Master's Thesis of Engineering

Designing and Implementing Core Runtime Libraries for Splash Programming Framework

Splash 프로그래밍 프레임워크를 위한 핵심 런타임 라이브러리의 설계와 구현

August 2019

Graduate School of Electrical and Computer Engineering
Seoul National University

Jaeho Ahn

Abstract

The paradigm of autonomous machines has shifted with the remarkable advancement in machine intelligence. To support machine intelligence, autonomous machines are now equipped with diverse sensors, heterogeneous multicore processors, and distributed computing nodes that require complex software architecture to utilize them properly. With the introduction of new sensors and computing powers, autonomous machines must now support applications that performs complex processing on unbounded sequences of stream data produced at real time. However, with the increase in software complexity it is becoming difficult for developers to coordinate the multiple streams of data and still meet the system requirements. To tackle such difficulty, we are currently developing a graphical programming framework we named Splash.

Splash provides effective programming abstractions that allow the users to establish multiple stream processing applications effortlessly. Splash also gives users the ability to specify genuine end to end timing constraints required by their system. The timing constraints in turn are automatically monitored for their violations by the Splash framework.

This thesis will introduce the components of the Splash graphical programming framework and focus on how Splash provides stream processing capabilities for its applications. The thesis will also introduce the internal workings of Splash's monitoring capability for end-to-end system timing constraints. Lastly the thesis will validate Splash application's functional correctness and tests its timing constraint monitoring capability by implementing ACC (Adaptive Cruise Control) and LKAS (Lane Keeping Assistance System) algorithm using Splash.

Keyword: Stream processing, timing constraints, programming abstractions, component-based programming
Student Number: 2017-27804

Table of contents

Figures

1. Introduction

Overtime, the number of autonomous machines that integrated machine intelligence to their system has increased tremendously. This integration allowed autonomous machines to achieve new feats of engineering, requiring diverse domains to come together to utilize the machine learning capabilities. The domains which are utilized include wide range of sensors, heterogeneous multi-core processors, and distributed computing. To efficiently make use of the newly generated sensor data and available computing powers, the underlying software complexity of the autonomous machine has increased exponentially. The complexity has reached a point where maintaining the execution and communication substrate of the system is becoming extremely difficult, making it harder for developers to focus on the application core logic.

It is also evident that applications for autonomous machines, especially for the ones that utilizes machine intelligence, often require processing of unbounded sequences of sensor data generated at real time[1][2]. To that end, in order to utilize the data that are produced at real time, and to make sure the data does not lose its validity, developers must deliver and process the data in a timely manner. This essentially signifies that the developers must maintain a thorough check on the system timing constraints.

To handle the increase in complexity and to satisfy the system timing constraints, modern autonomous systems require a modular development process that hides implementation details through abstraction and monitor end-to-end system timing constraints automatically. It must automate the monitoring process for two reasons. First, validating each component of its timing constraint is already an arduous process that often leads to programming faults. Increasing complexity will cause more timing faults to go unnoticed and cause critical failures to occur at runtime. Second, the end-to-end timing constraints are a result of different components of the system interacting with each other, therefore it is extremely difficult to test for validation separately. Testing for timing constraints within only isolated processes and data would be meaningless.

To satisfy the above requirements we are developing a graphical programming framework named Splash. Splash allows users to design their applications graphically and uses the design result to automatically establish the communication and execution substrate of the applications. Splash also allows users to specify the system timing constraints required during the processing and networking of data using its graphical programming language. The constraints are then monitored for violations at runtime, invoking different handlers for different violations.

Splash largely consists of three components. The first component is the schematic editor that allows the user to design their application graphically, in the form of a data-flow process network model. The second component is the code generator, which translates the result of the schematic editor into executable source codes. The third component is the Splash runtime library which is a set of libraries implemented to support the execution of the generated source code.

The Splash runtime library contains set of libraries that is used by the code generator to compose the executable source codes. The runtime library is comprised of two components. The first component is a set of APIs used by the code generator to establish stream connection between tasks. The first component also initializes the timing constraint monitors with the configurated values of the schematic editor and sets up the correct exception handlers for differing timing constraints. It also includes APIs for mode change capabilities that allows Splash applications to change its mode of execution at runtime. The second component of the runtime library establishes background processes that operates through the entire framework’s runtime. It monitors for the violation of the timing constraints initialized by the first component of the runtime library and calls the assigned handlers.

The Splash runtime library is implemented using a distributed communication middleware called DDS (Data Distribution Service). Splash utilizes the publisher and subscriber communication model of the middleware to provide abstraction from the system's topology and support soft real time communication through its rtps protocol [3].

This thesis focuses on the implementation of the Splash runtime library. Specifically, it explains how the execution and communication substrate of the Splash application is implemented and how they are used by the code generator to establish functionally correct source codes. The thesis will also introduce the mechanism behind the timing constraint monitoring capabilities of the runtime library. Lastly, the thesis will demonstrate the functional correctness and timing constraint monitoring capacity of Splash by implementing LKAS (Lane Keeping Assistance System) and ACC (Adaptive Cruise Control) algorithms using Splash. The Splash applications will be injected with a time complexity error to imitate an engineering mistake and be tested to see if it can handle the error successfully.

2. Background

2.1 Data Distribution Service

In order to support a real-time and scalable communication capabilities, the OMG (Object Management Group) has published a distributed communication middleware standard called DDS (Data Distribution Service) [4]. To aid in the readers understanding on how exactly DDS was used to implement the Splash runtime library, this section will briefly introduce the communication model and key components of DDS.

2.1.1 DDS communication model

DDS uses the publisher and subscriber communication paradigm for its communication model. The publisher and subscriber communication paradigm use a virtual global data space called topic which identifies each path of the data stream. The task that assumes the role as a publisher, sends data to a specific topic without the knowledge of the receiver. The task that assumes the role as the subscriber listens to a specific topic and receives data that are sent through it. Different subscribers that listens to the same topic receives identical data.

Publisher and subscriber communication model may use an intermediary broker that connects the publishers and subscribers that shares a topic. However, the original DDS standard published by the OMG does not use a broker. DDS instead, allows each publishers and subscribers to hold the meta data of their connected counter parts. The publishers and subscribers maintain the network location of their counterpart internally and use the known location to route their data.

DDS standard uses the UDP internet protocol as its main method of communicating data. On top of the UDP protocol, DDS maintains another software layer on called the RTPS wire protocol. The RTPS wire protocol uses the either-wise unreliable UDP protocol to implement a reliable and scalable data. The RTPS protocol provides DDS with plug and play connectivity allowing DDS applications to discover newly joining or leaving members at runtime. The RTPS protocol also creates a subnet out of participating nodes without the need of a broker allowing generic machines to form a large scalable network.

