusion**

Multisensor data fusion, or sensor fusion, is a technique that estimates information about nearby situation by processing data from multiple sensors [5,42]. Sensor fusion-based algorithms are widely used in real-time control systems since they have higher accuracy,

reliability and robustness than algorithms using a single sensor. For example, an autonomous vehicle, one of the representative autonomous machines, fuses various sensors such as camera, LiDAR and radar to perform recognition algorithms such as object detection and localization [43–45].

The research on sensor fusion can be divided into measurement fusion and situation fusion. The measurement fusion receives raw measurement data from each sensor and performs sensor fusion [43,46–48]. In contrast, the situation fusion first estimates the situation individually for each sensor, and then uses the results to perform sensor fusion [44,45,49]. In the following sections, we introduce existing studies about measurement fusion and situation fusion.

### 2.6.1 Measurement Fusion

The measurement fusion takes a set  $\{z_1, z_2, \dots, z_m\}$  of measurements from each sensor as inputs and outputs an estimation  $\tilde{s}$  of the current situation  $s$  as shown in Figure 4. It is also called low-level sensor fusion because it performs sensor fusion on raw sensor data. Programmers have high degree of freedom when developing algorithms with the measurement fusion because they have access to raw data from all sensors. However, the measurement fusion has limitations to be used in the distributed systems. First, it is difficult to distribute work throughout machine since the process for fusion

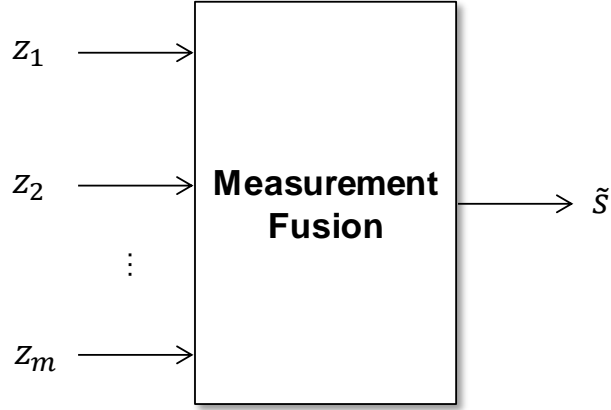


Figure 4. Measurement fusion.

should do all the work for estimation alone. Second, excessive communication overhead occurs in transferring large raw data between processes.

The most primitive form of measurement fusion is stateless fusion. In the stateless fusion, an estimation  $\tilde{s}$  of the situation  $s$  is computed through a pure function of a set  $\{z_1, z_2, \dots, z_m\}$  of measurements currently arrived. It does not use any state during the estimation. However, these stateless fusion algorithms are vulnerable to sensor's noise or malfunctions because they do not utilize past results of the estimation. Therefore, they are rarely used in recent research on sensor fusion.

In the other hand, state fusion computes an estimation  $\tilde{s}$  of the situation  $s$  using a function of a set  $\{z_1, z_2, \dots, z_m\}$  of measurements and its internal state  $\varphi$ . Unlike stateless fusion, state fusion outputs a more stable and robust results since past estimations are reflected in the state  $\varphi$ .

Drolet et al. proposed a positioning system using an underwater

positioning sensor and an accelerometer for underwater ROV (remotely operated vehicle) [46]. The proposed approach periodically invokes a fusion algorithm that estimates the current position of the ROV using the Kalman filter [50,51]. Since measurements may not yet arrive from some sensors at the time of invocation, the fusion algorithm performs estimation by selecting one of Kalman filters according to the currently available input combination.

Liu et al. proposed a sensor fusion-based moving object detection and tracking for self-driving cars [47]. This approach takes measurements from camera and radar sensor as inputs and estimates the position, velocity and acceleration of objects near the ego vehicle. Unlike [46], Liu took the data-driven processing. The fusion algorithm is invoked whenever a measurement from the radar sensor is received. This is because the measurement frequency of the radar is relatively lower than that of the camera. On invocation, the fusion algorithm performs estimation using the radar input and a set of the camera inputs that have arrived after the previous invocation.

Geneva et al. presented a localization technique that estimates the current position of an autonomous vehicle using GPS, camera and LiDAR sensors [48]. Similar to [47], this approach also calls a fusion algorithm when the measurement from the LiDAR, which has the lowest frequency, comes in. The proposed approach additionally performs linear interpolation and extrapolation on the sensor inputs

that did not arrived on invocation and use them as the inputs of the algorithm.

Cho et al. proposed a multi-sensor fusion system for moving object detection and tracking in urban driving environments [43]. The proposed approach takes cameras, LiDAR and radar sensors as inputs and outputs the estimation of position, velocity and acceleration of nearby objects. It also takes the data-driven processing, but it invokes the fusion algorithm every time a measurement comes in from any one of sensors.

As a result of analyzing the existing approaches on measurement fusion, the programming framework should support two important implementation issues: (1) it should allow the programmer to specify the triggering condition for the fusion algorithm and (2) it must enable the programmer to determine which of the measurement from each sensor to select as input to the fusion algorithm. In Splash, we provide a language construct named fusion operator to meet these requirements. More information about the fusion operator is covered in Chapter 5.

### 2.6.2 Situation Fusion

The situation fusion first computes the estimation  $\tilde{s}_i$  of the situation  $s$  using the measurement  $z_i$  from each sensor, then fuses a set of estimations  $\{\tilde{s}_1, \tilde{s}_2, \dots, \tilde{s}_m\}$  of  $s$  to compute the final estimation  $\tilde{s}$ . Figure 5 shows the overall architecture of the situation fusion. It is



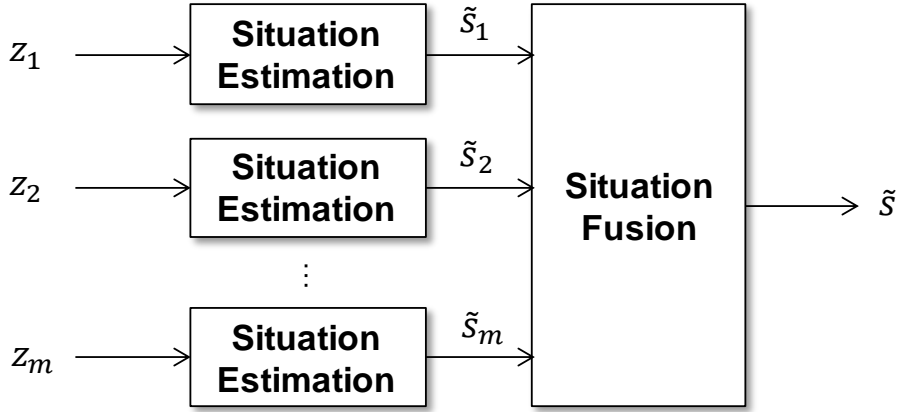


Figure 5. Situation fusion.

also called high-level sensor fusion since it performs sensor fusion on the high-level data that has already been processed once. Programmers have relatively low degree of freedom when developing algorithms with the situation fusion. However, the situation fusion has advantages to be used in the distributed systems. First, it is easier to distribute work throughout machine since the tasks for estimation are divided between the processes of estimating using individual sensors and the process of fusing individual estimations. Second, the amount of data communication is reduced because the size of estimation  $\tilde{s}_i$  is smaller than the measurement  $z_i$  in most cases [42].

In the situation fusion, each sensor's noise or malfunction is handled in per-sensor estimation phase. Therefore, the situation fusion assumes that reliable and robust estimations are taken as inputs and uses stateless fusion to merge a set of estimations.

Floudas et al. proposed two sensor fusion techniques for object detection in self-driving cars using the camera and radar sensors

[49]. First, they introduced the track-level fusion that first estimates the position and size of objects on the road using each sensor, then merges them together using multidimensional data association [52]. Second, they proposed the grid-based fusion that computes occupancy grid which maintains probabilistic estimates of the occupancy state of each cell in a lattice for each sensor, and combines them together using Bayesian inference [53].

Chavez-Garcia and Aycard proposed a sensor fusion method that merges the results of object detection using camera, radar and LiDAR sensors [45]. Similar to [49], they used the data association technique. The proposed approach determines whether two objects detected by different sensors are identical by using information such as the location, shape and type of the objects.

Zhang et al. presented a semantic segmentation method using camera and LiDAR sensor for autonomous vehicle [44]. The proposed approach uses separate classifiers in each sensor to perform semantic segmentation. It then trains an additional classifier named a fusion classifier with previous classifier outputs incorporated as input features for regions with overlapping sensor coverage. They used the stacking hierarchical labeling to train the fusion classifier [54].

Existing studies on situation fusion assumes that they use temporally synchronized sensors as inputs. However, to be deployed to the real platform, they must handle implementation issues caused by asynchronous sensors. Like the measurement fusion, the situation

fusion also should allow the programmer to specify the triggering condition and enable the programmer to determine which of the measurement from each sensor to select as inputs.

## **2.7 Mode Change**

The functional and non-functional requirements of the real-time control system of an autonomous machine can vary depending on the situation. One of the example system with variable requirements is an aircraft control system. It has different requirements depending on the situation of the aircraft, such as take-off, normal cruise, landing and emergency [55].

The real-time control systems with varying requirements must support a multimode system which can change its internal execution logic depending on the situation [55-57]. A multimode system is defined through a set of modes and a set of mode changes. Each mode is presented by a set of tasks that are executed in the mode. One of these modes is designated as the initial mode. Each mode change specifies which mode the system should change to when a particular event occurs.

In the mode change process, there is a transient state where the tasks of the previous mode remain and the tasks of the next mode are not prepared. Mode change techniques are classified into two categories depending on how the transient state is handled. The first category is a synchronous mode change that starts the execution of

next mode tasks after completing the previous mode tasks. This type of mode change has the disadvantage that the start time of the next mode tasks is relatively late, but there is no performance interference between the previous mode tasks and the next mode tasks.

The second category is an asynchronous mode change that starts the execution of next mode tasks before completing the previous mode tasks. The asynchronous mode change can start the next mode tasks faster than the synchronous mode change. However, there is performance interference between the previous mode tasks and the next mode tasks. In order to use asynchronous mode change in real-time control systems, delays due to these performance interferences should be analyzed in advance to ensure that all tasks complete their execution in time.

Splash selects a synchronous mode change that can prevent performance interference between tasks in the previous and next modes. To overcome the limitations of the synchronous mode change, Splash provides a runtime mechanism that tries to finish the execution of the previous mode tasks as soon as possible.

The following subsections explain existing approaches about the synchronous and asynchronous mode change.

### **2.7.1 Synchronous Mode Change**

Tindell and Alonso proposed a simple synchronous mode change

mechanism that waits for all CPUs to become idle before starting the tasks in the next mode [58]. When all CPUs became idle, a task named mode changer prevents further execution of previous mode tasks and starts running the next mode tasks. The proposed approach takes a long time to change modes since it should wait indefinitely until all the tasks in the previous modes are finished.

Real proposed a synchronous mode change that reduces the delay incurred during the mode change [59]. It prevents tasks of previous mode from starting additional job after a mode change event occurs. When the tasks in the previous mode are finished, the next mode tasks are started. Unlike [58], the mode change delay of this approach is bounded to a constant.

### **2.7.2 Asynchronous Mode Change**

Sha et al. proposed an asynchronous mode change scheme [62] for systems that use rate monotonic scheduling [60] and priority inheritance protocol [61]. This approach performs a schedulability analysis based on the current CPU utilization when the next mode task is requested. It allows the task to start only when the schedulability analysis is passed. If not, it should wait for the tasks in the previous mode to be completed and it passes the schedulability analysis.

Tindell et al. proposed an asynchronous mode change mechanism [64] for systems that use deadline monotonic scheduling [63]. The

proposed approach prevents the previous mode tasks from starting a new job after a mode change event occurs, and allows the execution of the next mode tasks after the end of the previous mode tasks' period. They provided a method to perform a schedulability analysis for the proposed approach so that it can be used in real-time systems.

## Chapter 3. Motivation and Contributions

This chapter explains the motivation of the Splash. First, Section 3.1 defines the problems we are trying to solve. Then, Section 3.2 explains the limitations of the KPN. Finally, Section 3.3 describes the main contribution of this dissertation.

### 3.1 Problem Description

This dissertation aims to propose Splash, a new graphical programming framework for autonomous machines. Splash should achieve four key design goals. First, it should provide an easy-to-use, effective programming abstraction that can hide implementation details and supports a model-based code generation capability. Second, it must support real-time stream processing for deep-learning based machine learning intelligence. Third, it must provide programming support for real-time control system of autonomous machine such as sensor fusion and mode change. Finally, it needs to support performance optimization of software system running on a heterogeneous multicore distributed computing platform.

We describe the issues that Splash must address to achieve its four goals. First, Splash must provide programming abstraction for three distinct programming style: time-triggered, event-triggered and data-triggered. Programming an autonomous machine is a collaborative effort among developers having diverse technical

backgrounds such as control engineers, software programmers, and AI engineers. Whereas control engineers favor time-driven triggering such that periodically invoked tasks execute the control algorithms, AI engineers prefer data-driven triggering such that an incoming data item on a channel wakes up a handler task. On the other hand, software programmers often rely on event-driven triggering. In order for Splash to effectively support collaboration between them, all three programming styles must be supported without restriction. Also, support for integration between them should be provided.

Second, Splash must support specifying and handling end-to-end timing constraints for real-time stream processing. Programmers should be able to specify the genuine, end-to-end timing constraints: freshness constraint, correlation constraint and rate constraint while developing the Splash program [65]. Also, it should provide a best-effort runtime system to satisfy the timing constraints annotated in the program, and exception handling mechanism to monitor and handle violations of such constraints.

Third, Splash should provide programming support for real-time control systems, such as sensor fusion and mode change. Since many sensors on autonomous machines are not timely synchronized, complex implementation issues arise while performing sensor fusion, such as determining the triggering condition, selecting input data items for the fusion algorithm and handling timeout. Similarly, in the case of mode change, complex implementation issues arise in order to develop safe and fast mode change mechanism, such as processing



data items in transition state and ensuring consistency of shared data. Splash should provide a programming abstraction that can hide these implementation details.

Finally, Splash should provide multithreaded process model to exploit parallelism explicitly from the distributed multicore computing platform. In addition, it should help programmers easily determine where to execute the processes and threads on the distributed multicore computing platforms.

### **3.2 Limitations of Kahn Process Network**

Splash is based on the KPN, a process network that offers a programming model in that developers are allowed to write an application in a parallel way such that constituent processes are independently written, allocated and executed on a multiprocessor system. The KPN model helps developers avoid error-prone issues such as data races and non-determinism. In accordance with these advantages, many programming models provided a programming abstraction based on the KPN [24,66–69].

However, the pure form of the KPN cannot be used directly in the programming framework for the automatic machine because it fails to achieve our design goals. First, the KPN only supports the data-driven processing and does not support the time-driven or event-driven processing. Also, the KPN has inherent practical limitations in terms of the expressibility of program logic and the

performance of a resultant system. This is because the KPN model is based on many simplifying restrictions such as freedom of global side effects to achieve determinacy.

Second, the KPN lacks support for real-time processing. It does not support the specification and handling of end-to-end timing constraints. Therefore, developers still have to resort to time-consuming and error-prone manual tuning in the implementation phase of an autonomous machine to meet such timing constraints.

Third, the KPN provides not support for sensor fusion or mode change. Therefore, when developing a sensor fusion algorithm, programmers must deal with complex implementation issues such as triggering the algorithm, selecting input data items and handling timeouts. Also, when implementing a mode change mechanism, programmers should handle implementation issues such as keeping consistency of shared data.

Finally, the KPN does not support multithreaded process model, leaving programmers responsible for thread creation and synchronization. In addition, it does not support any programming abstraction for performance optimization in distributed multicore computing platforms.

As a consequence, they are still in need of a high level programming paradigm that has a versatile programming abstraction for specifying the complex software architecture of an autonomous machine. We present a new programming framework named Splash to address such grave problems arising in programming an

autonomous machine. Splash eliminates the determinism, one of the benefits of the KPN, but instead effectively achieves all four core goals of this paper.

### **3.3 Contributions of this Dissertation**

In this thesis, we propose a new graphical programming framework for an autonomous machine that overcomes the KPN's limitations described in the previous section. The main technical contributions can be summarized as below.

- **Providing a best-effort runtime system that tries to meet the annotated timing constraints and exception handling mechanisms to monitor the violation of such constraints**
  - We propose a graphical programming language that allows developers to specify three genuine end-to-end timing constraints: freshness constraint, correlation constraint and rate constraint. Splash provides a best-effort runtime system to satisfy the timing constraints annotated in the program. It also supports exception handling mechanism to monitor and handle timing constraint violations at runtime.
- **Introduction of the sthreads and the build units for development in distributed multicore computing platforms**
  - In order to exploit parallelism explicitly from the

underlying operating system and distributed multicore computing platform, Splash offers a multithreaded process model. In the multithreaded process model, a processing component consists of a group of sthreads that are logical entities of independent execution. As an sthread is an abstract entity, it needs to be mapped to a process and a thread of an underlying operating system during the system implementation process. To facilitate this process, Splash offers an allocation entity called a build unit.

- **Integrating three distinct triggering styles: time-driven, data-driven and event-driven**
  - To be an effective programming language with sufficient expressibility, Splash supports all the three triggering styles in a unified manner. Among the three, Splash takes the data-driven triggering as the default style, unless specified otherwise. We notice that the data-driven triggering in its purest form is not the most suitable programming abstraction for autonomous machine developers since it may have serious side effects such as uncontrolled jitter and an unbounded FIFO queue. To solve this problem, Splash offers rate control semantics.
- **Introduction of the fusion operator to effectively handle timing issues in sensor fusion**
  - We propose a language construct named fusion operator

to automatically handle the complex time synchronization issues of sensor fusion. The fusion operator provides a fusion rule and a fusion function that can be used by programmers to clearly specify the triggering condition and input data item selection policy of the fusion algorithm. We also introduce a runtime mechanism that automatically satisfies the conditions specified using the fusion operator.

- **Support for mode change to satisfy the variable requirements of the real-time control systems**
  - Splash provides a language construct named multimode factory to support multiple modes of operations. A programmer can describe the behavior of each mode and the specification of mode change. The Splash runtime system then performs mode changes according to the programmer's specification. During the mode change, the consistency of data used by sthreads is preserved.

## Chapter 4. Underlying Timing Semantics of Splash

Time is a first-class entity in Splash. This chapter explains Splash's underlying timing semantics that is the basis for the all other language semantics provided by the Splash. Section 4.1 introduces three genuine end-to-end timing constraints required by autonomous machines. Section 4.2 describes the global time base that Splash provides to handle end-to-end timing constraints. This section also introduces in-order delivery semantics that is the most basic programming abstraction provided by Splash. Finally, Section 4.3 explains three distinct computing models which Splash deals with.

### 4.1 End-to-End Timing Constraints

Splash supports three types of genuine, end-to-end timing constraints [65].

- (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 can explicitly annotate these three types of timing constraints via language constructs in a Splash program. The Splash runtime provides mechanisms to satisfy annotated timing constraints as much as possible. It also raises an exception if it detects the violation of the timing constraint at runtime.

## 4.2 Global Time Base and In-order Delivery

Reading the time in a Splash program is supported by an abstract global clock that is possibly implemented via distributed local clock synchronization such as precision time protocol(PTP) [70,71]. 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 provided by the abstract global clock. The ways of handling each type of timing constraint is covered in more detail in the next chapter.

### **4.3 Integrating Three Distinct Computing Models**

Programming an autonomous machine is a collaborative effort among developers having diverse technical backgrounds such as control engineers, software programmers and AI engineers. Whereas control engineers favor time-driven triggering such that periodically invoked tasks execute the control algorithms, AI engineers prefer data-driven triggering such that an incoming data item on a channel wakes up a handler task. On the other hand, software programmers often rely on event-driven triggering. To be an effective programming language with sufficient expressibility, Splash supports all the three triggering styles in a unified manner.

Among the three, Splash takes the data-driven processing as the default style, unless specified otherwise in a program. The data-driven processing is the most commonly used programming style in data stream processing [72,73]. However, we noticed that the data-driven triggering, in its purest form, was not the most suitable programming abstraction for an autonomous machine since it may have serious side effects such as uncontrolled jitter and an



unbounded FIFO queue on a port. Variability in communication delay and execution time in a physical system can easily cause bursty data traffic on communication channels and eventually deteriorate the resultant control quality to a significant degree. To solve these problems, Splash provides a programming abstraction for rate control. The details are explained in the next chapter.

## Chapter 5. Splash Language Constructs

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 6 shows the hierarchical relationships among the diverse Splash components in the UML diagram format.

The Splash component can have stream input ports and stream output ports. 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 7 shows a sample Splash

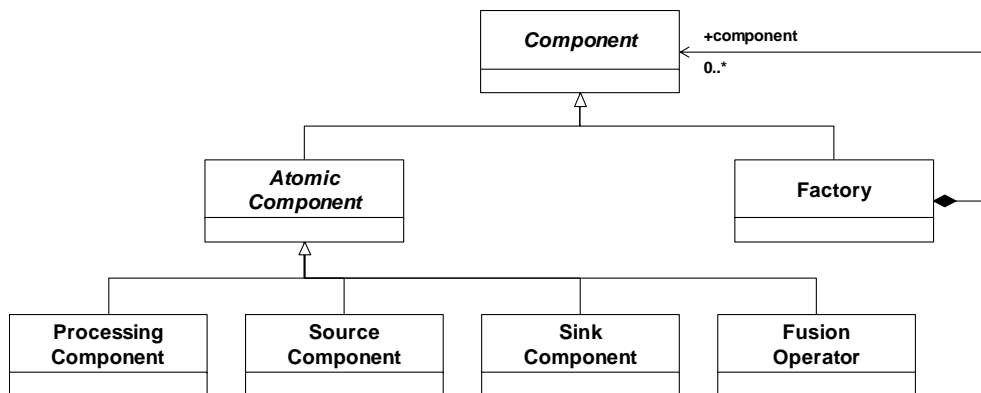


Figure 6. Hierarchy of Splash components.

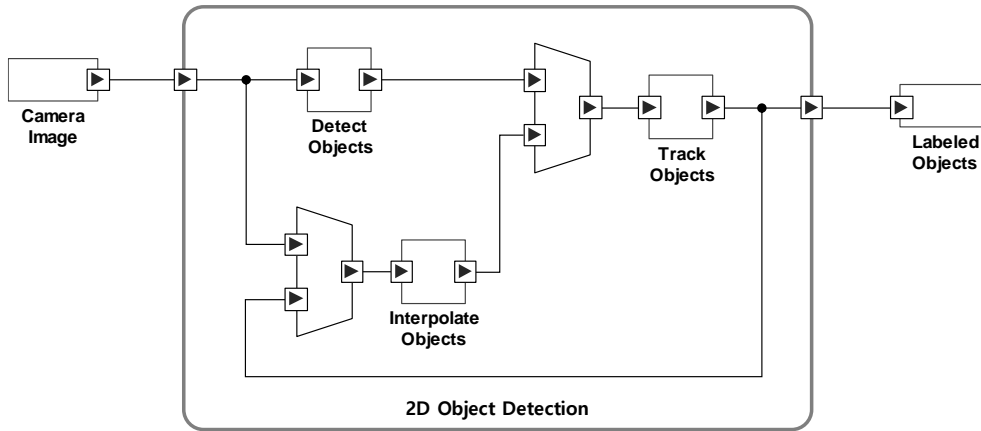


Figure 7. Splash program example: 2D object detection.

program that consists of various components, channels and ports.

## 5.1 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 8 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

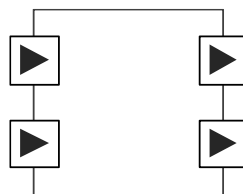


Figure 8. Processing component.

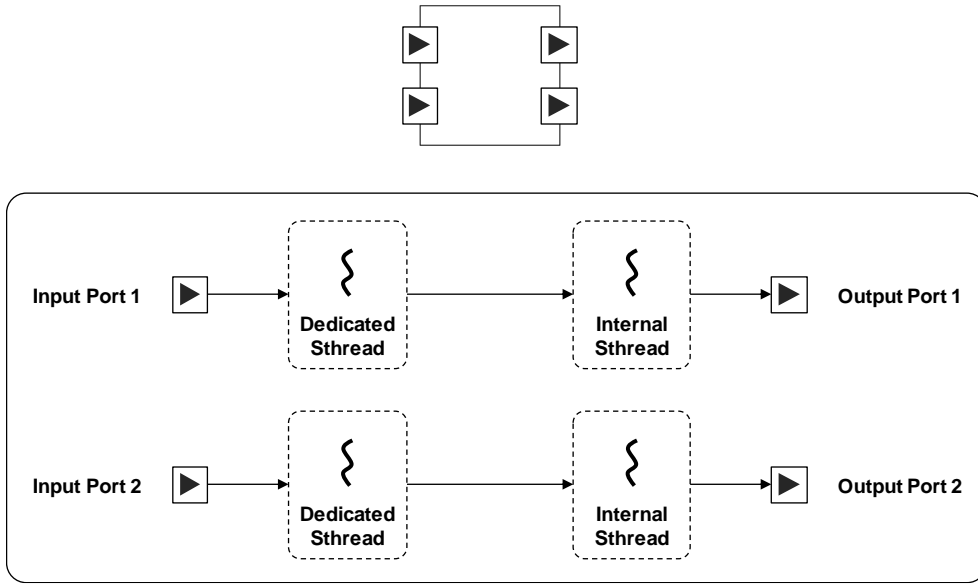


Figure 9. Processing component and its sthreads.

multithreaded process model. 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. The sthreads are classified into dedicated sthreads that are attached to each stream input port and internal sthreads that are generated from other sthreads. When using an internal sthread, a programmer should specify which sthread the internal sthread is created from. The programmer must also specify which stream output ports each sthread writes to. Then, Splash automatically generates code based on information specified by the programmer. Figure 9 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 [74].



**Figure 10. 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 10 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.

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



**Figure 11. Sink component.**

representation of a sink component is shown in Figure 11.







## 5.2 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 in Table 1.

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 12. Each port type is associated with one of three port interfaces: stream, event and mode change port interfaces.

**Table 1. Graphical Symbols for Ports**

Port Type	Input	Output
Stream Port		
Event Port		
Mode Change Port		

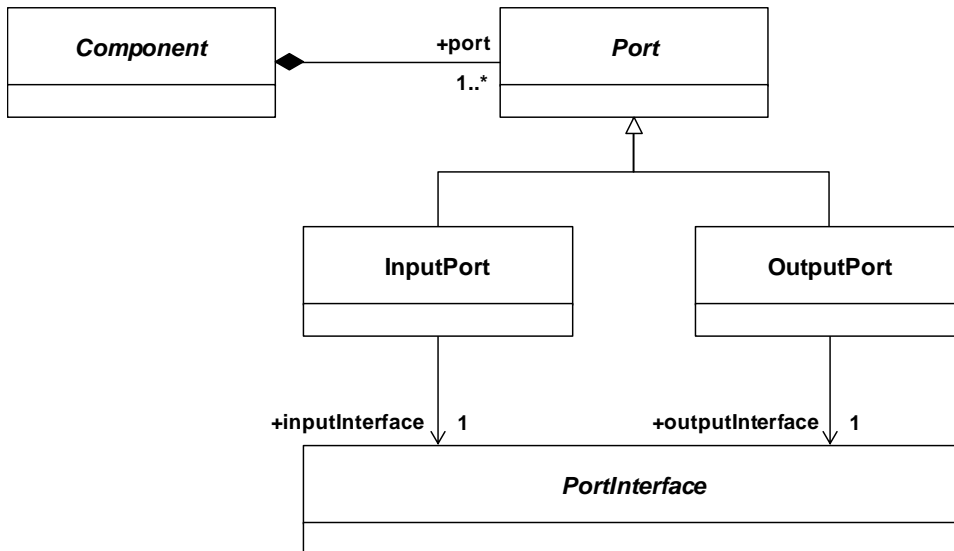


Figure 12. Input and output ports as subtype of port.

Clearly, an output port and an input port connected by a channel or a link must share the same port interface. Figure 13 shows the three port interfaces. As in the figure, 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

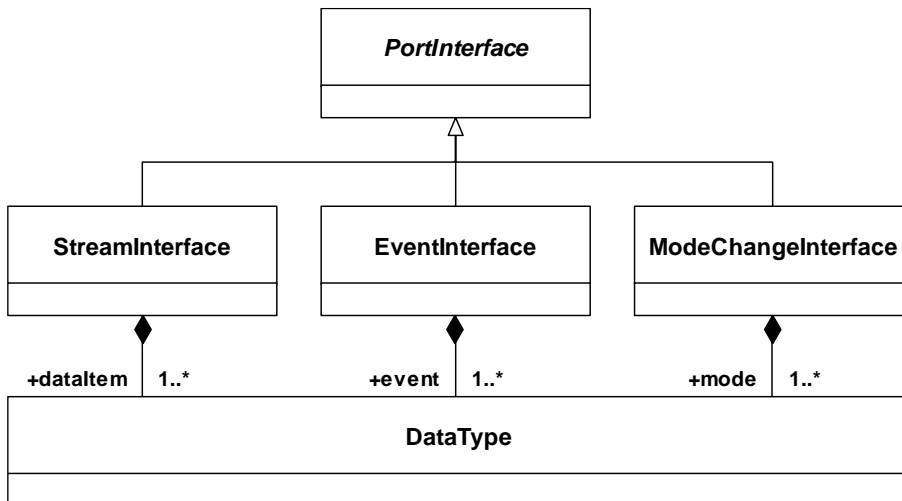


Figure 13. Hierarchy of port interfaces.

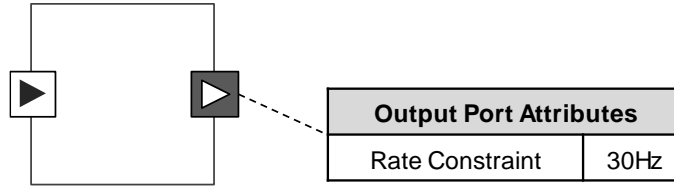


Figure 14. Rate-controlled stream output port.

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. Also, programmers can selectively specify a stream output port as a rate-controlled stream output port. Then the Splash code generator transparently attaches a rate-controlling module to the stream output port. We call the module a rate controller, which reduces jitter and bounds the maximum FIFO queue size, at the cost of tolerable delay of a data item. The existence of a rate controller is hidden from programmers. Figure 14 pictorially depicts a stream output port annotated with a desired data generation rate. The stream output ports that have been augmented with rate controllers are marked with a different symbol for the user to distinguish from the ones that are not.

By definition, a stream output port with a rate constraint  $r$  is assigned a time window of size  $1/r$ . The semantics of a rate-controlled stream output port is that it will guarantee the production of exactly one data item in each time window. More formally, the port will generate one data item each time interval  $[t_0 + n/r, t_0 + (n + 1)/r)$



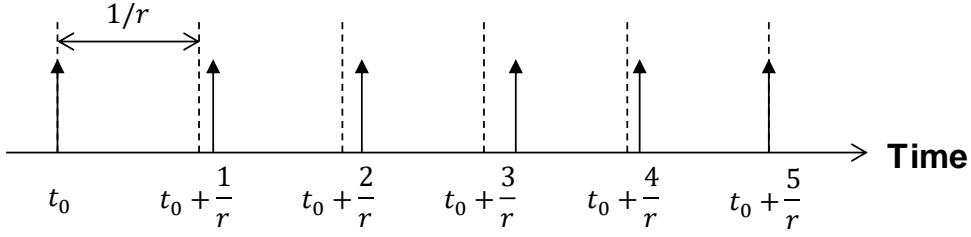


Figure 15. Behavior of a rate controller.

where  $t_0$  is the time when the first data item is generated, and  $n$  is an index starting from 0. Figure 15 shows the behavior of the rate-controlled output stream port.

The rate-controlled stream output port generates two types of outputs. The first output is a genuine data item while the second output is an extrapolation command. A rate-controlled stream output port yields a genuine data time if it has a data item to send within the current time window; otherwise, it outputs an extrapolation command. When a processing component receives an extrapolation command from its stream input port, it must invoke a function that performs a required extrapolation task. Splash enables programmers to write an extrapolation handler for a processing component associated with a stream input port connected to a rate-controlled stream output port.

### 5.3 Channel and Clink

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 16 shows the graphical representation of a channel.

In order to store data items on a channel until they are consumed

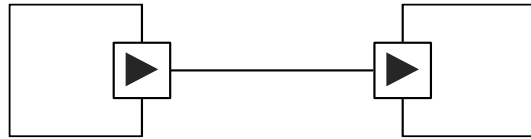


Figure 16. Channel.

by a downstream component, a queue is used. In Splash, a 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. Figure 17 shows a channel with three fan-outs to distinct stream input ports. 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.

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.

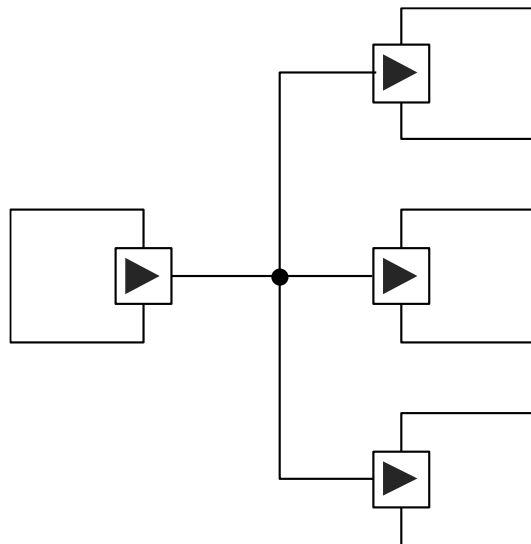
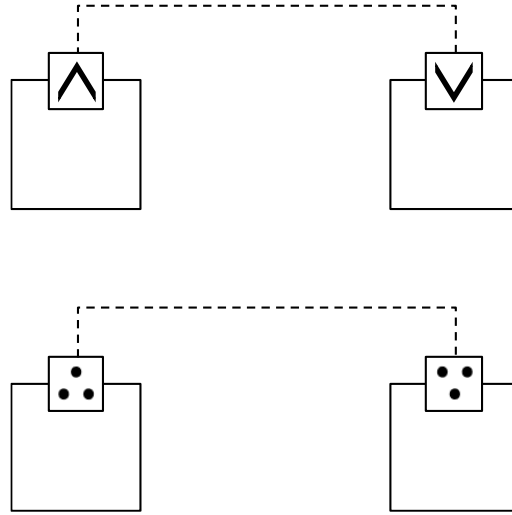


Figure 17. Channel with three fan-outs.



**Figure 18. Clinks.**

Figure 18 shows the graphical representation of a clink between event ports and a clink between mode change ports. Unlike channel, both fan-in and fan-out of a clink can be greater than one.

The graphical presentation of clinks can be omitted to reduce the complexity of the program. In this case, the connections between event ports and mode change ports should be indicated using a table.

## 5.4 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 19. Fusion operators provide a way for programmers to handle the complex implementation issues of sensor fusion algorithms. Specifically, the programmer can use the fusion operator to clearly describe two issues: (1) specifying triggering

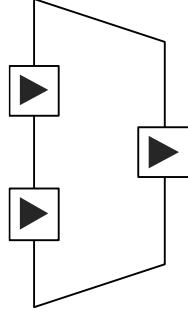


Figure 19. Fusion operator.

conditions of the fusion algorithm and (2) choosing which data items in the input queues to use as inputs for the fusion algorithm.