2.1.2 Key components and mechanisms

The Data Distribution System is mainly composed of two layers. The DCPS (Data Centric Publish-Subscribe) layer is the upper layer that provides the application layer with interfaces that allows easier access to its networking functionalities. The lower layer is the RTPS (Real Time Publish and Subscribe) layer, which holds the RTPS wire protocol previous mentioned in the paper. The RTPS layer provides implementations for the interfaces of the upper DCPS layer, supporting primary functionalities such as message passing, discovery, and interoperability. For the purpose of explaining Splash implementations, this thesis will only cover the DCPS layer.

The DCPS layer’s key components are as follows: DomainParticipant, Topic, Publisher, DataWriter, Subscriber, DataReader, Condition, Listener, and, Waitset. The DomainParticipant is a fundamental grouping unit that allows logical partitioning among DDS applications. Other DDS key components belong to a single DomainParticipant and each DomainParticipant is assigned with a Domain number. Different DomainParticipants can be assigned with the same domain number, and only the key components that belongs to the DomainParticipant with the same domain number can interact with each other. A set of DomainParticipants that shares the same domain number is called the Domain. A Domain is a logical space that separates DDS applications from each other, providing component-based programming, as well as fault tolerance. Before any other component can be created or accessed, a DomainParticipant must be created.

Same as the publisher and subscriber communication model, DDS DCPS layer's topic is a fundamental way of interacting between publishing and subscribing tasks. It acts as a virtual global data channel that can be used to publish and subscribe data. A topic is distinguished using a topic name, and each topic name is unique within a Domain. A topic must also be assigned with a specific data type that it publishes. A publishing application must specify a topic when publishing a data, and a subscribing application requests data via the topic.

A Publisher is a DCPS layer component that disseminates data to all applications that are subscribing to a specific Topic. While the DataWriter is used to pass a type specific data to the Publisher. An application uses the DataWriter to encapsulate and marshal its output so that it can be tailored to a specific topic. Once that is complete, the data is passed on to the Publisher. A Publisher can manage multiple DataWriters, but a DataWriter must only be assigned to a single Publisher. Therefore, a group of DataWriters can be managed through a single Publisher. Each DataWriter can be assigned only with a single topic and can output data to the assigned Topic.

Similarly, a Subscriber is a DCPS layer component that receives data and passes the data to the correct DataReader. The DataReader takes the data from the assigned Subscriber, de-marshals it into appropriate data type of the connected topic and delivers the data to the applications. A Subscriber can manage multiple DataReaders but DataReaders can only be connected to a single Subscriber. The DataReader can only be assigned with a single topic, and therefore can only read data from a single Topic.

A Condition is a component that represents a specific event in the DCPS layer. Different types of Condition component can be created depending on the type of event it represents. It is not independently used but must be attached to a Listener or a WaitSet. The Listener provides asynchronous call back ability to the applications. The Listener can assign multiple Condition that are created beforehand, and once a Condition is created and attached to a Listener, it will monitor for the attached Condition. One or more Conditions can be attached to the Listener. Once the Condition is fulfilled, it will asynchronously call upon a handler. The handler can be a user defined function, or a preset sub-routine provided by the middleware.

DDS applications use what is called a Simple Discovery Protocol to discover other communicating applications within a local area network. The SDP (Simple Discovery Protocol) is broken down into two steps. First a DDS application uses UDP multicast to announce it-self to its local network. The announcement is done by multicasting best effort messages to all local nodes, with message that include a global unique ID and its own IP address. The global unique ID identifies each DDS DomainParticipant; therefore each DDS application must create a DomainParticipant. Once the announcement is made, all DomainParticipant within the local network that shares the same domain number respond and establishes a connection. Since DomainParticipants act as an entry to all other DDS components, each connected component shares information on the number of Subscribers, Publishers, DataWriter, DataReaders, and most importantly what topic they are publishing and subscribing to. Once the meta information has been exchanged, DomainParticipants that has matching topics will start to exchange actual data [5].

3. Splash programming language

Splash provides Schematic editor to allow the users to design their system graphically using interfaces similar to designing a data -flow process network model. The schematic editor provides the user with a graphical programming language which is used to compose the data-flow process network model. Figure 1 shows the screen view of the schematic editor. Applications developed using the Splash framework is composed of series of nodes and edges. Each node acts as an operator that receives one or more data stream as input, processes them, and output the results. The Splash framework calls a node a component. A component is further classified into following types: processing component, source component, sink component, fusion operator, and a factory. Each component has one or more ports that acts as a gateway for data to enter and exit. There are different type of ports and each port are either an input port or an output port. An output port and an input port are linked together to express the stream data connection between two components. Figure 2 shows an example of a data-flow process network model that has been assembled using the Splash language constructs.

Through the combinations of different language constructs, Splash

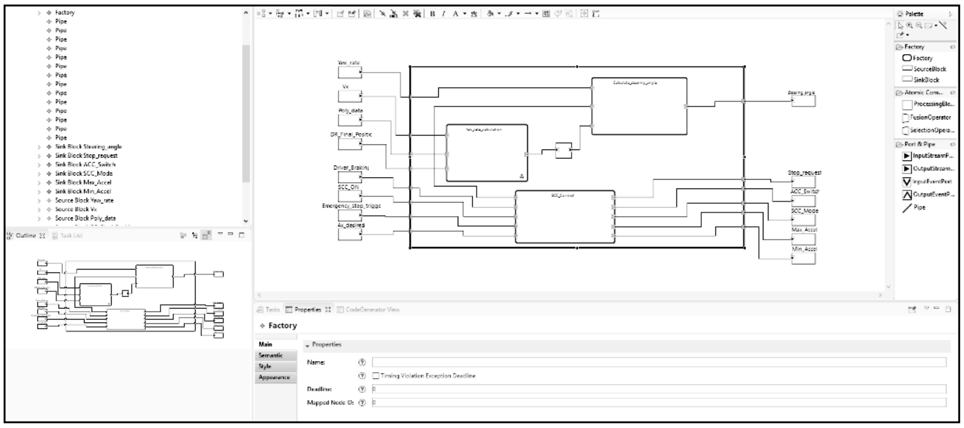
Figure 1. Splash schematic editor

provides underlying language semantics that collectively serve as a high-level programming abstraction. There are four underlying semantics and they are as follows: Basic operational semantics, timing semantics , in-order delivery semantics, rate-controlled data-driven processing semantics. This section first explains, each key language construct of the graphical programming language along with their key functionalities. The section will then describe each of the underlying semantics that the language constructs can be used to express.