A fusion operator with a set of  $m$  stream input ports  $P = \{p_1, p_2, \dots, p_m\}$  has a fusion rule  $R$  associated with a fusion function  $f$ . 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 optional ports ( $M \cap O \neq \emptyset$ ). Also,  $\theta$  is optional ports threshold and  $c$  is a correlation constraint. Let  $(d_1, d_2, \dots, d_m)$  be an input tuple where  $d_i$  is a data item placed in the input queue of the port  $p_i$  or an empty data item. If  $d_i$  is an empty data item, we denote it as  $d_i = \perp$ . For  $R$  to be satisfied, there must exist an input tuple  $(d_1, d_2, \dots, d_m)$  that meets the following conditions.

- (1) For any mandatory port  $p_i \in M$ , the data item  $d_i$  is placed in the input queue of  $p_i$
- (2) Let us denote the number of optional ports  $p_i \in O$  where the data item  $d_i$  is placed in the input queue of  $p_i$  as

$n(O, (d_1, d_2, \dots, d_m))$ . Then,  $n(O, (d_1, d_2, \dots, d_m)) \geq \theta_i$ .

(3) For any two data items  $d_i$  and  $d_j$ ,  $|b(d_i) - b(d_j)| \leq c$  where

$b(d)$  is the birthmark of a data item  $d$  ( $d_i \neq \perp$ ,  $d_j \neq \perp$ ).

We call  $(d_1, d_2, \dots, d_m)$  an input tuple that satisfies the fusion rule  $R$ .

A fusion operator invokes its fusion function  $f$  when its fusion rule  $R$  is satisfied. On invocation,  $f$  produces an output data item using an input tuple that satisfies  $R$ . If more than one such input tuple exists, the fusion operator selects one that meets the following two goals.

- (1) A fusion operator first chooses a data item with smaller birthmark to build an input tuple. By processing older data items first, we can reduce the number of freshness constraint violation of data items stored in the input queues.
- (2) A fusion operator tries to select data items from as many optional ports as possible. This is to provide the fusion algorithm with information from as many sensors as possible. However, in the case of conflict between (1) and (2), the fusion operator prioritizes (1).

We denote a set of input tuples that satisfy a fusion rule  $R$  as  $V(R)$ . We now define a binary relation over  $V(R)$  as follows.

**DEFINITION 2:** For any two input tuples  $(d_1, d_2, \dots, d_m) \in V(R)$  and  $(d'_1, d'_2, \dots, d'_m) \in V(R)$  that satisfy a fusion rule  $R = (M, O, \theta, c)$ , we define a binary relation  $(d_1, d_2, \dots, d_m) \leq (d'_1, d'_2, \dots, d'_m)$  on  $V(R)$  by

- For any mandatory port  $p_i \in M$ ,  $b(d_i) \leq b(d'_i)$ .
- Let  $D(O, (d_1, d_2, \dots, d_m))$  be a set of data items stored in the input queue of any optional port among  $d_1, d_2, \dots, d_m$ . Also, let  $l(k, S)$  be a birthmark of the  $k$ th oldest data item in a set  $S$  of data items when  $k \leq |S|$ , and  $\infty$  when  $k > |S|$  ( $\perp \notin S$ ). For any integer  $k$  where  $1 \leq k \leq |O|$ , the following inequality holds.

$$l(k, D(O, (d_1, d_2, \dots, d_m))) \leq l(k, D(O, (d'_1, d'_2, \dots, d'_m)))$$

The fusion operator selects the least input tuple among the elements of  $V(R)$  as defined below.

**DEFINITION 3:** For a set  $V(R)$  of input tuples that satisfy a fusion rule  $R = (M, O, \theta, c)$ , we define a least input tuple  $(d_1, d_2, \dots, d_m)$  as an input tuple that satisfies the following condition.

$$\forall (v_1, v_2, \dots, v_m) \in V(R), (d_1, d_2, \dots, d_m) \leq (v_1, v_2, \dots, v_m)$$

The theorem that follows states that a least input tuple of  $V(R)$  always uniquely exist unless  $V(R) = \emptyset$ .

**THEOREM 1:** For a fusion rule  $R = (M, O, \theta, c)$ , there always exist a unique least input tuple of  $V(R)$  unless  $V(R) = \emptyset$ .

**PROOF:** We first prove the existence a least input tuple of  $V(R)$ . Let  $b_i^{\min}$  be the birthmark of the oldest data item among the  $i$ th elements of input tuples that belong to  $V(R)$ . However, if  $d_i = \perp$  for all input tuples  $(d_1, d_2, \dots, d_m) \in V(R)$ , we set  $b_i^{\min}$  to  $\infty$ . We define an input tuple  $(d'_1, d'_2, \dots, d'_m)$  that satisfies three following conditions.

- For all mandatory ports  $p_i \in M$ , we select a data item  $d'_i$  such that  $b(d'_i) = b_i^{\min}$
- For optional ports  $p_i \in O$  where  $\left| b_i^{\min} - \min_{1 \leq k \leq m} b_k^{\min} \right| \leq c_i$ , we also select a data item  $d'_i$  such that  $b(d'_i) = b_i^{\min}$
- For all input ports  $p_i$  that does not meet the above two conditions, we select an empty data item  $d'_i = \perp$

To prove the existence of the least input tuple in  $V(R)$ , we first show that  $(d'_1, d'_2, \dots, d'_m) \in V(R)$ , then we show that  $(d'_1, d'_2, \dots, d'_m)$  is the least input tuple of a set  $V(R)$  and a binary relation  $\leq$ .  $(d'_1, d'_2, \dots, d'_m)$  satisfies all three conditions of **DEFINITION 1** as follows.

- (1) For all mandatory ports  $p_i \in M$ ,  $d'_i$  is a data item placed in the input queue of  $p_i$ .
- (2) There exists at least one input tuple  $(d''_1, d''_2, \dots, d''_m) \in V(R)$

that contains a data item with the birthmark of  $\min_{1 \leq k \leq m} b_k^{\min}$ .

Since  $n(O, (d_1'', d_2'', \dots, d_m'')) \geq \theta$  and  $n(O, (d_1', d_2', \dots, d_m')) \geq n(O, (d_1'', d_2'', \dots, d_m''))$  hold,  $n(O, (d_1', d_2', \dots, d_m')) \geq \theta$ .

(3) In order to show that  $(d_1', d_2', \dots, d_m')$  satisfies the condition (3)

of **DEFINITION 1**, we prove  $|b(d_i') - \min_{1 \leq k \leq m} b_k^{\min}| \leq c$  holds for

$1 \leq i \leq m$  ( $d_i' \neq \perp$ ). Obviously, this inequality holds for each

optional port  $p_i \in O$ . To prove that the inequality holds for a

mandatory port  $p_i \in M$ , we select an input tuple

$(d_1'', d_2'', \dots, d_m'') \in V(R)$  that contains a data item with the

birthmark of  $\min_{1 \leq k \leq m} b_k^{\min}$ . Since  $|b(d_i'') - \min_{1 \leq k \leq m} b_k^{\min}| \leq c$  and

$b(d_i') \leq b(d_i'')$  hold for  $1 \leq i \leq m$ ,  $|b(d_i') - \min_{1 \leq k \leq m} b_k^{\min}| \leq c$ .

Therefore,  $(d_1', d_2', \dots, d_m') \in V(R)$ .

Now we prove that  $(d_1', d_2', \dots, d_m')$  is the least input tuple of a set  $V(R)$  and a binary relation  $\leq$ . The following two conditions always hold for any input tuple  $(v_1, v_2, \dots, v_m) \in V(R)$ .

- For any mandatory port  $p_i \in M$ ,  $b(d_i') = b_i^{\min} \leq b(v_i)$ .
- $l(k, D(O, (d_1', d_2', \dots, d_m'))) \leq l(k, D(O, (v_1, v_2, \dots, v_m)))$  holds for  $1 \leq k \leq |O|$ .

Therefore, the least input tuple of  $V(R)$  and  $\leq$  always exists if  $V(R) \neq \emptyset$ .



Now we prove the uniqueness of the least input tuple. Suppose for the purpose of contradiction that two distinct input tuples  $(d_1, d_2, \dots, d_m)$  and  $(d'_1, d'_2, \dots, d'_m)$  in  $V(R)$  are both the least input tuples. From the definition of the least input tuple,  $(d_1, d_2, \dots, d_m) \leq (d'_1, d'_2, \dots, d'_m)$  and  $(d'_1, d'_2, \dots, d'_m) \leq (d_1, d_2, \dots, d_m)$ . Therefore, the followings two conditions hold.

- For any mandatory port  $p_i \in M$ ,  $b(d_i) = b(d'_i)$
- $l(k, (d_1, d_2, \dots, d_m)) = l(k, o(R_i, (d'_1, d'_2, \dots, d'_m)))$  for  $1 \leq k \leq |O|$

Since  $D \neq D'$ , there exist two distinct optional ports  $p_i, p_j \in O$  such that  $b(d_i) = b(d'_j)$  and  $d'_i = d_j = \perp$ . There exists an input tuple  $(d''_1, d''_2, \dots, d''_m) \in V(R)$  such that  $d''_r = d'_r$  if  $r = j$ , and  $d''_r = d_r$  if  $r \neq j$  for  $1 \leq r \leq m$ . Then,  $(d_1, d_2, \dots, d_m) \leq (d''_1, d''_2, \dots, d''_m)$  does not hold, which is a contradiction since  $(d_1, d_2, \dots, d_m)$  is the least input tuple. Therefore, the least input tuple of  $V(R)$  and  $\leq$  is unique  $\square$

The fusion operator additionally provides a timeout mechanism that outputs an extrapolation command when its fusion rule is not satisfied for a certain period. If a timeout value is specified for a fusion operator, programmers must write an extrapolation handler in the next processing component connected with this fusion operator.

## 5.5 Factory and Mode Change

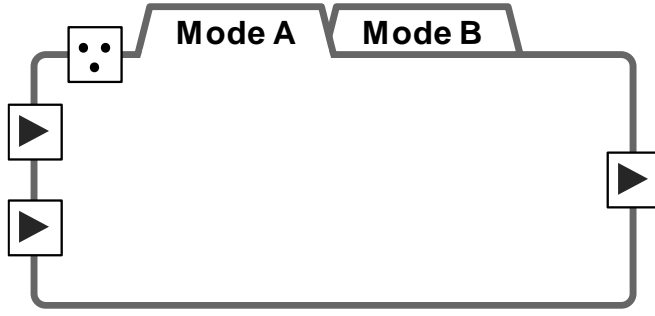


Figure 20. A multimode factory with two modes.

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. In the Splash program, the largest factory is called the top-level factory, and the internal factory is each called a subfactory.

In Splash, a factory may have multiple modes of operations. Such a factory is called a multimode factory. A multimode factory consists of as many alternative factories as the mode. Each alternative factory corresponds to a certain mode. Figure 20 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.

A set of mode in a multimode factory is denoted by  $M = \{m_0, m_1, \dots, m_{n-1}\}$ . The mode  $m_0$  is the mode of this factory when the program starts. It is called initial mode. Alternative factories that are mapped to each mode of the factory is called mode factories.

Figure 21 shows an example of mode factories. The localization factory in this example has three stream input ports for GPS signal, previous position of the ego vehicle, and current acceleration of the

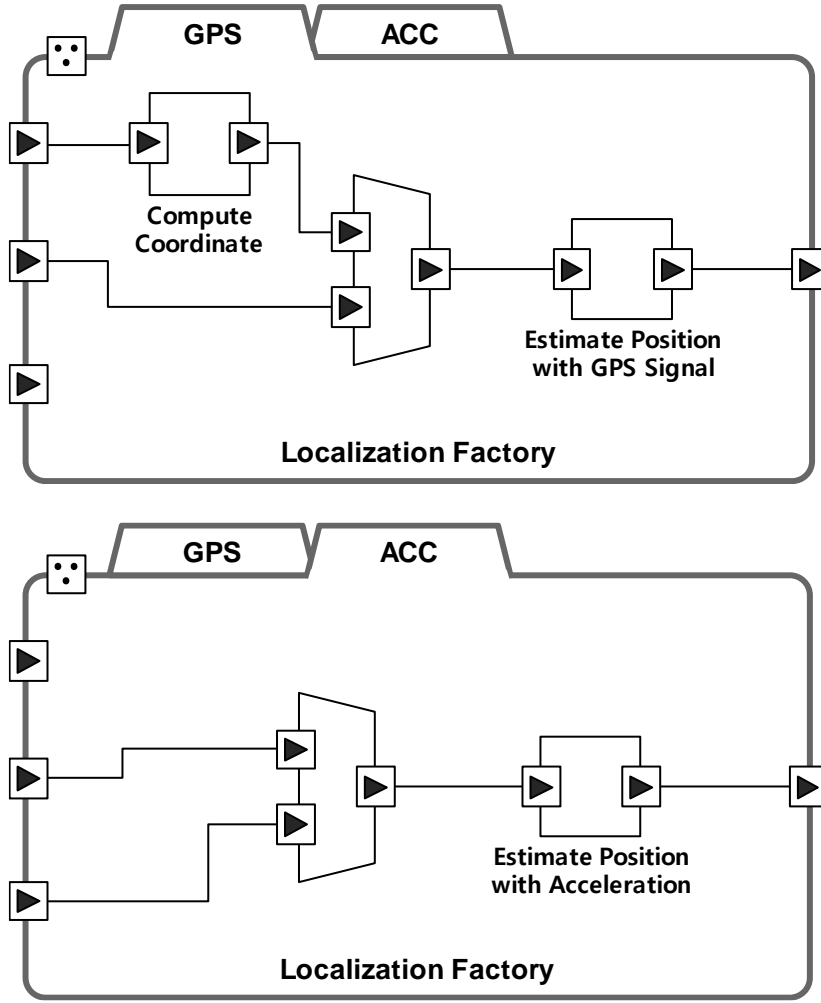


Figure 21. Example of mode factory.

ego vehicle. It also has a stream output port to produce current position of the ego vehicle. The localization factory has two modes: GPS and ACC. The mode factory of GPS and ACC are placed in the top and bottom of the figure, respectively.

The mode change of a multimode factory is triggered when the factory receives a mode change event. Let  $E$  be a set of all mode change events that can be received by the multimode factory. The mode change rule of the factory is defined using a mode change

Table 2. Example of Mode Change Table

Previous Mode	Mode Change Event	Next Mode	Output Remaining Internal Data Items
GPS	GPS_signal_found	GPS	-
	GPS_signal_lost	ACC	X
ACC	GPS_signal_found	GPS	O
	GPS_signal_lost	ACC	-

function  $\delta: M \times E \rightarrow M$ . For two modes  $m_i, m_j \in M$  and a mode change event  $e \in E$ ,  $\delta(m_i, e) = m_j$  if and only if the mode of the factory with mode  $m_i$  is changed to  $m_j$  when  $e$  occurs. A programmer can determine a mode change function  $\delta$  using a mode change table as shown in Table 1. For two modes  $m_i, m_j \in M$  and a mode change event  $e \in E$  that satisfy  $\delta(m_i, e) = m_j$ , the programmer should fill in  $m_i$ ,  $e$  and  $m_j$  in the previous mode, mode change event and next mode columns in the mode change table, respectively. In addition, the programmer should determine whether the factory should process and output the remaining internal data items that were being processed in the previous mode.

During the mode change, the factory's internal data items are classified into (1) queued data items and (2) in-processing data items. The queued data items are stored in the input queues of each component. The in-processing data items are data items that are currently being processed by sthreads and will be produced as outputs. Figure 22 shows queued data items and in-processing data

items of a multimode factory.

Splash runtime provides two types of synchronous mode change behavior depending on how the programmer decided whether to output remaining internal data items during the mode change. If the programmer decided to output internal data items, the mode change is performed as follows.

- (1) The incoming data items of the factory are blocked
- (2) Sthreads of the factory continue to process data items. If an output data item is generated during the execution of a sthread, the item is produced using the stream output port as usual.
- (3) When all input and output queues are empty and all sthreads become idle, the factory changes its mode to the next mode

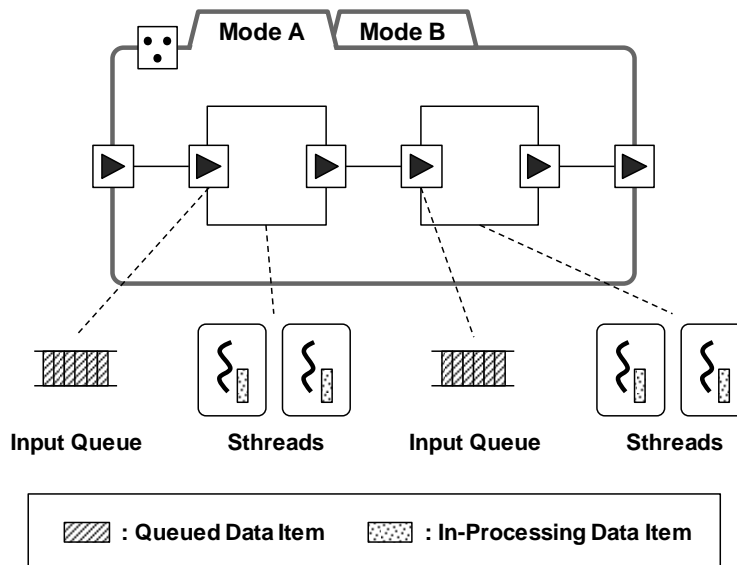


Figure 22. Internal data items of a multimode factory.

and resumes execution.

If the programmer chose not to output internal data items, the mode change is performed as follows.

- (1) The incoming data items of the factory are blocked
- (2) The sthreads are configured to no longer retrieve data items from input and output queues inside the factory.
- (3) Wait for all sthreads to finish their iteration. While doing so, all output data items generated during the execution of a sthread are dropped.
- (4) When the iteration of all sthreads is complete, the factory changes its mode to the next mode and resumes execution. Data items remaining in the input and output queues of inside the factory are discarded.

In both mode change operations Splash's mode change mechanism guarantees that an sthread does not terminate during its iteration. By doing so, the consistency of global data used by sthreads is preserved.

## **5.6 Build Unit**

As an sthread is an abstract entity, it needs to be mapped to a process and a thread of an underlying operating system during the system

implementation process. Since the process and thread are execution entities, they must eventually run on a specific core of a specific processor on a specific computing node. To facilitate this process, Splash offers an allocation entity called a build unit.