3.1 Port

A port is the entry and exit for the stream data that flows between components. There are three types of ports, a stream data port for sending and receiving stream data, a mode change port for passing mode change signals, and an event port for passing event signals. For stream ports, each input and output port is assigned with a data type. The stream data port can only communicate using the assigned data type.

notepad The mode change port does not require a passing of

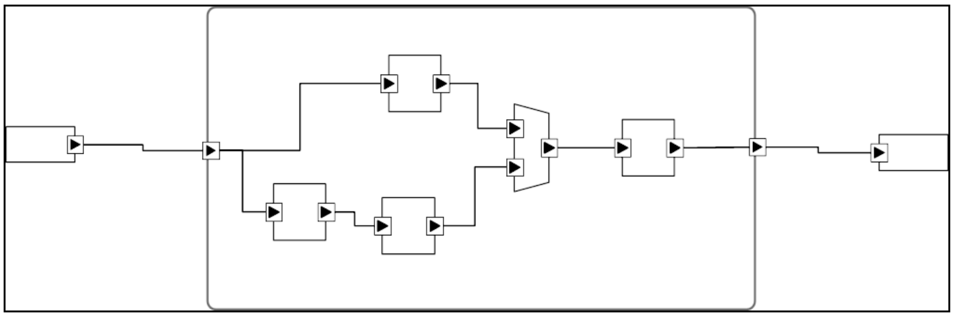

Figure 2. Splash application schematics

sophisticated data, but rather needs to send intermittent signal that notifies the receiver of a mode request. The event port is used to send and receives event signals that are pre-defined by the Splash framework therefore its data types are already defined. Both the mode change port and the event ports are
differentiated from the stream port because, data streams can be periodic and requires a constant data delivery, while both the mode change signal and event signals are relayed sparingly.

3.2 Processing component

A processing component represents a user defined operator that receives one stream data as input and processes them with the user's logic. Once the data has been processed, it outputs the processed data using the attached output port's interfaces. A processing component can be attached with multiple stream input ports, and multiple stream output ports. When a processing component is translated into executable source codes by the code generator, it provides the users with a skeleton code that provides programmable area. User can apply their own logic to the area using the attached ports as input and output interfaces.

3.3 Source component

A source component receives external signal from machines outside the Splash system and produces stream data from the result, acting as an entry to the Splash's data streams. It does not require an input port because it will receive signals originating from outside the Splash application system, but it does require a single output port for relaying the generated stream data. When translated into executable codes, the source component provides a user programmable area that allows the user to establish their own interface with the outside machine.

3.4 Sink component

Similar to the source component a sink component links the internal Splash system's stream data to the external machine. It must be attached with a single input port and receives a single stream of data from another Splash component. It does not require an output port because it will produce and output data compatible to the outside machine, using the machine' interface. When translated into executable codes, the sink component provides a user programmable area that allows the user to establish their own interface with the outside machine.

3.5 Fusion operator

The fusion operator represents an operator that merges multiple stream data into a single stream data output. It is attached with multiple input ports and a single output port. The fusion operator does not require any user programmable areas, therefore once it is translated into executable codes, it only contains automatically generated communication and execution substrates.

3.6 Factory

A factory is a grouping component that contains other Splash language constructs. It aids the user with compartmentalizing systems and acts as a unit for mapping Splash generated source codes, to an OS process. Each factory is attached with multiple input ports and multiple output ports that connects its internal language constructs with the outer constructs. The factory can also be attached with a mode change port in order to have multiple mode that contains different combinations of the language constructs. The mode can be changed at runtime, and it is triggered when a mode change signal is received through the mode change port.

3.7 Underlying semantics

There are total of four types of underlying semantics that the language constructs can be used to implement. The first semantic is the basic operational semantics. The basic operational semantics represent functionalities that the users require to develop the communicating and processing part of stream processing applications. The basic operational semantics are further divided into data processing semantics, path control semantics, and triggering semantics. The data processing semantics allow users to freely program their own stream processing operations. Path control semantics allow users to control the inputs and outputs of stream data and control the connections between the inputs and outputs. Triggering semantics allow the stream processing operations to be triggered by data arrival, time, and event.

The second semantics is the timing semantics. Timing semantics enable the users with the act of detecting end-to-end timing constraints. In order to do so, Splash supports two things. First, Splash synchronizes all local clocks between nodes that are using the Splash framework. Splash utilizes the precision time protocol daemon provided by the Linux kernel to do so [6]. Second, Splash maintains a timestamp in each of the data delivered through the data stream. Each time the source component receives an external signal, it assigns a timestamp called the birthmark and outputs the result as a Splash data stream. A birthmark keeps record of the time the data has been created (entered the Splash application's system). The assigned birthmark is maintained through out the data's entire flow through the system. When the input data is processed into a new output data, the output data inherits the input data's birthmark. If several data are merged into a single output, the oldest among the different birthmarks are assigned to the output. With the two functionalities in place, the timing constraints the users can monitor for violations are as follows. The *freshness constraint* configures the amount of time it takes before a generated data is deemed expired, and unviable. The source component assigns each data that it introduces to the system, a freshness constraint. Therefore, users assign a freshness constraint using the source component in the schematic editor. The *correlation constraint* configures the maximum difference between the birthmarks, carried by the group of distinct stream data from different input port. The correlation constraint is used by the Fusion operator to filter out data that should not be merged together due to it being too further apart in terms of its birthmarks. Therefore, users assign a correlation constraint using the Fusion operator in the schematic editor. The *rate constraint* configures the numbers of data that should be produced per second. This is done to ensure a continuous processing of data and to prevent jitters from occurring on the downstream node. Often if data is not available within the set constraint, users can provide handlers that outputs an extrapolation signal that tells the downstream node they must extrapolate a data at this time. Users assign correlation constraint using the output stream port.

The third semantics is the in-order delivery semantics. The in-order delivery semantics guarantees that data items are always delivered in order of their birthmarks. Since network variations make it difficult to control the delivery order of transmitting data, the input data are re-ordered at the input queue of the stream data port. Each time a data is enqueued to the stream data port, the enqueued data are sorted in non-decreasing order of their birthmarks.

The last semantics is the rate-controlled data-driven processing semantics. As mentioned earlier, Splash supports three types of triggering mechanisms for its stream processing operations. Among the three, the default triggering mechanism is the data triggering semantics. However, data triggered operations often lead to side effects such as queue overflows and jitter. This is because if stream processing operations are triggered each time a data arrives, it will output with irregular rate. If the data arrival rate is too fast, it can cause a queue overflow at the down stream input port, if the data arrival rate is too slow it can cause jitters to occur at the down stream processing operations. To compensate for these limitations, Splash provides users with what is called a rate controller. A rate controller is a transparent module that attaches itself to the output port if the user chooses to. The rate controller guarantees the production of exactly one data item in each time window with size 1/r where r is the rate constraint. If there are no data items to be sent within the time window, the output port will raise a rate constraint violation and call the registered handler. The user can choose to use the handler to extrapolate an output or simply send an extrapolation command. When a processing component receives an extrapolation command it must invoke a function that performs a extrapolation task that will generate an input instead of receiving it from the input stream data port.

4. Splash runtime library

Splash framework provides the schematic editor to streamline the development process of stream processing applications by allowing users to design their programs graphically. The code generator uses the schematic editor's output to generate source codes that has been established with communication and execution substrate. The source codes also include runtime monitors that detects the violation of the annotated timing constraints. This section will illustrate how each language components are implemented in the runtime library.

4.1 Runtime library

The Splash runtime library is a collection of libraries used by the code generator to implement the functionalities designed by schematic editor. Therefore, it must contain entities or functions that reflects the different Splash language construct and their corresponding timing constraints. The Splash runtime library supports the execution, communication, timing behavior initialization/monitoring, mode change, and exception handling. The runtime library is implemented as a series of class definitions, each representing their corresponding language construct. The Splash library is implemented using the DDS middleware; therefore, it inherits the publisher and subscriber communication model. This section will explain how each language constructs are implemented in the runtime library.

4.1.1 Port runtime library

Port attaches itself to a component and becomes the communication interface for the component language construct. Since Splash uses publisher and subscriber communication model, it must be able to provide interfaces for establishing a connection with a specific topic, and to exchange data through the topic. Two classes, `Input_port` and `Output_port` is created to be the base class for all other types of port language construct. `Input_port` and `Output_port` classes are not directly used but exists as a base implementation for other inheriting classes.

The `Input_port` class maintains a single FIFO queue for storing input data and creates two tasks that runs simultaneously. The first task will receive the input data and store it into the queue while the second task will generate an asynchronous signal to call a registered handler each time the data arrives into the queue. The handler will notify the connected component language construct of the available input data and pass the input data to the connected component language construct. The first task utilizes a DDS DataReader to receive data into the queue. The second task utilizes a DDS Listener to generate an asynchronous interrupt and call a handler. However, to use the DataReader and the Listener, they must both be configured with the Topic they will be receiving data from as well as maintain a connection with a Subscriber. To do so the `Input_port` class implements a method called `attach()` which receives a component language construct's object and a topic name as input parameters. The method creates an access to a DDS Subscriber that are maintained in the component language construct's object, both of which are used to create the DataReader and the Listener. The passed topic name is used to subscribe to the configured topic.

The `Output_port` is identical to the `Input_port` class and maintains a single FIFO queue for storing transmit ready data. However, while the `Input_port` class manages two tasks for receiving data and notifying the connected component language construct, the `Output_port` class only maintains a single task. The single task transmit data from the queue whenever a data is available. The connect component language construct will access the output queue independently to insert the prepared output data. Thus, the `Output_port` class does not require any Listener but only maintains a DataWriter. Same as the `attach()` method of the `Input_port` class the method receives a component language construct's object to access its DDS Publisher as well as the its publishing topic name.

The `attach ()` method of both `Input_port` and `Output_port` will

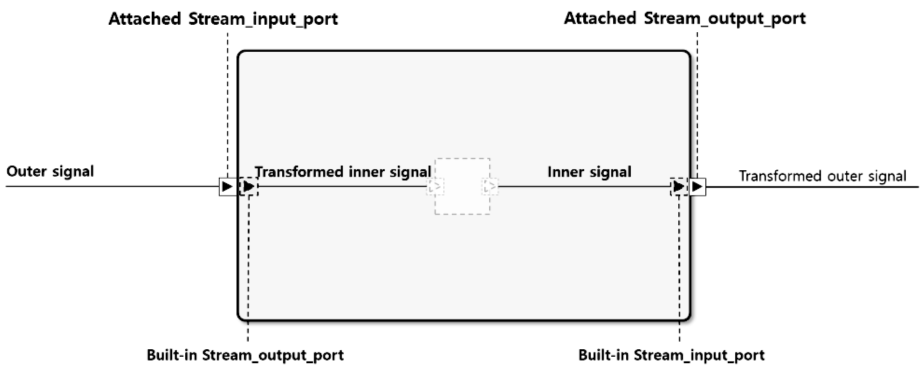
Attached Stream_input_port
Attached Stream_output_port
Outer signal
Transformed inner signal
Inner signal
Transformed outer signal
Built-in Stream_output_port
Built-in Stream_input_port

Figure 3. Factory's built-in stream data ports

determine what kind of data types the attached Topic will be exchanging. DDS uses a typed interface, therefore some of the communication interfaces provided by DDS must know the data type of the exchanged data before runtime. We utilize the `attach ()` method to configure all related runtime libraries with the correct data types.

The first class to inherit the `Input_port and Output_port` is the `Stream_input_port` and `Stream_output_port` class. `Stream_input_pot` and `Stream_output_port` implements the stream data port. Both the `Stream_input_port` and `Stream_output_port` class differs from its base class, only from the fact that both the `Stream_input_port` and the `Stream_output_port` has an additional method called `initialize()`. Both classes' `initialize()` method receives an integer value for configuring its queue size. However, the `initialize()` method of the `Stream_output_port` receives an additional input parameter for configuring the rate constraint. The parameter for configuring the rate constraint expresses the amount of the data the stream output port must produce per second.

The second and third class to inherit the base port classes are the `Event_port` and `Mode_switch_port`. Both ports do not require an initialization method because it does not require a carefully selected queue size. The Splash runtime library provides a default queue size for both that can be configured if required. The event port and mode switch ports will deliver signals scarcely therefore does not utilize an rate constraint either. If a user finds a data that takes on a characteristic which borderline between stream data and an event, it is better to use a stream data port with a proper processing logic. Both classes do have an `attach ()` method but the data type it assigns to the connected components do not vary. An `Event_port` has set of define events that it represents as integers while a `Mode_switch_port` only needs to transmit a integer value to signal a which mode the factory should switch to.