A build unit is an entity that allocates a set of sthreads to a process. Each build unit is mapped to one or more components in the Splash program. The following rules should be observed in the process of mapping components and build units.

- (1) All atomic components must be mapped to a build unit.
- (2) A factory can be mapped to a build unit. However, when a factory is mapped to a build unit, all components belonging to that factory must also be mapped to the same build unit.

Splash automatically detects and generates a syntax error if the programmer does not follow the abovementioned rules. Figure 23 shows an example of component–build unit mapping.

Splash allocates sthreads to processes and threads based on the component–build unit mapping. Splash assigns a single process to all sthreads of components mapped to the same build unit. On the other hand, Splash attempts to reduce context switches and communication overhead by reducing the maximum number of threads while allocating sthreads to threads. The details of the build unit–based allocation are explained in Chapter 7.

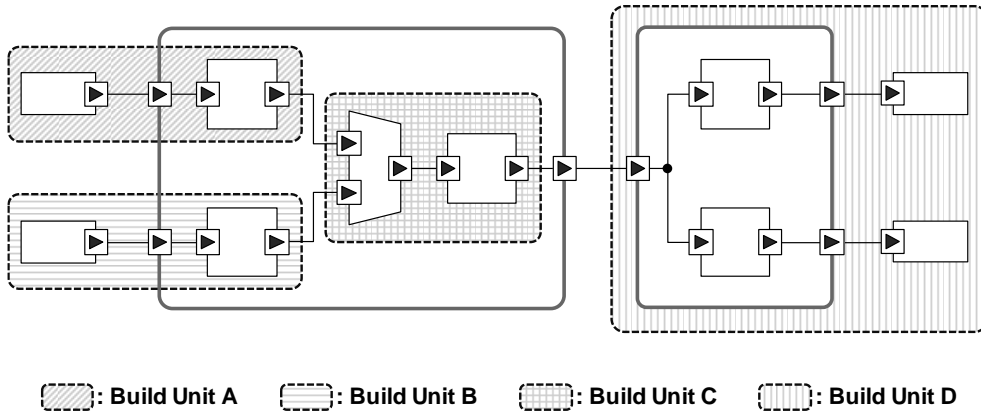


Figure 23. Example of component–build unit mapping.

## 5.7 Exception Handling

Splash provides exception handling to automatically handle exceptions. The Splash runtime monitors the occurrence of exceptions specified by programmers at runtime. If an exception is detected, the Splash runtime creates an exception object corresponding to the exception and invokes an exception handler with the exception object.

Splash supports three types of exceptions. Figure 24 shows the hierarchy of exception class in the UML diagram format. The first type of exception is a timing violation exception that is caused by a violation of end-to-end timing constraints. Splash specifically supports the handling of freshness constraint violation. The Splash runtime checks for violations of freshness constraints at the time each data item is inserted or removed from the input and output queues. If a freshness constraint is violated, the Splash runtime calls



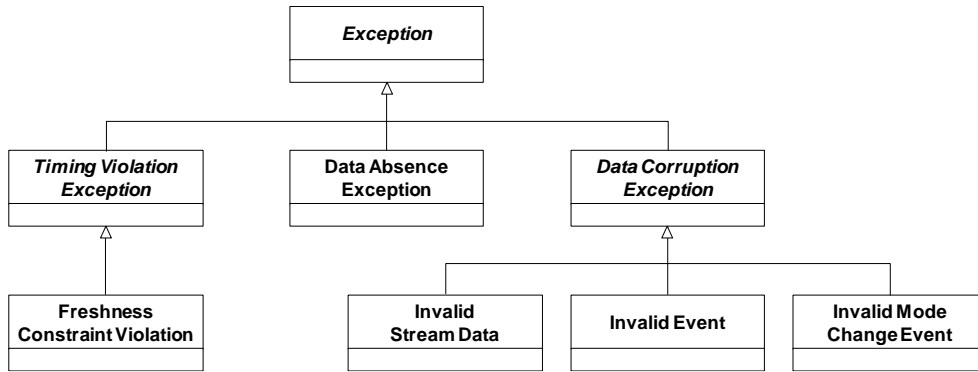


Figure 24. Hierarchy of exception class.

a default exception handler that removes the data item immediately.

The second type of exception is a data absence exception that occurs when a data item is not arrived when needed. The exception handler for the data absence exception should be written by the programmer.

The last type of exception is a data corruption exception that is caused by a stream data item, event or mode change event with unacceptable value. If a programmer specifies a range of values that are allowed for a stream data item, event or mode change event, the Splash runtime checks for an input outside of the range and throws an exception. The exception handler for the data corruption exception also should be written by the programmer.

## Chapter 6. Splash Runtime Mechanisms

This chapter describes the runtime mechanism that realizes language semantics of Splash. Section 6.1 explains a rate control mechanism for implementing rate control semantics. Then, Section 6.2 describes a sensor fusion mechanism that implements a fusion operator. Finally, Section 6.3 explains a mode change mechanism for a multimode factory.

### 6.1 Rate Control Mechanism

The runtime mechanism of a rate-controlled stream output port consists of an output queue and a rate controller as shown Figure 25. A sthread inside a processing component enqueues a data item into the output queue. Our runtime mechanism can effectively bound the size of the output queue via the freshness constraint of data items

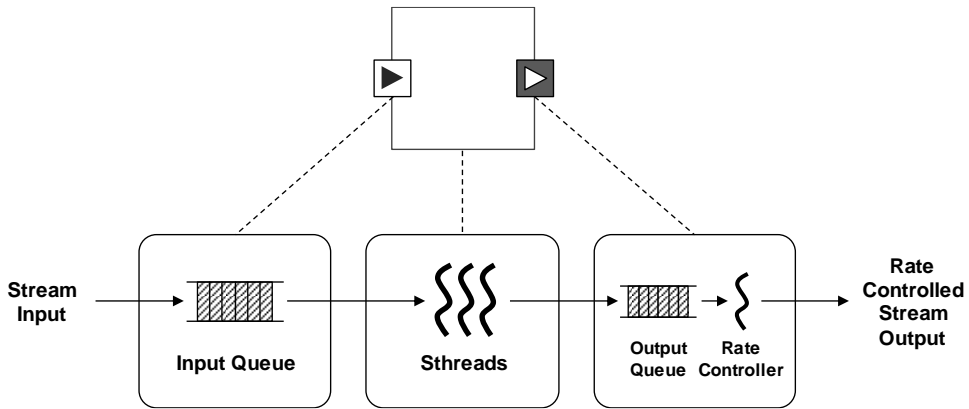


Figure 25. Runtime mechanism of a rate controller.

stored in the queue. The bound is computed as below.

$$s_{\max} = \lfloor r \times f \rfloor$$

where the  $s_{\max}$  is the output queue size and  $f$  is the freshness constraint of the data items.

If the output queue is full when a sthread attempts to insert a data item, the data item at the front of the output queue is first discarded and then the incoming data item is stored at the tail of the output queue.

A rate controller with a rate constraint  $r$  is invoked every  $1/r$  interval. Let  $d_{\text{last}}$  be the last sent data item and  $b(d)$  be the birthmark of the data item  $d$ . On each periodic invocation, the rate controller looks up the output queue from the head to find the first data item  $d_{\text{next}}$  whose birthmark is greater than  $b(d_{\text{last}})$ . If there is such  $d_{\text{next}}$ , it discards all the data items before  $d$  in the output queue and sends out  $d_{\text{next}}$ ; otherwise, it generates an extrapolation command. The extrapolation command is newly assigned a birthmark whose value is  $b(d_{\text{last}}) + 1/r$ .

## 6.2 Sensor Fusion Mechanism

The runtime mechanism of a fusion operator is shown in Figure 26. Each stream input port in the fusion operator has an input queue that

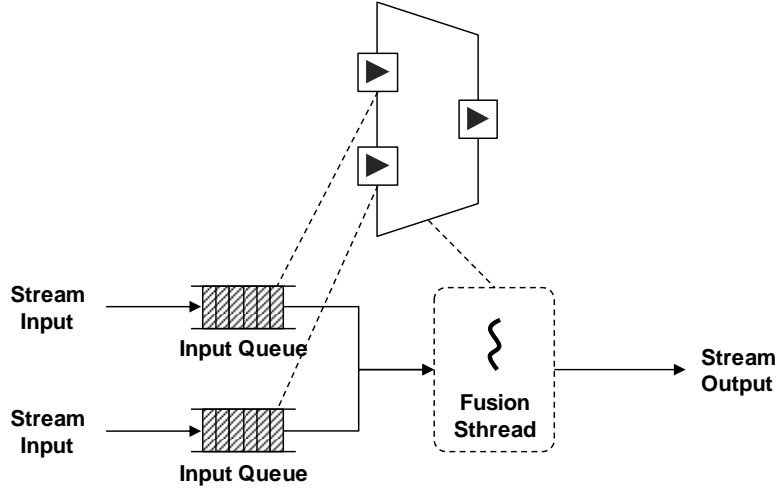


Figure 26. Runtime mechanism of a fusion operator.

stores data items in ascending order of their birthmark. A fusion sthread is invoked when a data item is inserted into one of the input queues to check whether there are input tuples that satisfy the fusion rule. If so, it retrieves the data items from each input queue and calls the corresponding fusion function. After the execution of the fusion function is completed, the fusion sthread checks again for the presence of an input tuple that satisfies the fusion rule. If there exists such a tuple, it retrieves the input data items again and calls the fusion function. If no satisfying fusion rule exists, the fusion sthread is blocked until the next data item arrives at one of the input queues.

Figure 27 shows the pseudocode of the **FINDVALIDINPUTTUPLE** algorithm used by the fusion sthread. The goal of the algorithm is to return a least input tuple that satisfies a given fusion rule if one exists. Its inputs are a fusion rule  $R = (M, O, \theta, c)$  and a set of sequences  $S = \{s_1, s_2, \dots, s_m\}$  of data items stored in each input queue of the fusion

---

**ALGORITHM 1. FINDVALIDINPUTTUPLE**

---

**Input:** A fusion rules  $R = (M, O, \theta, c)$   
A set of data item sequences  $S = \{s_1, s_2, \dots, s_m\}$

```
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[ $i$ ]  $\leftarrow 1$ 
5:   else
6:     index[ $i$ ]  $\leftarrow \text{NIL}$ 
7: while index[ $i$ ] = NIL for  $1 \leq i \leq m$ 
8:   if ISVALIDTUPLE( $R, S, \text{index}$ )
9:     return BUILDTUPLE( $S, \text{index}$ )
10:   $k \leftarrow \text{GETEARLISTINDEX}(S, \text{index})$ 
11:  if index[ $k$ ] <  $|s_k|$ 
12:    index[ $k$ ]  $\leftarrow \text{index}[k] + 1$ 
13:  else
14:    index[ $k$ ]  $\leftarrow \text{NIL}$ 
15: return ( $\perp, \perp, \dots, \perp$ )
```

---

Figure 27. Pseudocode of **FINDVALIDINPUTTUPLE** algorithm.

operator where  $s_i$  is a sequence of data items stored in the  $i$ th input queue sorted in ascending order according for their birthmark. The output of the algorithm is an input tuple  $(d_1, d_2, \dots, d_m)$ . If there is no input tuple satisfying the fusion rule  $R$ , the algorithm outputs a tuple of empty data items  $(\perp, \perp, \dots, \perp)$ .

The **FINDVALIDINPUTTUPLE** algorithm first initialize the array index[1 ...  $m$ ] which stores indices of sequences  $s_1, s_2, \dots, s_m$  in lines 1–6. For  $1 \leq i \leq m$ , if the port  $p_i$  is a mandatory or optional port and there is more than one data item in  $s_i$ , index[ $i$ ] is initialized to 1 so that  $s_i[\text{index}[i]]$  is a data item with the smallest birthmark in  $s_i$ . Otherwise, index[ $i$ ] is initialized to NIL.

The algorithm then iteratively searches for an input tuple that

satisfies the fusion rule  $R = (M, O, \theta, c)$  (lines 7–11). At the start of each iteration, it invokes the **ISVALIDTUPLE** function that checks whether an input tuple that satisfies  $R$  can be built from the data items pointed by the index array (line 8). If such input tuple can be built, the algorithm returns that input tuple by calling the **BUILDTUPLE** function (line 9). If such input tuple cannot be built, the algorithm increases an element in the index array that points to the oldest data item by one (lines 10–12). If this element is already pointing to the last data item in the sequence, we set it to NIL instead of increasing it by one (line 14). If no valid input tuple has been found until all elements in the index array become NIL, the algorithm returns a tuple of empty data items (line 15).

The **FINDVALIDINPUTTUPLE** algorithm always return the least input tuple in  $V(R)$  if there exists at least one input tuple that satisfies  $R$ .

**THEOREM 2:** For a fusion rule  $R = (M, O, \theta, c)$ , let us denote a set of input tuples that satisfy  $R$  as  $V(R)$ . The **FINDVALIDINPUTTUPLE** algorithm always return the least input tuple in  $V(R)$  if  $V(R) \neq \emptyset$ .

**PROOF:** Let us denote the least input tuple in  $V(R)$  that the **FINDVALIDINPUTTUPLE** algorithm should return as  $(d'_1, d'_2, \dots, d'_m)$ . The following three sets are defined from a set of data items included in the any one of sequences  $s_1, s_2, \dots, s_m$ .

$$I_{\text{older}} = \{d_i: b(d_i) < b(d'_i) \ (d'_i \neq \perp)\}$$

$$I_{\text{answer}} = \{d_i: d_i = d'_i \ (d'_i \neq \perp)\}$$

$$I_{\text{newer}} = \{d_i: b(d_i) > b(d'_i) \ (d'_i \neq \perp)\}$$

Figure 28 shows an example of  $I_{\text{older}}$ ,  $I_{\text{answer}}$  and  $I_{\text{newer}}$  for a fusion operator with three mandatory input ports.

Now we show that the following loop invariant holds for the while loop in lines 7–14.

- Let  $(d''_1, d''_2, \dots, d''_m)$  be an input tuple pointed by the index array at the start of each iteration. If  $d'_i \neq \perp$ ,  $d''_i \in I_{\text{older}}$  or  $d''_i \in I_{\text{answer}}$  for  $1 \leq i \leq m$ .

Initialization: For  $1 \leq i \leq m$ , the  $\text{index}[i]$  is initialized to 1 if  $d'_i \neq \perp$  because the port  $p_i$  is a mandatory or optional port and there is at

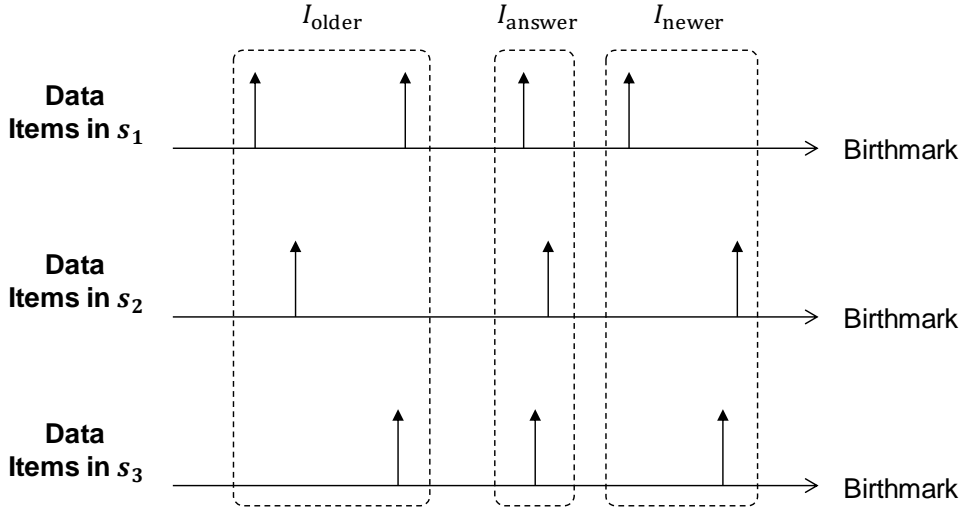


Figure 28. Example of  $I_{\text{older}}$ ,  $I_{\text{answer}}$ ,  $I_{\text{newer}}$ .

least one data item in  $s_i$ . Since the  $s_i[\text{index}[i]]$  is a data item with the smallest birthmark in  $s_i$ ,  $b(d_i'') \leq b(d_i')$  holds. Therefore,  $b(d_i'') \in I_{\text{older}}$  or  $b(d_i'') \in I_{\text{answer}}$ , and thus the loop invariant holds prior to the first iteration of the loop.

Maintenance: Let us first suppose that the loop invariant holds for the input tuple  $(d_1'', d_2'', \dots, d_m'')$  at the start of the iteration. If this iteration is not terminated in line 9,  $\text{index}[k]$  is incremented by 1 on line 12 or is set to NIL in line 14. Let  $d_k^*$  be the new data item pointed by the  $\text{index}[k]$  if it is not set to NIL and  $\perp$  if it is set to NIL.

In order to prove that the loop invariant is maintained, we must show that  $d_k^* \in I_{\text{older}}$  or  $d_k^* \in I_{\text{answer}}$ . Suppose for the purpose of contradiction that  $d_k^* \in I_{\text{newer}} \cup \{\perp\}$ . Then,  $d_k'' \in I_{\text{answer}}$ , and thus  $d_k'' = d_k'$ . Since the loop is not terminated in line 9, there exists at least one data item  $d_j''$  in  $(d_1'', d_2'', \dots, d_m'')$  where  $d_j'' \in I_{\text{older}}$  ( $j \neq k$ ). By the definition of  $I_{\text{older}}$ ,  $b(d_j'') < b(d_j')$  holds. Also,  $b(d_k'') < b(d_j'')$  because  $d_k''$  is the data item with the smallest birthmark in for the input tuple  $(d_1'', d_2'', \dots, d_m'')$ . The relationship between the data items  $d_k'$ ,  $d_k''$ ,  $d_j'$  and  $d_j''$  is illustrated in Figure 29.