4.1.2 Component runtime library

The component language construct acts as a logical node that processes inputs and outputs the result. Therefore, one or more port language constructs are attached to a component language construct to become its communication interfaces. In order to implement this dynamic between the component and port language construct, each processing component must have one or more of the following DDS components: DomainParticipant, a Subscriber, and a Publisher. As explained in the background section of this thesis, for a DDS application to discover each other, a DomainParticipant is required. While a Subscriber or a Publisher is required to utilize a DataReader or a DataWriter. This section will explain how the port runtime libraries and DDS components are used together to implement component language construct.

A. Factory

Factory component is the largest building block of the Splash language construct. Factory is viewed as a unit of execution and a container for all other language constructs except source components and sink components. A factory component is mapped as a thread to a process same as other component constructs, but it determines which set of components run within the process, therefore logically all other component constructs are mapped to a factory. This design choice was selected deliberately to utilize the lightweight nature of threads compared to processes and to make mode changes faster.

Each `Factory` class will have a DomainParticipant that acts as an entry way to connect all other DDS substrates used by component constructs within the `Factory`. Factory component have stream data ports and mode change ports to relay data stream and to receive mode change requests. Consequently, `Factory` class must also have a Publisher and a Subscriber. `Factory` class can be attached with a `Stream_input_port` and a `Stream_output_port` object by using each ports `attach()` method. However, since a `Factory` class must connect an external stream to an internal stream, for each stream input port connected to the factory, it must also have a corresponding built-in output port. Figure 3 illustrates the implementation of factory stream data ports. `Stream_input_port` attached to a factory component relays the input data to the corresponding built-in `Stream_output_port` and transmits the data to the internal data stream. The main role of `Stream_input_port` attached to factory component is to relay the external data stream to an internal one. It becomes vice-versa for `Stream_output_port` attached to the factory component.

Alternative design choice would have been to remove stream data port from the factory component, and just consider the first component language construct contained within the factory to be the entry point for external stream data. However, since component constructs are implemented as threads, components within the same factory can communicate using inter thread communication such as pointer passing to implement pub sub communications [7]. Basic DDS communication stack employs RTPS protocol and UDP protocol for distributed computing, which creates unnecessary delay for IPC within the same node. Therefore, ports that creates an entry and exit to factory component can utilize the DDS communication stack while ports attached to components within the factory component can utilize different inter-thread communication to decrease the communication delay. Currently this optimization is still in works and components within factory components use the original DDS communication stack exchange data.

For mode changes, previously discussed `Mode_switch_port` is attached to the factory to receive mode change requests. For each mode, factory maintains a thread pool of different runtime library components. When a mode change request arrives, all factory `Stream_input_port` blocks further input and finishes processing the remaining input data in its queue. Once the remaining Input data is done processing, the factory component switches the thread pool, and unblocks the `Stream_input_port` to restart receiving data. This is where the previous discussed design choice shines its effectiveness. A method `map_mode()` receives an integer and a pointer to an object of a component construct, that maps component constructs to a mode. `Mode_switch_ports` receive an integer that corresponds to the integer used to configures modes by the `map_mode()` method, the starting default mode is 1. Once all internal components been created and the `Factory` component executes the run () method to activate the factory.

B. Processing component

A processing component performs user programmed computation on input data and outputs the results. The `Processing_component` class have both a Subscriber and a Listener that connects itself to the factory components DomainParticipant at creation. The `Processing_component` can be attached with one or more `Stream_input_port` using its Subscriber, therefore it can receive more than one type of input data. However, it can only operate on one input data at a time, consequently allowing each `Stream_input_port` to have its own logic that processes the input. For data received in all of the attached `Stream_input_port`, data are chosen in order of its birthmark and passed to the matching user logic. `Processing_component` can be attached with multiple `Stream_output_port` as well. `Processing_component` has a `write()` function that receives an output data and topic name as its input parameter. The topic name is used to determine which `Stream_output_port` is used to send the output data. Using all of the above `Processing_component class` provides a way for users to program their own processing logic. To do so, `Processing_component` provides an empty method `user_function()` for each attached `Stream_input_port`. When input data is available, `user_function()` is invoked by a `Stream_input_port`'s listener and receives input data as its parameter. Developers can use the input data and the `write()` method to implement their own logic.

C. Source component and sink component

Source component and sink component connects external device to the internal Splash data stream. They have similar architectures to the processing component because source and sink components must also provide users with programmable areas where they can implement their own interface with the external device. Since they exist outside the boundary of a factory component they are implemented as a separate process not a thread. This is done because, source component and sink component are not part of mode changes. It is also because multiple factory components can share a source component or a sink_component, therefore if they are mapped to a process that includes other component constructs, it will not be able distribute data equally to all factory components.

The `Source_component` class maintains a Publisher but does not require a Subscriber because it receives data from the external device via the device's interface. The Subscriber connects to the factory's DomainParticipant at creation. `Source_component` can only attach a single `Stream_output_port` which is used to relay the external data and publish it to the internal stream. The `Source_component` also provides a `user_function()` method to allow users to program their own interface with the external machine. The trigger for the `user_function()` cannot be invoked by a port but must be done independently by the user. The `Source_component` will only provide a programmable area that is mapped to a thread, and output interfaces for the user. The `Source_component` assigns birthmarks to each data for detecting freshness constraint. After the `Stream_output_port()` is attached the `Source_component` uses its own `initialize()` method to determine how much freshness constraint we would assign. The `initialize()` method will receive integer value as its input parameter for configuring the freshness constraint. The unit will be in miliseconds.

The `Sink_component class` functions as the exact opposite of the `Source_component` therefore must have a Subscriber and attach it to the factory's DomainParticipant at creation. It also provides programmable areas for the user to establish connection interface with external device.

Since `Sink_components` must receive Splash stream data and relay it to an external machine, a `Stream_input_port` is attached to its Subscriber, allowing the user programmable method to be triggered by incoming internal stream data. `Sink_component` also has the same `user_function()` method as the `Source_component`. `Sink_component's user_function()` however receives input data as its parameter, using the attached `Stream_input_ports` Listener.

D. Fusion operator

A fusion operator merges multiple input stream into single output stream therefore, `Fusion_operator` class have both the Publisher and Subscriber and can attach multiple `Stream_input_port` and a single `Stream_output_port`. Fusion operator does not require any user programmable methods so it does not provide empty methods to the users. In order to provide varying options to the users, `Fusion_operator` provides its own `initialize()` method that configures the optionality of each attached `Stream_input_port`. When data is merged into a single output, mandatory `Stream_input_port` blocks the merging process until there are available data. Optional `Stream_input_port` uses a default value if there are no available data and does not block the merging process. `The initialize()` method also receives an integer parameter that configures the correlation constraint between the attached `Stream_input_port`. The unit is in milliseconds.

```
Module example_data_definition
{
        struct data_formation_1
        {
                double trans_data_1;
                long    trans_data_2;
        };
        …
};
```

Figure 4. Example IDL file

4.2 Data types

As mentioned earlier DDS uses typed interfaces, therefore Splash runtime libraries must be able to pass the user configured data types to the underlying DDS substrates. To do so, Splash schematic editor produces a set of IDL (Interface Definition Languages) that is used by the code generator to create language specific data types. An example IDL file can be seen in Figure 4. Splash runtime library is implemented as type generic template interface therefore when it is used by the code generator, the language specific data types assigned to the runtime library's template interfaces.