Now we define an input tuple  $(d_1', d_2', \dots, d_j'', \dots, d_m')$  with  $d_j'$  replaced by  $d_j''$  in  $(d_1', d_2', \dots, d_m')$ . Since  $(d_1', d_2', \dots, d_m')$  satisfies the correlation constraint,  $(d_1', d_2', \dots, d_j'', \dots, d_m')$  also satisfies the correlation constraint as shown in Figure 29. Therefore,  $(d_1', d_2', \dots, d_j'', \dots, d_m') \in V$ . However,  $(d_1', d_2', \dots, d_m') \leq (d_1', d_2', \dots, d_j'', \dots, d_m')$  does not hold, which is a contradiction since  $(d_1', d_2', \dots, d_m')$  is the least



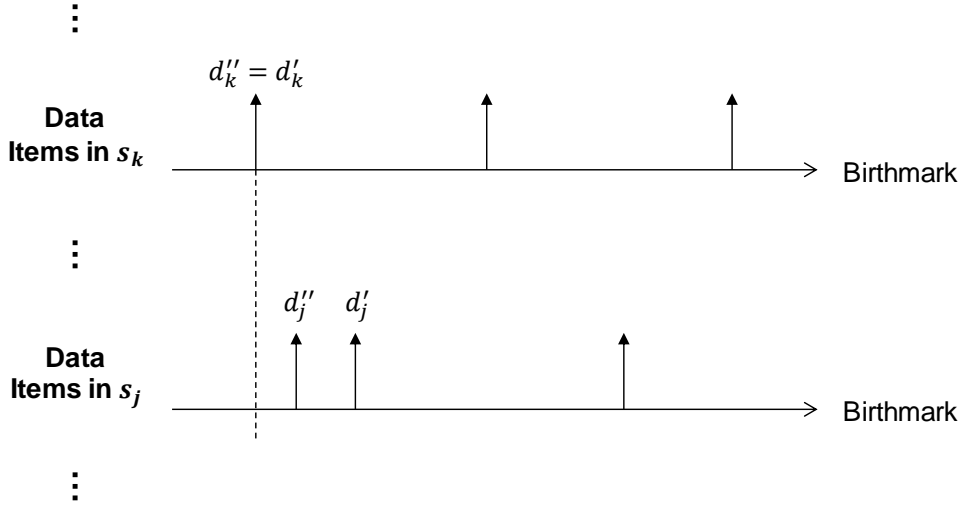


Figure 29. The relationship between  $d'_k$ ,  $d''_k$ ,  $d'_j$  and  $d''_j$ .

input tuple in  $V(R)$ . Therefore,  $d_k^* \in I_{\text{older}}$  or  $d_k^* \in I_{\text{answer}}$ .

Termination: If there exists a data item  $d_i'' \in I_{\text{older}}$  for  $1 \leq i \leq m$ , the while loop cannot be terminated. In a finite number of iterations, the input tuple  $(d_1'', d_2'', \dots, d_m'')$  pointed by the index array satisfies the following condition.

$$d_i' \neq \perp \rightarrow d_i'' \in I_{\text{answer}} \quad (1 \leq i \leq m)$$

Then, the algorithm finds the least input tuple in  $V(R)$  and returns it in line 9. Therefore, **THEOREM 2** holds.  $\square$

We now analyze the runtime complexity of the **FINDVALIDINPUTTUPLE** algorithm. Let  $m$  be the number of input ports of the fusion operator and  $l$  be the maximum input queue size.

The while loop of the **FINDVALIDINPUTTUPLE** algorithm is repeated at most  $m \cdot l$  times. If we implement the **ISVALIDTUPLE**, **BUILDTUPLE** and **GETEARLIESTINDEX** functions with a linked list, it takes  $O(m)$  to run lines 7–9 and  $O(1)$  to run lines 10 and 11. Therefore, the time complexity of the **FINDVALIDINPUTTUPLE** algorithm is  $O(m^2 \cdot l)$ .

### 6.3 Mode Change Mechanisms

The Splash runtime supports two types of mode change mechanisms depending on whether internal data items are produced during mode change. It provides the following five operations to implement these mechanisms.

- (1) **BLOCKINPUTDATAITEMS**: The incoming data items into the factory is blocked and stored in a queue/
- (2) **DISABLESTHREADREAD**: The sthreads are configured to no longer read queued data items from input and output queues of the factory. Each sthread becomes idle after finishing its current iteration.
- (3) **AREQUEUESEEMPTY**: Check if all input and output queues inside the factory is empty.
- (4) **ARESTHREADSIDLE**: Check if all sthreads are idle.
- (5) **CHANGEANDRESUME**: Deallocate all input and output queues and terminate all sthreads of the previous mode. Then, allocate and initialize input and output queues and create

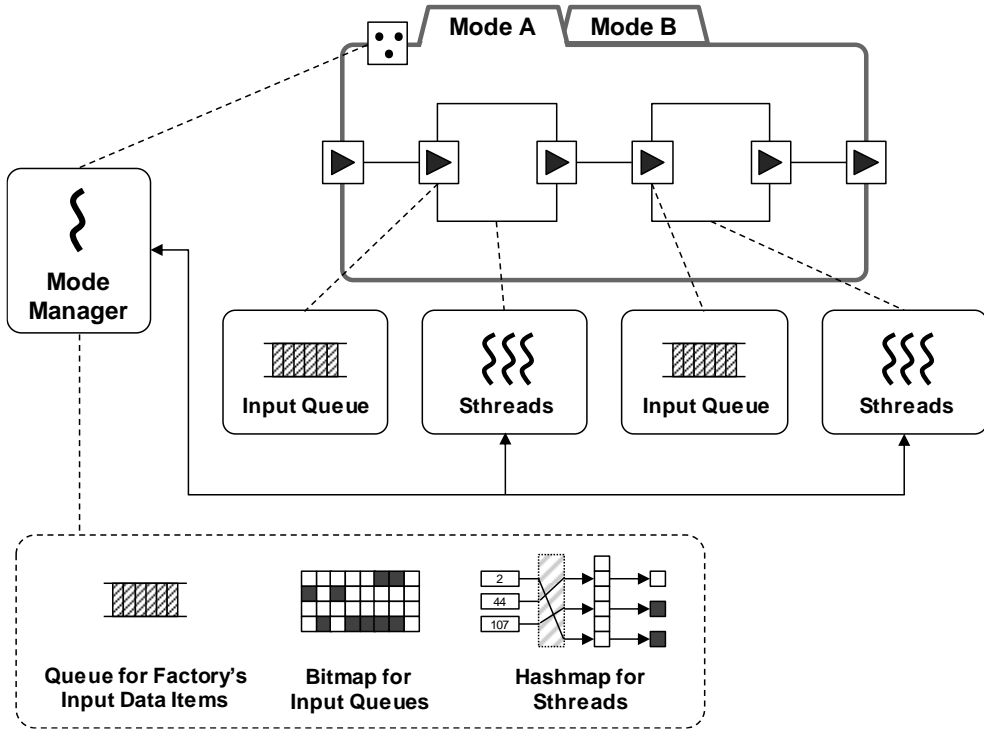


Figure 30. Runtime mechanism of mode change.

stthreads for the next mode. Finally, put the data items stored in the **BLOCKINPUTDATAITEMS** operation to newly created input queues, and resume execution.

Figure 30 shows the runtime mechanism of mode change. The Splash runtime creates a new sthread named mode manager to support mode change operation. The mode manager has three data structures. First, it has a queue that stores blocked data items while running the **BLOCKINPUTDATAITEMS** operation. This queue stores data items in birthmark order to provide in-order delivery semantics. Second, it has a bitmap that checks whether each input queue inside the factory is empty or not. This data structure is used to implement

---

**ALGORITHM 2.    CHANGEMode**

---

**Input:** A flag that indicates whether to output the internal data items  $f$

```
CHANGEMode( $f$ )
1:  BLOCKINPUTDATAITEMS()
2:  if  $f$  = true
3:    Wait until AREQUEUESEEMPTY() and ARETHREADSIDLE() = true
4:  else
5:    DISABLESTHREADREAD()
6:    Wait until ARETHREADSIDLE()
7:  CHANGEANDRESUME()
```

---

Figure 31. Pseudocode of **CHANGEMode** algorithm.

the **AREQUEUESEEMPTY** operation. Finally, the mode manager has a hashmap that checks whether each sthread inside the factory is idle or not. This data structure is used to implement the **ARETHREADSIDLE** operation.

Figure 31 shows the pseudocode of **CHANGEMode** algorithm. This algorithm is invoked by the mode manager when the mode change event arrives at the mode change port of the factory. The algorithm first blocks input data items of the factory by calling **BLOCKINPUTDATAITEMS** operation in line 1. If the programmer decided to output the remaining internal data items, the algorithm waits until there are no more internal data items inside the factory (line 3). If the programmer decided not to output the internal data items, the algorithm set all sthreads to no longer read queued data items (line 5). It then waits until there are no more in-process data items inside the factory (line 6). When all the work is done, the mode of the factory is changed to the next mode by calling the **CHANGEANDRESUME** operation.

## Chapter 7. Code Generation and Runtime System

This chapter describes the code generation and runtime system for executing Splash program. Section 7.1 explains how sthreads are allocated to processes and threads, and how the communication between sthreads are implemented. Section 7.2 introduces templates for the code generation. Section 7.3 explains the runtime system that runs the Splash program.

### 7.1 Build Unit-based Allocation

As an sthread is an abstract entity, it needs to be mapped to a process and a thread of an underlying operating system during the system implementation process. This process can be divided into two steps: allocating sthreads to processes and allocating sthreads to threads. Splash first allocates the sthreads of all components mapped to each build unit to the same process. It then allocates the sthreads to threads according to the following rules.

- (1) **Dedicated sthread of a processing component:** Splash allocates a dedicate sthread to the same thread as the sthread that sends data items to it. This allocation policy reduces the context switch overhead and communication overhead by reducing the number of threads running on the system. However, in the following four cases, the dedicated sthread is

allocated to a different thread as the sthread that sends data items to it.

- When an sthread that sends data items to a dedicated sthread is mapped to a different build unit, they are allocated to different threads because they are mapped to different processes.
- When a sthread that sends data items to a dedicated sthread sends data items to more than one stream output port, these sthreads are allocated to different threads for concurrent execution.
- Similarly, when a sthread that sends data items to a dedicated sthread communicates with two or more sthreads through a channel with multiple fan-outs, these sthreads are allocated to different threads.
- If the sthread sending data items to the dedicated sthread is a rate controller, they are allocated to different threads in order for the rate controller to work correctly.

**(2) Internal sthread of a processing component:** Splash assigns each internal sthread to a separate thread. This is because the internal sthread is created by the programmer with the intention of multithreading.

**(3) Fusion sthread of a fusion operator:** Splash allocates each fusion sthread to a separate thread to ensure that the fusion operator to work properly.

Splash provides three different types of implementations for communications between sthreads. First, communication between sthreads that are allocated to different processes is implemented using inter-process communication (IPC) based on DDS (data distribution service) [15,75]. Second, communication between sthreads that are allocated to the same process but different threads is done through a queue located in the global address space shared by both threads. Finally, Communication between sthreads assigned to the same process and the same thread is implemented using a simple function call.

## **7.2 Code Generation Template**

The Splash programming language is a coordination language that defines the interaction between components. A host language such as C++ which is used to define subprograms inside a component. Splash. Splash provides a schematic editor to write a coordination program in Splash programming language. After the programmers have completed writing the coordination program, the schematic editor produces a JSON file to be used as an input to the code generator. The JSON file contains the information about the factory and internal language constructs such as processing components, fusion operators, channels, stream input/output ports and build units.

The code generator takes the generated JSON file of the schematic editor as an input and produces template source code files

---

```
1:  SourceComponent sc;
2:  ProcessingComponent pc;
3:
4:  void sc_user_function();
5:  void pc_user_function(void *msg_ptr);
6:
7:  int main(void) {
8:      StreamOutputPort<fname::data1> sc_sout;
9:      StreamInputPort<fname::data1> pc_sin;
10:     StreamOutputPort<fname::data2> pc_sout;
11:
12:     sc.initialize("sname", 200);
13:     pc.initialize("pname");
14:     sc_sout.initialize();
15:     pc_sin.initialize();
16:     pc_sout.initialize(15);
17:
18:     sc.registerUserFunction(sc_user_function);
19:     pc.registerUserFunction(pc_user_function);
20:
21:     sc_sout.attach(&sc, INTRA_THREAD);
22:     pc_sin.attach(&pc, INTRA_THREAD);
23:     pc_sout.attach(&pc, INTER_PROCESS, "topic1");
24:
25:     pc.run();
26:     sc.run();
27: }
28:
29: void sc_user_function() {
30:     fname::data1 output_data;
31:     // Put user logic here
32:     sc.write(&output_data);
33: }
34:
35: void pc_user_function(void *msg_ptr) {
36:     fname::data0 input_data =
37:         *static_cast<fname::data0*>(msg_ptr);
38:     fname::data1 output_data;
39:     // Put user logic here
40:     pc.write(&output_data);
41: }
```

---

Figure 32. Example of template source code.



written in C++. A template source code file is created for each build unit to create, to initialize and to initiate components that are mapped to the build unit. Figure 32 shows a template code for a source component and a processing component mapped to a single build unit. As shown in the figure, the template code is structured in three segments: declaration (lines 1–10), configuration (lines 12–23) and execution (lines 25–26).

In declaration, a source component object (line 1), a processing component object (line 2), user functions for the components (lines 4–5), and internal stream input/output port objects (lines 8–10) are declared. In configuration, these objects are initialized in order (lines 12–16). Then the user functions are registered to the components (lines 18–19), and the stream input/output port objects are attached to the source component and the processing component objects (lines 21–23). In execution, it waits for a data item to come in on the input port (lines 25–26). When a data item arrives, the user function is called (lines 29–41). Programmers should fill in the user logic inside the user function (lines 31 and 39)

### **7.3 Splash Runtime System**

The Splash runtime consists of two layers of software as shown in Figure 33. 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

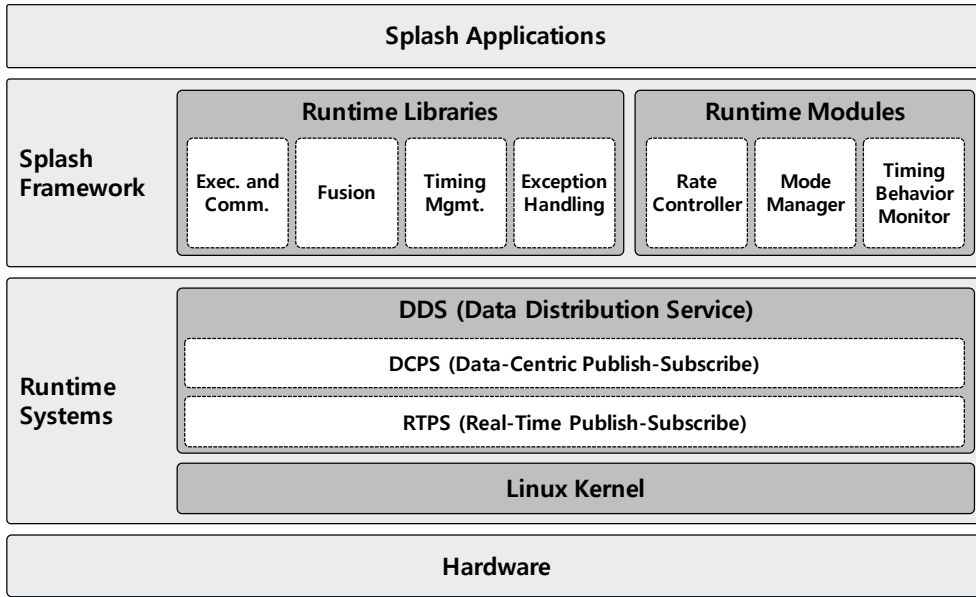


Figure 33. Splash runtime architecture.

Splash framework as shown in Figure 32. The runtime libraries are divided into five types according to their functions: (1) core execution and communication, (2) fusion, (3) timing management and (4) exception handling. The Splash framework also comes with three runtime modules: (1) the rate controller, (2) the mode manger and (2) the timing behavior monitor.

At the bottom layer lies a runtime system based on DDS (data distribution services) and Linux kernel. DDS is a well-known specification for real-time publish-subscribe communication. We chose OpenSplice DDS because it is open source and implements the specification efficiently [76].

## Chapter 8. Experimental Evaluation

In this chapter, we validate Splash by implementing the LKAS with the proposed framework and measuring its performance values. Section 8.1 explain overall application logic of the LKAS along with its timing constraints annotation and components-to-build unit mapping. Section 8.2 describes the experimental environment. Section 8.3 to 8.6 presents the experimental results for rate control, sensor fusion, mode change, and build unit-base allocation, respectively.

### 8.1 LKAS Program

We have designed LKAS based on the algorithm given in [77] with Splash. This application automatically adjusts the steering angle to keep the ego vehicle inside the detected lane. Figure 35 shows the

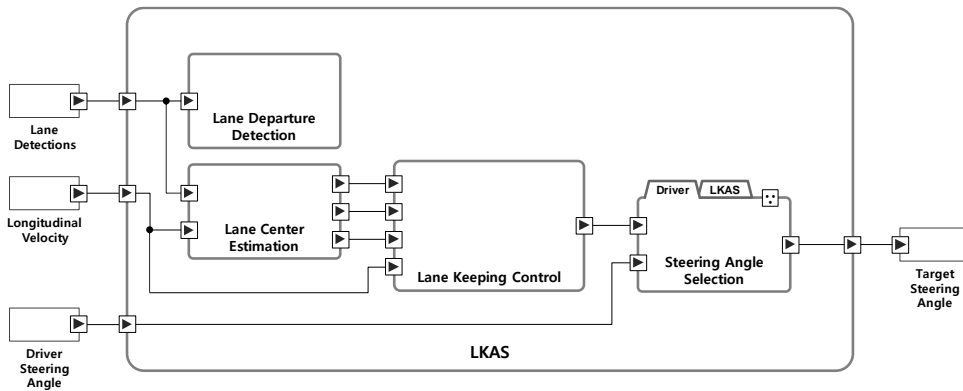


Figure 34. LKAS factory.

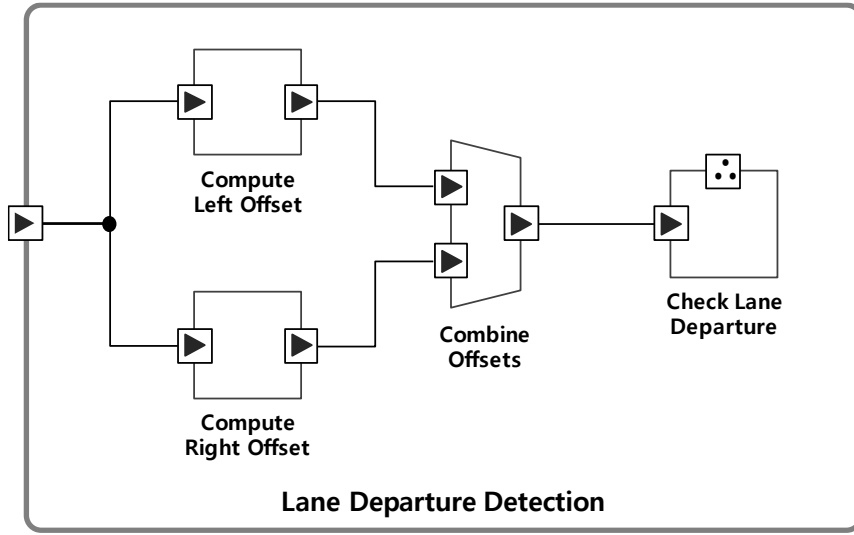


Figure 35. Lane departure detection factory.

top-level factory of the application. Its inputs include lane detections from the lane sensor, the longitudinal velocity of the ego vehicle, and the driver steering angle. Its output is the target steering angle of the vehicle. The top-level factory consists of four sub-factories: (1) lane departure detection, (2) lane center estimation, (3) lane keeping control, (4) steering angle selection.

Figure 35 is the lane departure detection factory. It checks if the ego vehicle is too close to the lane boundaries by computing the offset distance of the ego vehicle from both the left and right lane boundary. It then merges both offsets using a fusion operator and checks if any offset is lower than the predefined threshold. If any offset is lower than the threshold, it generates a mode change event.

Figure 36 is the lane center estimation factory. It computes the curvature of the lane, lateral deviation between the ego vehicle and center of the lane, and heading angle of the ego vehicle. To do

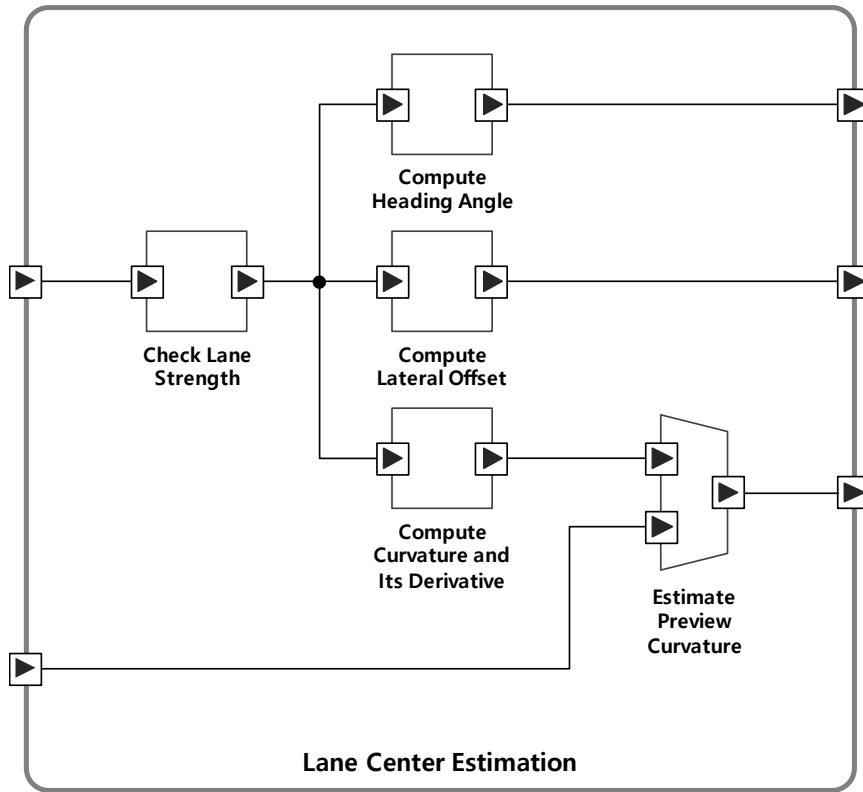


Figure 36. Lane center estimation factory.

so, it first checks whether left and right boundaries of the detected lane is clear enough to be used for estimation. It then computes curvature, lateral deviation and heading angle based on the selected boundaries. Finally, it estimates the curvature of the forward lane to be driven over the next three seconds using the curvature computed through the current detected lane.

Figure 37 is the lane keeping control factory. It generated assisted steering angle of the ego vehicle using the outputs of the lane center estimation factory and longitudinal velocity of the vehicle. It first uses a fusion operator to update states that will be sampled by vehicle controller. It then uses a controller which

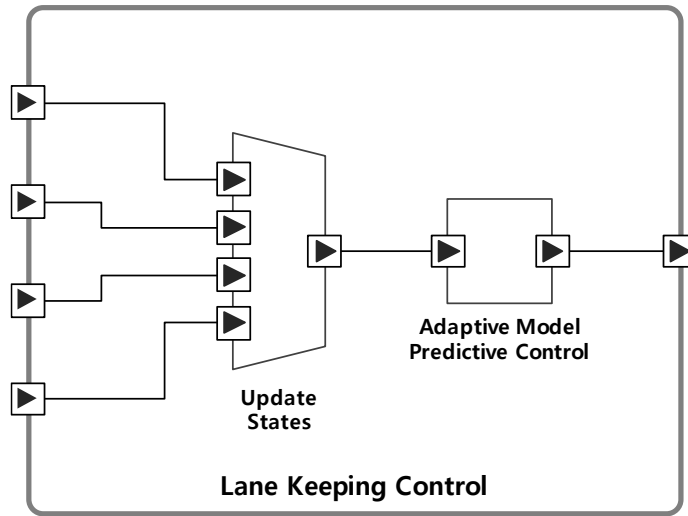


Figure 37. Lane keeping control factory

periodically invokes adaptive model predictive control logic to compute the assisted steering angle.

Figure 38 is the steering angle selection factory. It is a multimode factory which has two modes: Driver mode and LKAS mode. It takes both driver steering angle and assisted steering angle

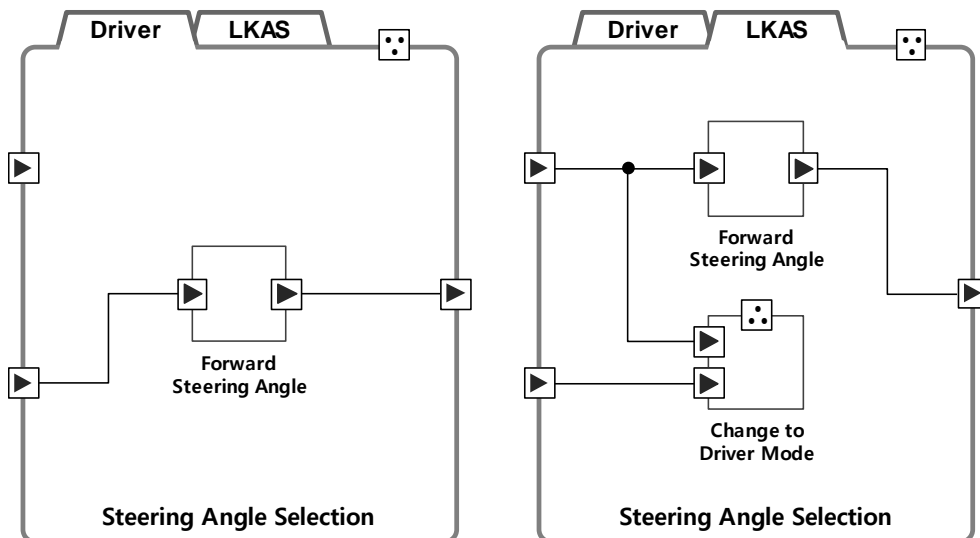


Figure 38. Steering angle selection factory.

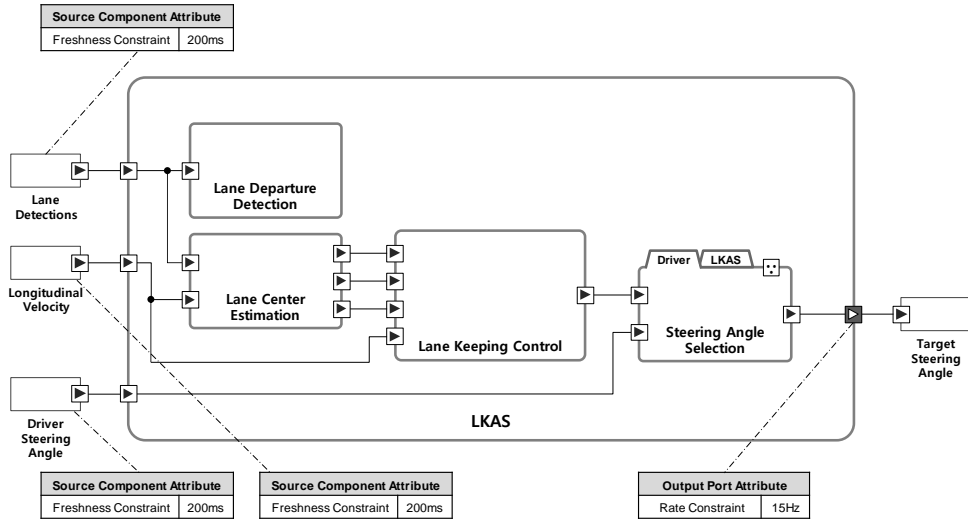


Figure 39. Timing constraints of LKAS factory

as inputs. It outputs driver steering angle for Driver mode and assisted steering angle for LKAS mode. Mode change from Driver mode to LKAS mode is triggered by a mode change event sent from the lane departure detection factory. On the other hand, mode change from LKAS mode to from Driver mode is triggered by a mode change event from change to driver mode processing component inside the steering angle selection factory. This processing component compared driver steering angle and assisted steering angle, and generates a mode change event only if it is safe for the driver to take control.

The end-to-end timing constraints of the LKAS factory is annotated as shown in Figure 39. First, we set we set freshness constraints to the same value of 200ms for the three source components. Second, we annotate a rate constraint of 15Hz for the stream output port of the LKAS factory. Finally, we set correlations

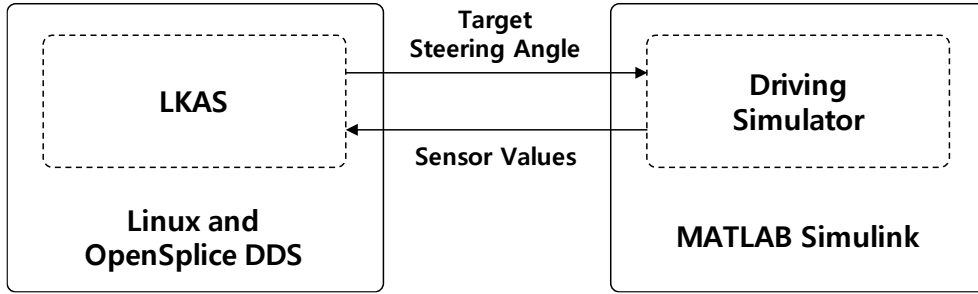


Figure 40. Software components of the platform.

constraint of each fusion operator to 10ms.

## 8.2 Experimental Environment

We simulated a driving environment with the Simulink to validate our

Table 3. Experimental Environment

Categories			Descriptions
LKAS	HW	CPU	Intel Core i7–3770 3.40 GHz
		Memory	8 GB
		Storage	Hitachi HDD 1 TB
	SW	OS	Linux version 4.15
		Framework	Vortex OpenSplice version 6.7
Driving Simulator	HW	CPU	Intel Core i7–7700 3.60 GHz
		Memory	8 GB
		Storage	SanDisk X400 SSD 128 GB
	SW	OS	Windows version 10
		Framework	MATLAB Simulink version 10.0



LKAS implementation [11]. The software organization is shown in Figure 40. We essentially created a closed loop simulator on top of two machines, one running the Splash implementation of LKAS and the other executing a driving simulator. The LKAS receives sensor values from the simulator and outputs a target steering wheel angle. The simulator in turn receives the steering wheel angle as its input. Table 3 shows the detailed hardware and software configuration of our target system.

### **8.3 Evaluating Rate Control**

To validate the effective of Splash’s rate controller, we analyzed five different metrics of the LKAS. First, we measured the output jitter of the LKAS to evaluate the controller’s rate manipulability. Second, we measured the end-to-end latency of LKAS to portray the low overhead of the rate controller. Third, we measured the change in the number of data items in the output queue of a rate controller to check the controller’s ability to bound the number of data items in the queue. Finally, we measured lateral deviation and heading angle of the ego vehicle to validate that the performance of the LKAS is improved by the rate controller.

In order to demonstrate the effectiveness of our approach, we attached rate controllers running with 10Hz rate constraint to the stream output ports of the LKAS factory as marked in Figure 39. We then measured the output jitter with and without the rate controllers,

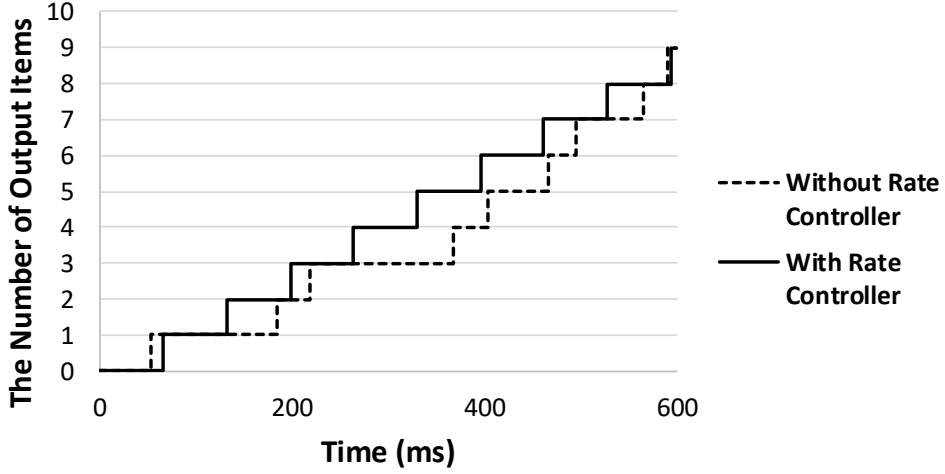


Figure 41. Comparison of the number of the accumulated output items.

respectively. We define the output jitter as discussed in [78].

$$J = \sqrt{\frac{\sum_{j=1}^h (E(O) - o_j)^2}{h}}$$

is a root mean square error where for a given  $h$  measurements,  $O$  is a set of  $h$  consecutive inter-output times of the LKAS,  $o_j \in O$  and  $E(O)$  is a mean of  $O$ .

Using the above definition as a metric, we found that the output jitter with our rate controllers was only 1.66 milliseconds, while the output jitter without rate controllers was 30.61ms milliseconds.

To better illustrate the effectiveness of the rate controller, we have plotted the accumulated number of data items that are output from the LKAS in Figure 41. As expected, the LKAS with rate controllers produced output data item every 66.67ms, while the

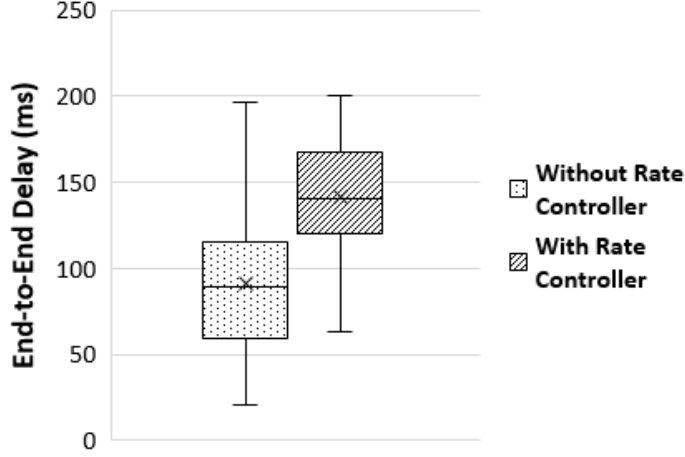


Figure 42. Comparison of the end-to-end latency.

LKAS without rate controllers produced the output irregularly.

In our second experiment, we measured the end-to-end latency of the LKAS for each data item, in order to display its minimal overhead. We defined the end-to-end latency as the time it takes for an input data to reach from the source component to a sink component. Figure 42 shows the box plot of the results. The average end-to-end latency was increased from 90.8ms to 141.7ms. The increase in the average end-to-end latency was caused by the delay in the output queue.

In the third experiment, we measured the number of data items in the output queue of a rate controller. Figure 43 plots the results. The result confirmed that the controller successfully bounded the number of items in the queue to  $\lceil r \times f \rceil$  as discussed in Section 6.1.