4.3 Timing constraint monitoring

For each timing constraints monitored by Splash, different language construct is used to annotate the constraints. The output stream data port is assigned with the rate constraint and source component is assigned with freshness constraint. Lastly the fusion operator is assigned with the correlation constraint. The corresponding runtime library to the language constructs also implements the mechanism for detecting the timing constraints.

For `Stream_output_port` class, after its has invoked `initialize()` and `attach()`, it automatically creates tasks that maintains a timer and a count of how much data has been issued through the port. Every second the `Stream_output_port` keeps count of the data that has been issued and checks them against the configurated rate constraint to monitor for violations. All of monitoring tasks, including rate constraint starts after the factory component has invoked its `run()`.

For source components, similar to stream output data port, invokes initialize() function to configure the freshness constraint. However, the monitoring for the freshness constraints does not occur on the source component. The monitoring for the freshness constraints occurs at each of the input stream data port and output stream data port. Each data carries a freshness constraint value, therefore, each time a data arrives at the stream data port, its birthmark is compared against the current time to calculate its latency so far. It is then compared against the freshness constraint value to see if it has violated it.

For fusion operators, it also invokes initialize() function to configure its correlation constraint. For each input stream data port attached to the fusion operator, fusion operator assigns each port with either mandatory or optional. For all port deemed mandatory, the fusion operator selects the oldest data among the input queues. It then checks oldest data from other input queues and compares them to see if the difference is within the correlation constraint value. If at the time the mandatory port does not has data in their queues that satisfies the correlation constraint, it creates a default item for delivery and contains a Boolean that signifies that the data that has been fused is a default value.

4.4 Programming model

The Splash runtime library is a collection of libraries that is used to implement applications designed by the schematic editor, therefore they do not have a context as a runnable process by themselves. The Splash code generator must take the Splash runtime library and assemble them into runnable processes. Figure 6 shows a generated source code that illustrates how the runtime library is assembled to implement a Splash application.

As mentioned in the runtime library, the factory component is the unit of execution that chooses which set of component construct threads to run. Therefore, in Figure 5, before any other runtime library is created, a (1) `Factory` object is created. (2) Afterwards, `Stream_input_port` and `Stream_output_port` are created to receive stream data. Since some of Splash runtime library is implemented using template interfaces, interfaces like `Stream_input_port` and `Stream_output_port` both have data types assigned to them. The language specific definitions of the

```
#include "…/Data_type_1.hpp"
…
#include "…/Data_type_11.hpp"

int main(void) {

   Factory Example_factory;                                                                    ...(1)

   Stream_input_port<Data_1::Msg> factory_input_port_1;                                        ...(2)
   …
   Stream_output_port<Data_4::Msg> factory_output_port_2;

   Input_Mode_switch_port factory_input_mode_port_1;

   factory_input_port_1.initialize(30);                                                        ...(3)
   …
   factory_output_port_2.initialize(10,20);

   factory_input_port_1.attach(&Example_factory,"Factory_input_topic_1");
   …
   factory_output_port_2.attach(&Example_factory,"Factory_output_topic_2");

   factory_input_mode_port_1.attach(&Example_factory,"Factory_input_mode_switch_topic_1");

   Procesing_component PC_1 (&Example_factory);                                                ...(4)

   Stream_input_port<Data_5::Msg> pb_input_port_1;

   Stream_output_port<Data_6::Msg> pb_output_port_1;

   pb_input_port_1.initialize(20);                                                             ...(5)

   pb_output_port_1.initialize(10,40);

   pb_input_port_1.attach(&PC_1,"pb_1_input_topic");

   pb_output_port_1.attach(&PC_1,"pb_1_output_topic");
   …
   Example_factory.map(1,&PC_1);                                                               ...(8)
   …
   Example_factory.run();

}
```

Figure 5. Runtime library assembled into source code

data types are included in automatically generated header files like `Data_type_1.hpp`. The factory is also attached with a mode switch port to receive mode change requests. Mode switch ports do not require template interfaces because it already knows that it only needs to receive integer data types. (3) The created stream data ports first use the `initialize()` method to configure their queue sizes. The `Stream_output_port` receives additional input parameter for

```
#include "…/Data_type_1.hpp"
…
…
#include "…/Data_type_11.hpp"

template <typename t>
void Processing_component<t>::user_function(t input_data, )          ...(6)
{
        Data_5::Msg Output_data;
        …

        //user logic here
        …
        write(&Output_data, "pb_1_output_topic");                     ...(7)

}
```

Figure 6. Component construct's user_function

configuring its rate constraint. Finally the stream data ports use the `attach()` method to assign themselves to the factory and to configure their communicating topic.

(4) After factory component has been created and established with the necessary ports additional component constructs like the processing component is created. The processing component receives the factory components address at creation to register itself to the factory and access its DomainParticipant. (5) Stream data ports are again created and attached to the processing component to receive input data and output the processed results. Figure 6 shows the structure of the empty method `user_function` that is used by the processing component to operate on the input data with the user programmed logic. (6) The empty function receives its input data through the parameter and the variable for the output data is automatically generated for the user to use when outputting the result. (7) The processing component's `write()` method receives the topic of the attached stream output port and the output data

to transmit the output data using the correct `Stream_output_port`.

Back at Figure 5, after every internal component construct has been created and prepared, (8) the factory then maps other component construct to a mode using its `map()` method. After a component construct has been mapped to a mode, its threads will only execute when the factory is currently running with its mode. Once every configuration is complete, the factory invokes its `run()` to activate itself.

5. Validating Splash through experimentation

This section will validate Splash's ability to implement a functionally correct algorithm as well as its capacity to handle timing constraint violations. The section will first explain the experiment environments set up. Second the section will describe the application developed using Splash. Lastly, the section will present the experiments result.

5.1 Experiment environment

The experiment was performed using a driving simulator called PreScan. PreScan driving simulator is a physics-based simulation platform that is used by automotive industries for developing

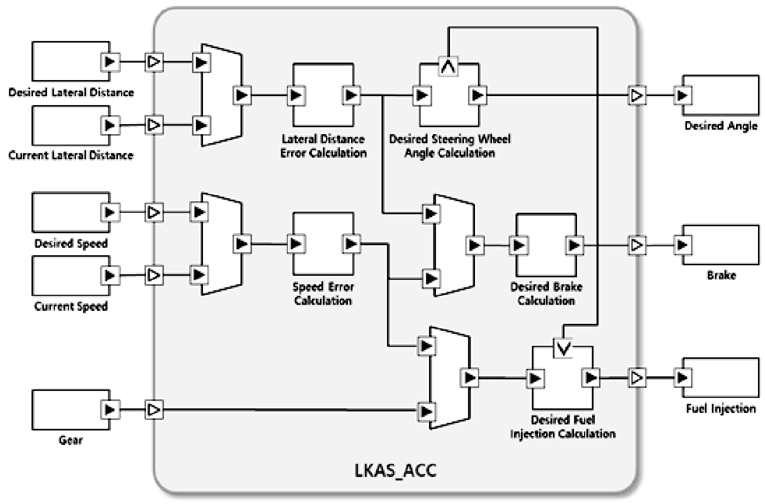

Figure 7. ACC LKAS Splash application

advanced driver assist system, vehicle to vehicle communication application, and other automotive related systems. We used the PreScan driving simulator to create a virtual vehicle and map that allowed the vehicle to drive in the map. Figure 8 shows the map that was created to simulate the driving environment. The simulated vehicle produced sensors values while the LKAS and ACC algorithm received its sensor data as input. The LKAS and ACC algorithm processed the received data to produce control signals that in turn controlled the simulated vehicle. This essentially created a software-in-the-loop environment. Two computing nodes are used to accommodate the experiment. First node ran the PreScan simulator while the second node ran the Splash application. Figure 10 shows the specifications of the two computing nodes.

5.2 LKAS and ACC application

The application developed using Splash receives lane detection data, desired speed, current speed, and current gear as input and produce the desired steering wheel angle. Brake, and fuel injection to adjust the speed and steering wheel angle of the vehicle. The lane detection data is divided into lateral distance and desired lateral distance. The lateral distance detects how far off the current vehicle's position is from the center line of the driving lane while the desired lateral distance is the pre-configured value that is adjusted to provide the right amount of distance. The algorithm uses the two lateral distances to calculate the desired steering wheel angle. The algorithm uses the lateral distances and the current speed to calculate the current brake strength of the vehicle. Finally, the algorithm uses the desired speed, current speed, and the current gear to calculate the fuel injection amount of the vehicle. The fuel injection controls the current target velocity of the vehicle.

5.3 Experiment and result

First to validate Splash application's functional correctness, we ran the driving simulation with the Splash LKAS and ACC as its control algorithm.

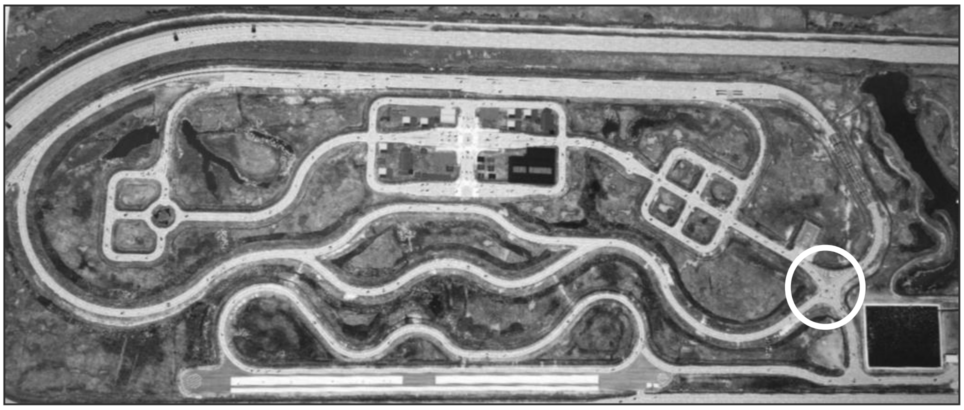

Figure 8. Simulated road and the location of failure

The Simulation was able to cruise through the entire simulated track.

Another experiment was conducted on the Splash's LKAS ACC application to verify Splash's timing constraint violation handling capability. Through heuristic experimentations it was determined that the *Desired Lateral Distance source tag* found in Figure 7 requires 80ms freshness constraint. During the default cruising of the simulated vehicle, the freshness constraint of the *Desired Lateral Distance source tag* was satisfied for its runtime duration. For experimentation, a fault was introduced into the *Desired Steering Wheel Angle Calculation processing component* to induce freshness constraint violations. The injected fault was caused by swapping an internally used algorithm at runtime to another algorithm that did not have sufficient time complexity. The fault was induced at the white circled location found in Figure 8, because it is one of the most computationally heavy part of the map due to the overlapping lane markers on the road. As seen on Figure 9, the output stream data port of the *Desired Steering Wheel Angle Calculation*

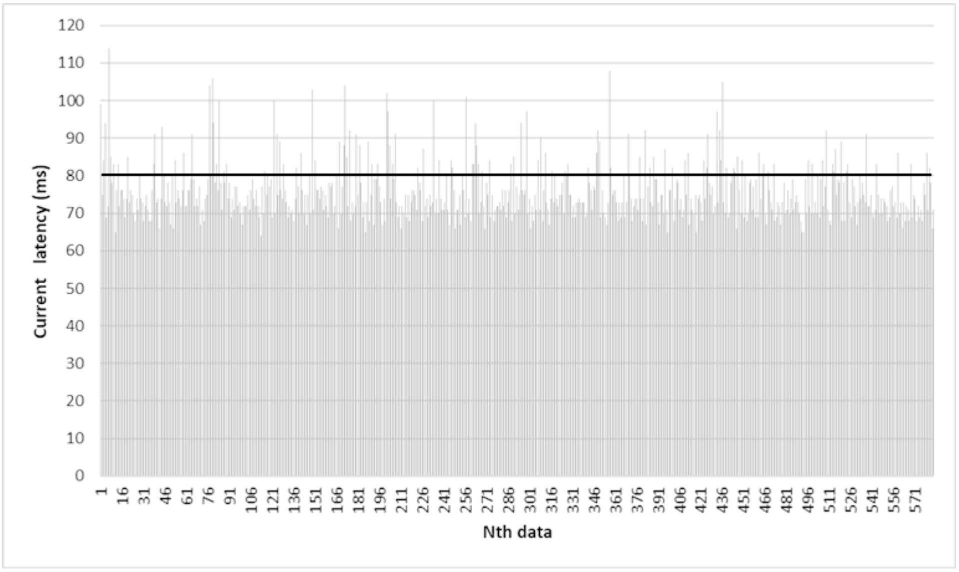

Figure 9. Monitored freshness constraint from output stream data port

processing component detected freshness constraint violations. The result was vehicle veering off its driving lane. To handle this fault, we set the freshness constraint handler of the stream data output port to keep count of the number of violations that has occurred. If a threshold was reached, it signaled the *Speed Error Calculation processing component using an* event port. When the *Speed Error Calculation* processing component receives the signal, it goes into safe driving mode and adjusts the target fuel injection value to decrease the vehicle's speed incrementally. As a result, even with the freshness constraint violation occurring, Splash was able to successfully monitor the violation and invoke a handler to stop a critical failure from happening.