In the last experiment, we measured the lateral deviation and heading angle of the ego vehicle while running the LKAS. The experimental results are shown in Figure 44 and Figure 45. The

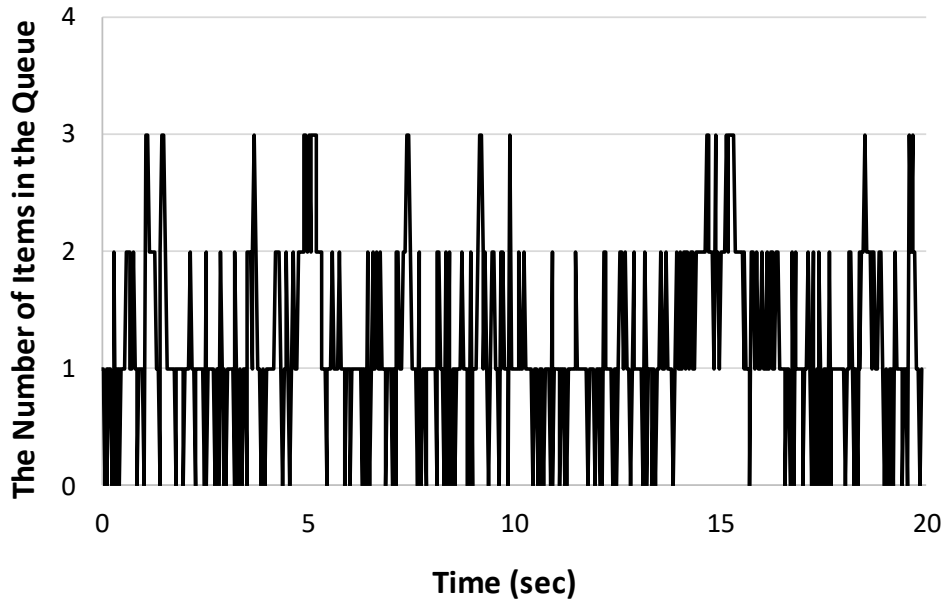


Figure 43. The number of data items in the output queue.

average lateral deviation without the rate controller was 0.180 meters, whereas it was only 0.016 meters with the rate controller. Also, the average heading angle was 0.043 rad without the rate controller, while it was 0.008 rad with the rate controller. Based on the results of these two experiments, it can be concluded that the performance of the LKAS is dramatically improved with a rate

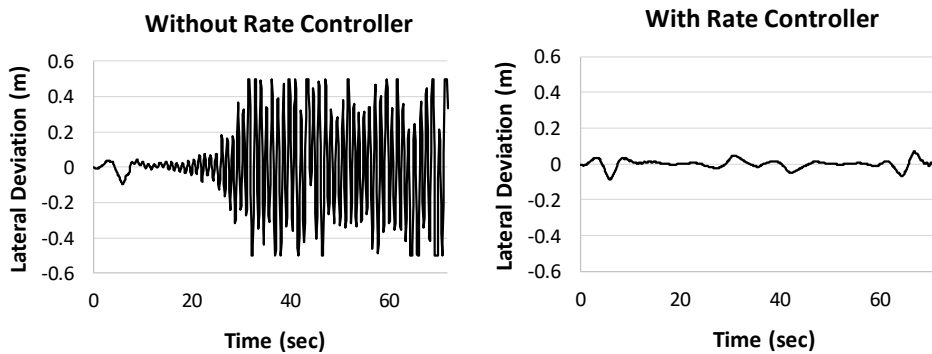


Figure 44. Comparison of the lateral deviation of the ego vehicle.

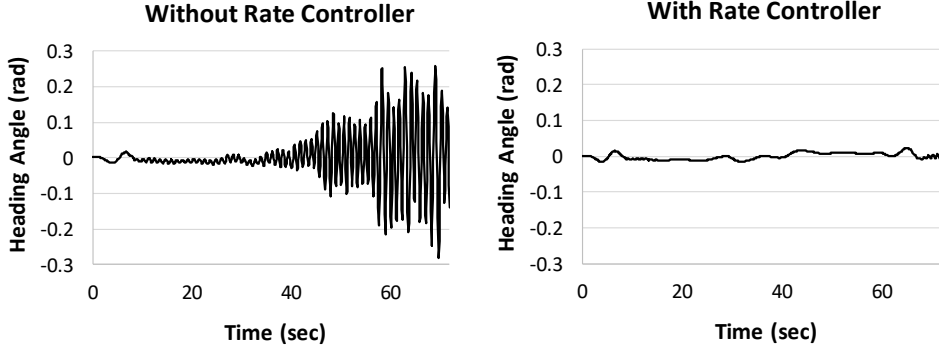


Figure 45. Comparison of the heading angle of the ego vehicle.

controller.

## 8.4 Evaluating Sensor Fusion

To validate the fusion operator, we selected two metrics. First, we measured the maximum birthmark differences between data items of an input tuple selected by the fusion operator. Second, we measured the average runtime overhead of the fusion operator incurred by running the **FINDVALIDINPUTTUPLE** algorithm.

In our first experiment, we ran the LKAS program for 30 seconds and measured the maximum birthmark differences of input tuples selected by the `estimate_preview_curvature` fusion operator inside the `lane_center_estimation_factory`. Figure 46 shows the results. As shown in the figure, the maximum birthmark differences between data items of input tuples were always less than the fusion operator’s correlation constraint, 10 milliseconds. This result clearly shows that our fusion operator effectively satisfied the annotated

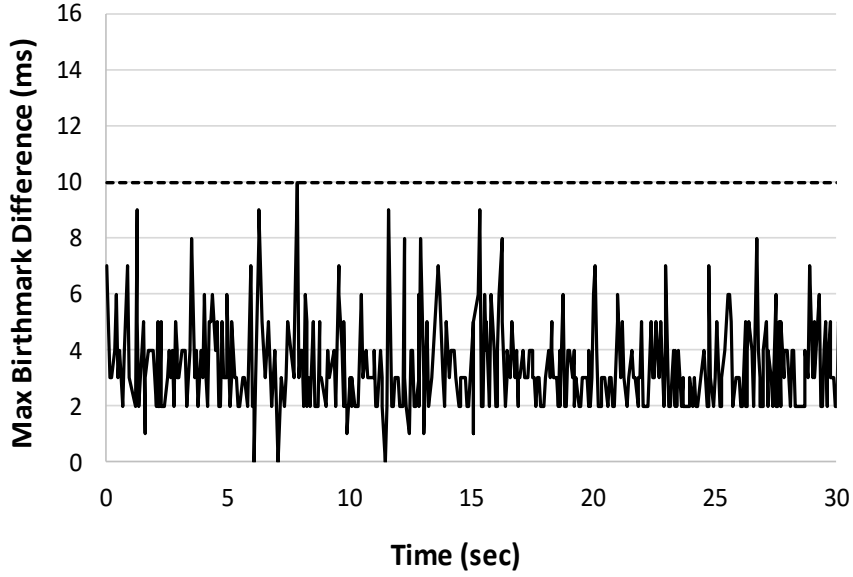


Figure 46. Maximum birthmark difference of input tuples chosen by the fusion operator.

correlation constraint.

In our second experiment, we measured the average running time of the **FINDVALIDINPUTTUPLE** algorithm to evaluate the overhead incurred by the fusion operator. As expected, the average running time of the algorithm was only 7 microseconds.

## 8.5 Evaluating Mode Change

In order to validate the mode change mechanism of Splash, we measured driving steering angle, assisted steering angle and final output steering angle of the LKAS factory. We also measured the average time it took for mode change.

We first ran the LKAS program for 30 seconds and measured

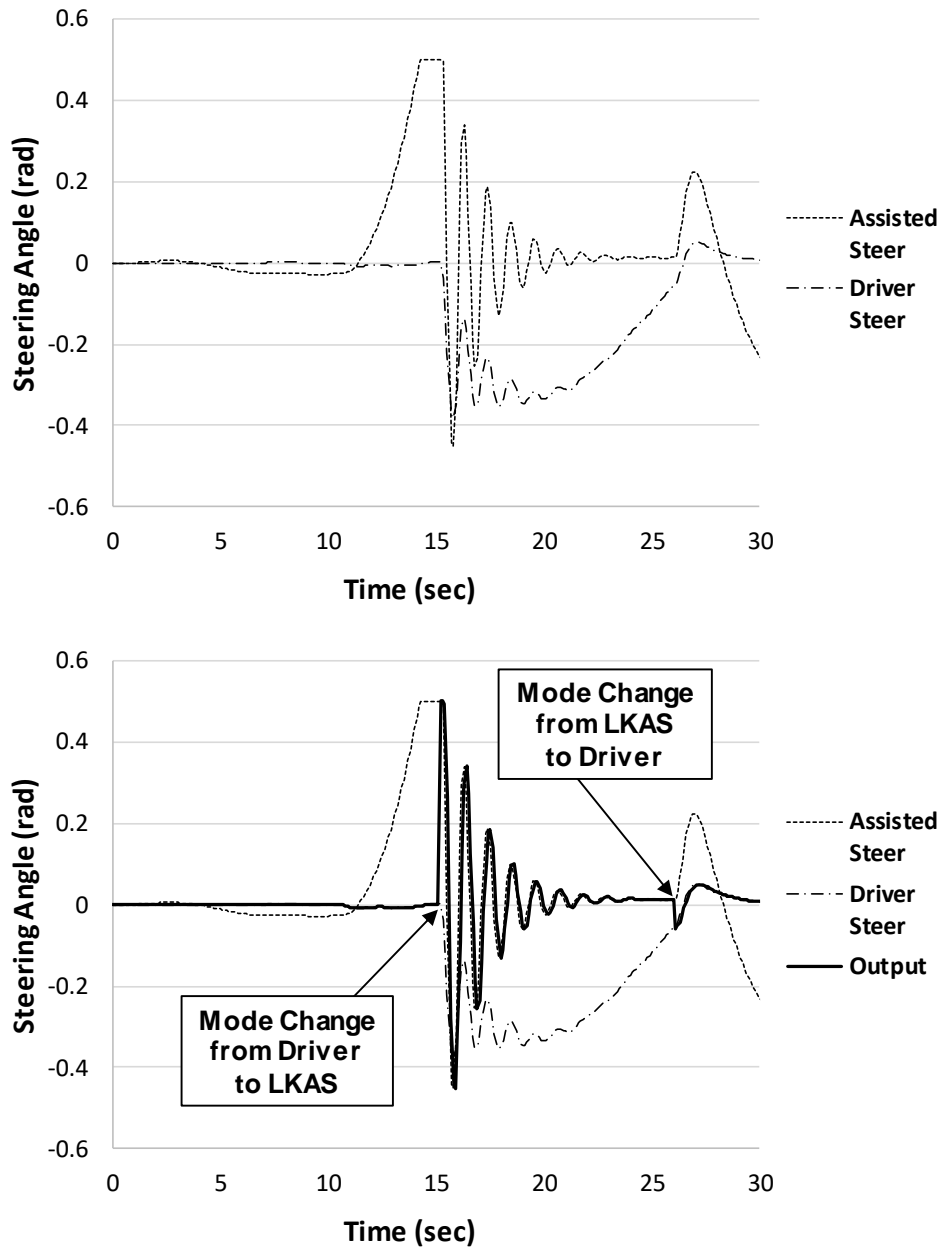


Figure 47. The steering angle selected using mode change.

three values: (1) driving steering angle, (2) assisted steering angle and (3) final output steering angle. Figure 47 shows the result. The driving steering angle and assisted steering angle is plotted using dotted lines, and the final out is presented using a solid line. As shown

in the graph, the mode change occurred twice while running the program. The first mode change occurred 15.2 seconds after the start. At this time, the check lane departure processing component of the lane departure detection factory detects that the distance between the ego vehicle and the center of the lane exceeds the threshold distance. It then sends a mode change event to the steering angle selection factory to change the factory's mode from Driver mode to LKAS mode. On the other hand, the second mode change occurred 26.1 seconds after the start. The change to driver mode processing component of the steering angle selection factory generates a mode change event that changes the factory's mode from LKAS mode to Driver mode.

We also measured the runtime overhead of the mode change mechanism. We measured the completion time of 10 mode changes and computed their average. Experimental results show that the mode change takes 0.53 milliseconds on average.

## **8.6 Evaluating Build Unit-based Allocation**

We used a Splash program that runs a synthetic workload to evaluate build unit-base allocation. The program and its component-build unit mapping configurations are shown in Figure 48. This program reads a data item from a source component once per second. It then uses six processing components placed in series to process the data item. Each processing component performs the same arithmetic operations.



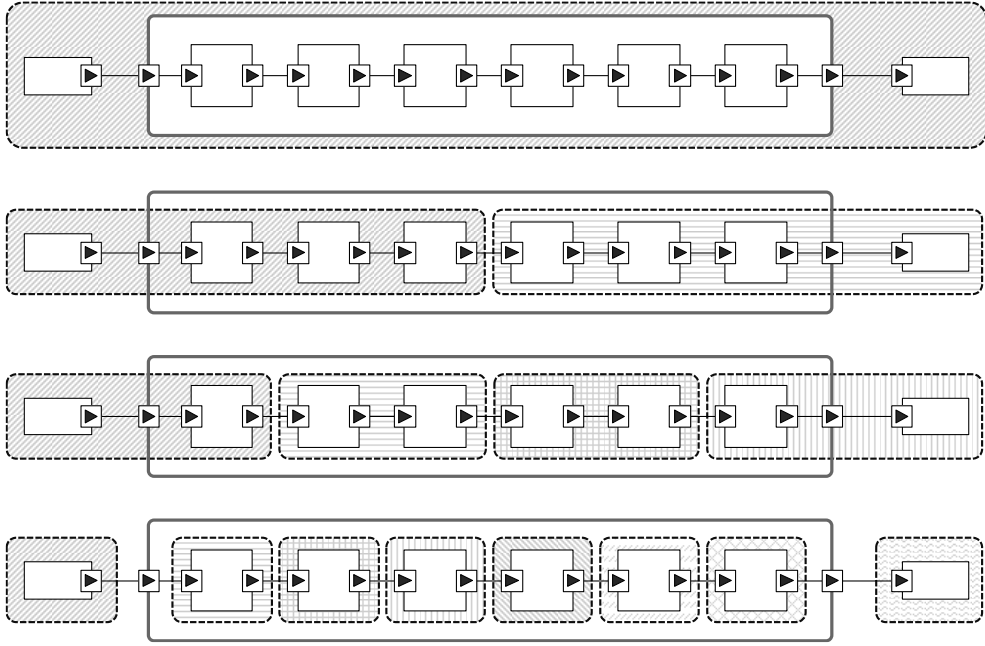


Figure 48. A Splash program and its build unit configurations.

The result is produced as an output using a sink component.

In the experiment, four component–build unit mappings are used. First, all components in the program are mapped to one build unit. Second, the source component and the first three processing components are mapped into one build unit, and the other three processing components and the sink component are mapped into one build unit. Third, each two adjacent atomic components are mapped into the one build unit. Finally, all atomic components are mapped into the different build unit.

We measured the end-to-end latency by varying the size of a data item from 4 bytes to 4 kilobytes. Figure 49 shows the results. As we increased the number of build units to 1, 2, 4, and 8, the average end-to-end latency increased to 75.79, 330.80, 591.87, and

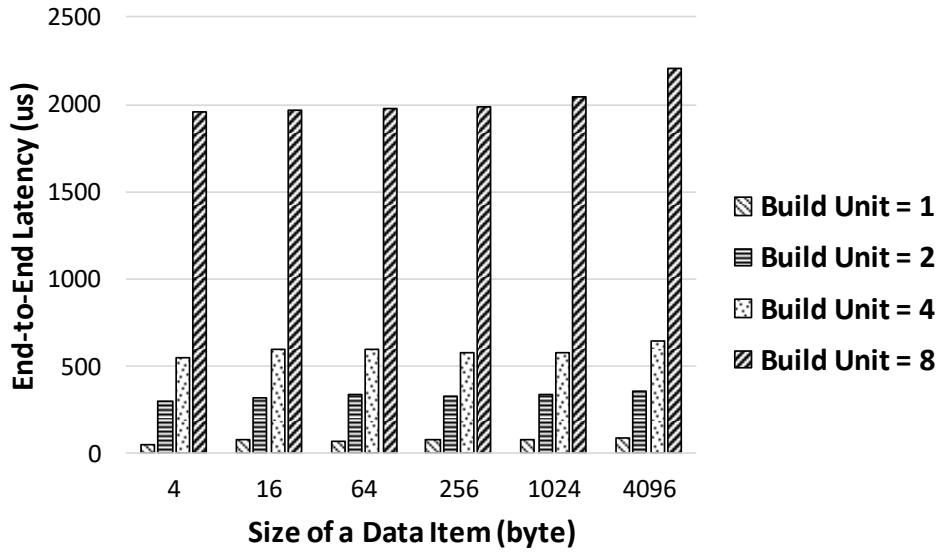


Figure 49. End-to-end latencies of the distinct build unit configurations.

2022.96 microseconds. This is because the number of threads and processes increases as the number of build units increases, resulting more context switch and communication overhead. Also, the end-to-end latency tends to increase as the size of a data item increases. This is because the communication delay time when sending a data item increases as the size of the data item increases.

## Chapter 9. Conclusion

In this dissertation, we presented a graphical programming framework named Splash for developing an autonomous machine. The Splash achieves our four design goals: (1) it provides an easy-to-use, effective programming abstraction, (2) it supports real-time stream processing for deep-learning based machine learning intelligence, (3) it provides programming supports for real-time control system of autonomous machine such as sensor fusion and mode change and (4) it supports performance optimization of software system running on a heterogeneous multicore distributed computing platform. In order to achieve these design goals, Splash first enables programmers to specify end-to-end timing constraints and provides timing semantics to handle such constraints at runtime. Also, it supports multithreaded process model to exploit parallelism explicitly from the distributed multicore computing platform. Splash provides exception handling semantics, rate control semantics, sensor fusion semantics and mode change semantics to support real-time stream processing and real-time control systems.

We validated the effectiveness of the Splash via the LKAS. First, the rate controller of the Splash reduced the jitter from 30.61 milliseconds to 1.66 milliseconds. The average lateral deviation and heading angle is reduced from 0.180 meters to 0.016 meters and 0.043 rad to 0.008 rad, respectively. Second, the sensor fusion and

mode change mechanism of Splash operated correctly with a run-time overhead of only 7 microseconds and 0.53 milliseconds, respectively. Finally, the average end-to-end latency was increased from 75.79 microseconds to 2022.96 microseconds as we increased the number of build units from 1 to 8.