Computer 1 (PreScan)	**Computer 2 (Splash)**
CPU: Intel® Core™ i7-7700 CPU @ 3.60 GHZ	**CPU: Intel® Core™ i7-7700 CPU @4.20GHZ**
Main memory: 64GB	**Main Memory: 63 GB**
OS:Windows 10	**OS: Linux ihawk 3.10.el7.x86_64**
Simulator: PreScan Version 8.5.0	**DDS: Vortex OpenSplice DDS v6.7.18**

Figure 10. Experiment specifications

6. Related Work

There has been many graphical programming frameworks similar to Splash. Popular examples include RTMaps, Simulink, and Ptolemy[8][9][10]. The example frameworks are also based on component based programming that include data-flow process network model as programming interface. The example frameworks are also geared towards providing useful extension that aid developers in testing and examining system requirements.

RTMaps is developed to tackle multisensory challenges that allows engineers to track the flow of data. RTMaps also supports time as a first-class entity meaning, each data will be marked with a time stamp. Similar to Splash, RTMaps can handle freshness and correlation constraints, however it does not support detection for rate constraints. It also does not support in-order delivery for all of its operators but must rely on select few to deliver them in order.

Simulink is developed to support myriad of engineering needs that ranges from simple signal transmission to deep learning support through multiple sensor fusion. Despite its versatile functionalities, what it lacks is its support for real time stream processing. It does not support timing constraint annotation in its operators therefore must rely on user implemented functions. It also does not support data-driven programming, and can rely only on time step driven execution, limiting the possible areas for parallel processing.

Lastly, Ptolemy II is a framework geared towards academic researches and verifying system models. Ptolemy offers varieties of support for imperative programing such as mode changes and exception handling, however Ptolemy II also lacks support for real time stream processing. It does not have support for correlation constraints or rate constraints.

7. Conclusion

This thesis presented the graphical programming framework named Splash geared towards developing stream processing applications for autonomous machines. The thesis introduced the key components of the Splash framework and illustrated each of the framework's language constructs. The thesis especially focused on the language implementation of the Splash runtime library and how it comes together to compose a functionality correct executable source code.

At the end, ACC and LKAS application was developed using Splash. To verify its functional correctness a driving simulator ran a cruising simulation using the Splash application as its control logic. The Splash application was also injected with a fault to test against freshness constraint violations. Splash was able successfully detect the freshness constraint violations and handle the fault by making the vehicle go into a safer driving mode.

For future work, Splash can improve by implementing an inter-thread communication interface between inter-factory component constructs. The interface must stay the same as much as it can in order to provide an effective abstraction.

References

[1] Siegel, Joshua E., et al. "Real-time Deep Neural Networks for internet-enabled arc-fault detection." Engineering Applications of Artificial Intelligence 74 (2018): 35-42.

[2] Cui, Yanling, et al. "Towards Adaptive Sensory Data Fusion for Detecting Highway Traffic Conditions in Real Time." DSFAA. Springer, Cham, 2018.

[3] The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification version 2.3, http://www.omg.org/spec/DDSI-RTPS/2.3, document number ptc/2018-09-3

[4] The Data Distribution Service Version 1.4, https://www.omg.org/spec/DDS/1.4/, document number formal/2015-04-10

[5] An, Kyoungho, et al. "Content-based filtering discovery protocol (CFDP): scalable and efficient OMG DDS discovery protocol." Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. ACM, 2014

[6] http://manpages.ubuntu.com/manpages/bionic/man8/ptpd.8.html

[7] ROS, “Nodelet”, http://wiki.ros.org/nodelet

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

[9] “Simulink,”[Online].Available:

https://www.mathworks.com/help/simulink/index.html.

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong, “Taming heterogeneity - the Ptolemy approach,” Proceedings of the IEEE, 2003

Abstract

오토너머스 머신의 패러다임은 기계 지능의 발전으로 인해 새롭게 변화했다. 머신 인텔리전스를 지원하기 위해 오토너머스 머신은 다양한 센서, 이기종 멀티 코어 프로세서 및 분산 컴퓨팅을 사용하기 시작했으며 이를 효율적으로 활용하기 위해 복잡한 소프트웨어 아키텍처가 필요하게됐다. 새로운 센서 및 컴퓨팅 리소스가 도입됨에 따라 오토너머스 시스템은 실시간 스트림 처리를 지원해야했다. 그러나 소프트웨어 복잡성이 증가함에 따라 개발자가 여러 데이터 스트림을 조정하고 시스템 요구 사항을 충족시키는 것이 어려워지고 있다. 이러한 어려움을 해결하기 위해 우리는 현재 Splash라는 그래픽 프로그래밍 프레임 워크를 개발 중이다.

Splash는 사용자가 여러 스트림 처리 응용 프로그램을 손쉽게 개발 할 수있도록 추상화된 프로그래밍을 제공한다. 또한 Splash는 사용자가 시스템에서 요구하는 end to end timing constraint 을 지원하기 위해 시간적 제약 사항의 탐지와 처리 기능을 제공합니다.

본 학위 논문에서는 Splash 그래픽 프로그래밍 프레임 워크의 구성 요소를 소개하고 Splash가 제공하는 스트림 처리 기능에 중점을 둡니다. 이 논문은 또한 end to end timing constraint 대한 Splash의 모니터링 기능의 내부 동작을 소개합니다. 마지막으로이 논문은 Splash 응용 프로그램의 기능적 정확성을 검증하고 Splash를 사용하여 ACC (Adaptive Cruise Control) 및 LKAS (Lane Keeping Assistance System) 알고리즘을 구현하여 시간적 제약 사항의 모니터링 기능을 검증합니다.

주요어: 스트림 프로세싱, 시간적 제약 사항, 프로그래밍 추상화, 컴포넌트 기반 프로그래밍
학 번: 2017-27804