The proposed approach can be extended in several future research directions. First, Splash should provide support for acceleration hardware such as GPU and NPU (neural processing unit). This support is important because many autonomous machines have recently been using GPUs or NPUs to improve inference performance while running deep learning algorithms.

Second, we aim to develop cross-layer optimization techniques for Linux kernel based on Splash’s timing semantics and runtime mechanisms. Currently, the Linux kernel has limitations to be used for autonomous machines because it has little support for real-time stream processing. However, if non-functional requirements and application context specified using Splash is passed to the Linux kernel, we will be able to develop new optimization techniques that will help Linux kernel to better support real-time stream processing.

Finally, we plan to design and implement various real-time applications for autonomous vehicles using Splash. To that end, we expect Splash to be further optimized in terms of performance and reliability. We look forward to apply Splash to edge computing technology, which is based on interoperability between cloud and embedded devices.

## Bibliography

- [1] M. Daily, S. Medasani, R. Behringer, and M. Trivedi, “Self-Driving Cars,” *Computer (Long. Beach. Calif.)*, vol. 50, no. 12, pp. 18–23, Dec. 2017.
- [2] A. Loquercio, A. I. Maqueda, C. R. Del-Blanco, and D. Scaramuzza, “DroNet: Learning to Fly by Driving,” *IEEE Robot. Autom. Lett.*, vol. 3, no. 2, pp. 1088–1095, Apr. 2018.
- [3] A. Mosavi and A. Varkonyi, “Learning in Robotics,” *Int. J. Comput. Appl.*, vol. 157, no. 1, pp. 8–11, Jan. 2017.
- [4] W. Shi, M. B. Alawieh, X. Li, and H. Yu, “Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey,” *Integration*, vol. 59, no. July, pp. 148–156, Sep. 2017.
- [5] 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.
- [6] S.-C. Lin *et al.*, “The Architectural Implications of Autonomous Driving,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems – ASPLOS ’18*, 2018, vol. 53, no. 2, pp. 751–766.
- [7] P. K. Gupta, “An overview of NVIDIA’s autonomous vehicles platform,” 2017.
- [8] A. Elkady and T. Sobh, “Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography,” *J. Robot.*, vol. 2012, pp. 1–15, 2012.
- [9] L. Reger, “The EE architecture for autonomous driving a domain-based approach,” *ATZelektronik Worldw.*, vol. 12, no. 6, pp. 16–21, Dec. 2017.

- [10] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, “Development of Autonomous Car – Part I: Distributed System Architecture and Development Process,” *IEEE Trans. Ind. Electron.*, vol. 61, no. 12, pp. 7131–7140, Dec. 2014.
- [11] “Simulink: simulation and model-based design.” [Online]. Available: <https://www.mathworks.com/help/simulink/index.html>.
- [12] N. Lac, C. Delaunay, G. Michel, J. Gévelot, and I. Moulineaux, “RTMaps: Real time, multisensor, advanced prototyping software,” in *First National Workshop on Control Architectures of Robots*, 2008.
- [13] J. Eker *et al.*, “Taming heterogeneity – the Ptolemy approach,” *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.
- [14] “MathWorks Documentation – Applications.” [Online]. Available: [https://www.mathworks.com/help/index.html?s\\_tid=CRUX\\_lftnav](https://www.mathworks.com/help/index.html?s_tid=CRUX_lftnav).
- [15] OMG, “Data distribution service (DDS) version 1.4,” 2015.
- [16] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ROS2,” in *Proceedings of the 13th International Conference on Embedded Software – EMSOFT ’16*, 2016, pp. 1–10.
- [17] G. Kahn, “The semantics of a simple language for parallel programming,” *Inf. Process. 74 (Proc. IFIP Congr. Stock. 1974)*, pp. 471–475, 1974.
- [18] G. Kahn and D. Macqueen, “Coroutines and networks of parallel processes,” 1976.
- [19] T. Huck, A. Westenberger, M. Fritzsche, T. Schwarz, and K. Dietmayer, “Precise timestamping and temporal synchronization in multi-sensor fusion,” *IEEE Intell. Veh.*

*Symp. Proc.*, no. Iv, pp. 242–247, 2011.

- [20] N. Kaempchen and K. Dietmayer, “Data synchronization strategies for multi–sensor fusion,” *Proc. IEEE Conf. Intell. Transp. Syst.*, no. November, pp. 1–9, 2003.
- [21] M. Geilen and T. Basten, “Requirements on the Execution of Kahn Process Networks,” in *European Symposium on Programming*, 2003, pp. 319–334.
- [22] A. A. Faustini, “An operational semantics for pure dataflow,” in *Automata, Languages and Programming*, no. 3, Berlin/Heidelberg: Springer–Verlag, 1981, pp. 212–224.
- [23] E. W. Stark, “A Simple Generalization of Kahn’s Principle to Indeterminate Dataflow Networks,” in *Semantics for Concurrency, Leicester*, 1990, pp. 157–174.
- [24] E. A. Lee and T. M. Parks, “Dataflow process networks,” *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
- [25] E. A. Lee, “A denotational semantics for dataflow with firing,” 1997.
- [26] E. A. Lee and E. Matsikoudis, “The semantics of dataflow with firing,” in *From Semantics to Computer Science*, vol. 9780521518, no. c, Y. Bertot, G. Huet, J.–J. Levy, and G. Plotkin, Eds. Cambridge: Cambridge University Press, 2009, pp. 71–94.
- [27] J. B. Dennis, “First version of a data flow procedure language,” 1974, pp. 362–376.
- [28] P. Derler, T. Feng, E. Lee, S. Matic, and H. Patel, “PTIDES: A programming model for distributed real–time embedded systems,” 2008.
- [29] M. Quigley *et al.*, “ROS: an open–source Robot Operating

System,” *ICRA Work. Open Source Softw.*, 2009.

- [30] “ROS 2 Documentation.” [Online]. Available: <https://index.ros.org/doc/ros2/>.
- [31] “ROS 2 – Roadmap.” [Online]. Available: <https://index.ros.org/doc/ros2/Roadmap>.
- [32] R. Tolosana–Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, “Enforcing QoS in scientific workflow systems enacted over Cloud infrastructures,” *J. Comput. Syst. Sci.*, vol. 78, no. 5, pp. 1300–1315, Sep. 2012.
- [33] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, “Adaptive Control of Extreme–scale Stream Processing Systems,” in *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*, 2006, pp. 71–71.
- [34] E. Wandeler, A. Maxiaguine, and L. Thiele, “On the use of greedy shapers in real–time embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 1, pp. 1–22, Mar. 2012.
- [35] J.–Y. Le Boudec and P. Thiran, *Network Calculus*, vol. 2050. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.
- [36] L. Thiele, S. Chakraborty, and M. Naedele, “Real–time calculus for scheduling hard real–time systems,” in *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, 2000, vol. 4, pp. 101–104.
- [37] I. Boutsis and V. Kalogeraki, “RADAR: Adaptive Rate Allocation in Distributed Stream Processing Systems under Bursty Workloads,” in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, 2012, pp. 285–290.
- [38] K. Santiago and S. Sarkinen, “System and method for hierarchical policing of flows and subflows of a data stream,”



2010.

- [39] Roman Avdanin, H. Bots, R. Y. Talla, Abhishek Chauhan, and R. Mirani, “Systems and methods for platform rate limiting,” 2015.
- [40] D. J. Abadi *et al.*, “Aurora: a new model and architecture for data stream management,” *VLDB J. Int. J. Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, Aug. 2003.
- [41] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, “Adaptive rate stream processing for smart grid applications on clouds,” in *Proceedings of the 2nd international workshop on Scientific cloud computing – ScienceCloud ’11*, 2011, p. 33.
- [42] Hugh Durrant–Whyte, “Multi–Sensor Data Fusion,” The University of Sydney, 2001.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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) (Cat. No.00CH37113)*, 2000, vol. 2, pp. 1434–

1439.

- [47] 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.
- [48] P. Geneva, K. Eickenhoff, 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.
- [49] 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.
- [50] D. Willner, C. Chang, and K. Dunn, “Kalman filter algorithms for a multi-sensor system,” in *1976 IEEE Conference on Decision and Control including the 15th Symposium on Adaptive Processes*, 1976, pp. 570–574.
- [51] S.-L. Sun and Z.-L. Deng, “Multi-sensor optimal information fusion Kalman filter,” *Automatica*, vol. 40, no. 6, pp. 1017–1023, Jun. 2004.
- [52] S. Blackman and Robert Popoli, *Design and Analysis of Modern Tracking Systems*. Boston: Artech House, 1999.
- [53] C. Coué, T. Fraichard, P. Bessière, and E. Mazer, “Multi-sensor data fusion using Bayesian programming: An automotive application,” *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 1, pp. 141–146, 2002.
- [54] D. Munoz, J. A. Bagnell, and M. Hebert, “Stacked Hierarchical Labeling,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6316 LNCS, no. PART 6, 2010, pp. 57–70.

- [55] L. T. X. Phan, I. Lee, and O. Sokolsky, “A Semantic Framework for Mode Change Protocols,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 91–100.
- [56] T. Chen and L. T. X. Phan, “SafeMC: A System for the Design and Evaluation of Mode-Change Protocols,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 105–116.
- [57] J. Real and A. Crespo, “Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal,” *Real-Time Syst.*, vol. 26, no. 2, pp. 161–197, Mar. 2004.
- [58] K. W. Tindell and A. Alonso, “A very simple protocol for mode changes in priority preemptive systems.”
- [59] J. Real, “Mode change protocols for real-time systems,” Universidad Politécnica de Valencia.
- [60] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
- [61] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronization,” *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Nov. 1990.
- [62] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, “Mode change protocols for priority-driven preemptive scheduling,” *Real-Time Syst.*, vol. 1, no. 3, pp. 243–264, Dec. 1989.
- [63] J. Y. T. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” *Perform. Eval.*, vol. 2, no. 4, pp. 237–250, Dec. 1982.
- [64] K. W. Tindell, A. Burns, and A. J. Wellings, “Mode changes in priority preemptively scheduled systems,” in *[1992] Proceedings Real-Time Systems Symposium*, pp. 100–109.

- [65] R. Gerber, Seongsoo Hong, and M. Saksena, “Guaranteeing real-time requirements with resource-based calibration of periodic processes,” *IEEE Trans. Softw. Eng.*, vol. 21, no. 7, pp. 579–592, Jul. 1995.
- [66] R. Stephens, “A survey of stream processing,” *Acta Inform.*, vol. 34, no. 7, pp. 491–541, Jul. 1997.
- [67] B. Goossens, “Dataflow management, dynamic load balancing, and concurrent processing for real-time embedded vision applications using Quasar,” *Int. J. Circuit Theory Appl.*, no. October 2017, pp. 1733–1755, Aug. 2018.
- [68] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, Jun. 2012.
- [69] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, p. 59, Jun. 2007.
- [70] H. Kopetz and G. Grunsteidl, “TTP – A time-triggered protocol for fault-tolerant real-time systems,” in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 524–533.
- [71] *IEEE Std 1588-2008, IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, vol. 2008, no. July. 2008.
- [72] A. Toshniwal *et al.*, “Storm@twitter,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data – SIGMOD ’14*, 2014, pp. 147–156.
- [73] M. Hirzel, S. Schneider, and B. Gedik, “SPL: An Extensible Language for Distributed Stream Processing,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 1, pp. 1–39, Mar. 2017.

- [74] Clay Breshears, *The art of concurrency*. 2009.
- [75] G. Pardo–Castellote, “OMG data–distribution service: architectural overview,” in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, 2003, pp. 200–206.
- [76] “Vortex OpenSplice.” [Online]. Available: <https://github.com/ADLINK-IST/opensplice>.
- [77] “Lane keeping assist with lane detection.” [Online]. Available: <https://www.mathworks.com/help/mpc/ug/lane-keeping-assist-with-lane-detection.html>.
- [78] Namyun Kim, Minsoo Ryu, Seongsoo Hong, M. Saksena, Chong–Ho Choi, and Heonshik Shin, “Visual assessment of a real–time system design: a case study on a CNC controller,” in *17th IEEE Real–Time Systems Symposium*, 1996, no. 3, pp. 300–310.

## 초 록

딥 러닝 기반 machine intelligence의 비약적인 발전으로 인해 autonomous machine들이 다양한 분야에서 활용되고 있다. 이런 기기들은 다양한 센서, 멀티코어 프로세서, 분산 컴퓨팅 노드를 장착하고 있기 때문에, 이들을 지원하기 위한 기반 소프트웨어 플랫폼의 복잡도는 빠른 속도로 증가하는 추세이다. 이에 따라 개발자들이 복잡한 소프트웨어 구조를 효과적으로 다룰 수 있도록 해주는 프로그래밍 프레임워크의 필요성이 대두되고 있다.

본 학위논문은 autonomous machine의 개발 과정에서 발생하는 문제들을 해결하기 위한 그래픽 기반 프로그래밍 프레임워크인 Splash를 제안한다. Splash라는 이름은 stream processing language for autonomous machine에서 앞의 세 단어의 첫 문자들을 따서 지어졌다. 이 이름은 물과 같이 흐르는 스트림 데이터를 다루기 위한 프로그래밍 언어와 런타임 시스템을 개발하겠다는 의도를 가진다. 본 논문에서는 복잡한 소프트웨어 구조를 효과적으로 다루기 위해 네 가지 디자인 목표를 설정한다. 첫째, Splash는 개발자에게 세부적인 구현 이슈를 숨기고, 쉽게 사용할 수 있는 프로그래밍 추상화를 제공하여야 한다. 둘째, Splash는 machine intelligence를 위한 실시간 스트림 처리를 지원할 수 있어야 한다. 셋째, Splash는 실시간 제어 시스템에서 널리 사용되는 센서 퓨전, 모드 변경, 예외 처리와 같은 기능들을 위한 지원을 제공하여야 한다. 넷째, Splash는 이기종 멀티코어 분산 컴퓨팅 플랫폼에서 수행되는 소프트웨어 시스템의 성능 최적화를 지원하여야 한다.

Splash는 실시간 스트림 처리를 위해 개발자가 프로그램 상에 본질적인 end-to-end timing constraints를 명시할 수 있도록 한다. 그리고 개발자가 명시한 timing constraints를 인지하고 이를 최대한 지켜주는 best-effort 런타임 시스템과 timing constraints의 위반을 모니터링하고 처리해주는 예외 처리 메커니즘을 함께 제공한다. 이런 런타임 메커니즘들을 구현하기 위해 Splash는 두 가지 기본적인 timing semantics를 제공한다. 첫째, 분산 시스템 상에서 모든 머신들이 공유할 수 있는 global time base를 제공한다. 둘째, Splash 상에 들어오는 모든 스트림 데이터 아이템에 자신의 birthmark를 기록하도록 한다.

Splash는 동시성 프로그래밍을 지원하기 위한 멀티 쓰레디드 처리 모델을 제공한다. Splash 프로그래머는 pthread라는 논리적인 수행 단위를 사용하여 프로그램을 개발할 수 있다. 그리고 Splash는 pthread들을 실제 운영체제의 수행 단위인 프로세스와 쓰레드에게 할당하는 과정을 돕기 위한 빌드 유닛이라는 language construct를 제공한다.

Splash는 timing semantics와 멀티 쓰레디드 처리 모델을 기반으로 실시간 스트림 처리와 실시간 제어 시스템을 지원하기 위한 세 가지 language semantics를 추가로 지원한다. 첫째는 스트림 데이터의 통신이나 처리 지연으로 인해 발생하는 지터나 바운드 되지 않는 큐 문제를 해결하기 위한 rate 제어 semantics이다. 둘째는 센서 퓨전 과정에서 시간적으로 동기화되지 않은 센서 입력들로 인한 타이밍 이슈들을 해결하기 위한 퓨전 semantics이다. 마지막은 가변적인 제어 시스템의 요구사항을 충족시키기 위해 수행 로직의 변경을 지원하는 모드 변경 semantics이다. 본 논문에서는 각각의 language semantics를 구체적으로 설명하고, 이를 실현하기 위한 런타임

메커니즘을 설계하고 구현한다.

Splash의 효용성을 검증하기 위해서, 본 논문은 Splash를 사용하여 LKAS 응용을 개발하고 이를 Splash 런타임 시스템 상에서 수행시키며 실험을 진행하였다. 본 논문에서는 rate 제어 메커니즘, 센서 퓨전 메커니즘, 모드 변경 메커니즘, 빌드 유닛 기반 allocation을 각각 선정된 성능 지표들을 사용하여 검증하였다. 첫째, Splash의 rate 제어기를 사용하면 지터가 30.61ms에서 1.66ms로 감소되었고, 이로 인해 주행 차량의 측면 편차와 방향각이 각각 0.180m에서 0.016m, 0.043rad에서 0.008rad으로 개선된다는 것을 확인하였다. 둘째, 센서 퓨전을 위해 제안된 퓨전 연산자가 설계된 의도대로 정상 동작하고, 평균 7us의 낮은 오버헤드만을 유발한다는 것을 확인하였다. 셋째, 모드 변경 기능의 정상 동작을 검증하였고 그 과정에서 발생하는 시간적 오버헤드는 평균 0.53ms에 불과하였다. 마지막으로, synthetic workload에 대해 컴포넌트들에 매핑된 빌드 유닛 개수를 1개, 2개, 4개, 8개로 증가시킴에 따라 평균 end-to-end 지연 시간은 75.79us, 330.80us, 591.87us, 2022.96us로 증가하는 것을 확인하였다. 이러한 결과들은 본 논문에서 제안하는 language semantics와 런타임 메커니즘들이 의도대로 설계, 구현되었고, 이를 통해 autonomous machine의 응용들을 효과적으로 개발할 수 있다는 것을 보여준다.

**주요어 :** Autonomous Machine, 실시간 스트림 처리, Rate 제어, 센서 퓨전, 모드 변경

**학 번 :** 2013-20785