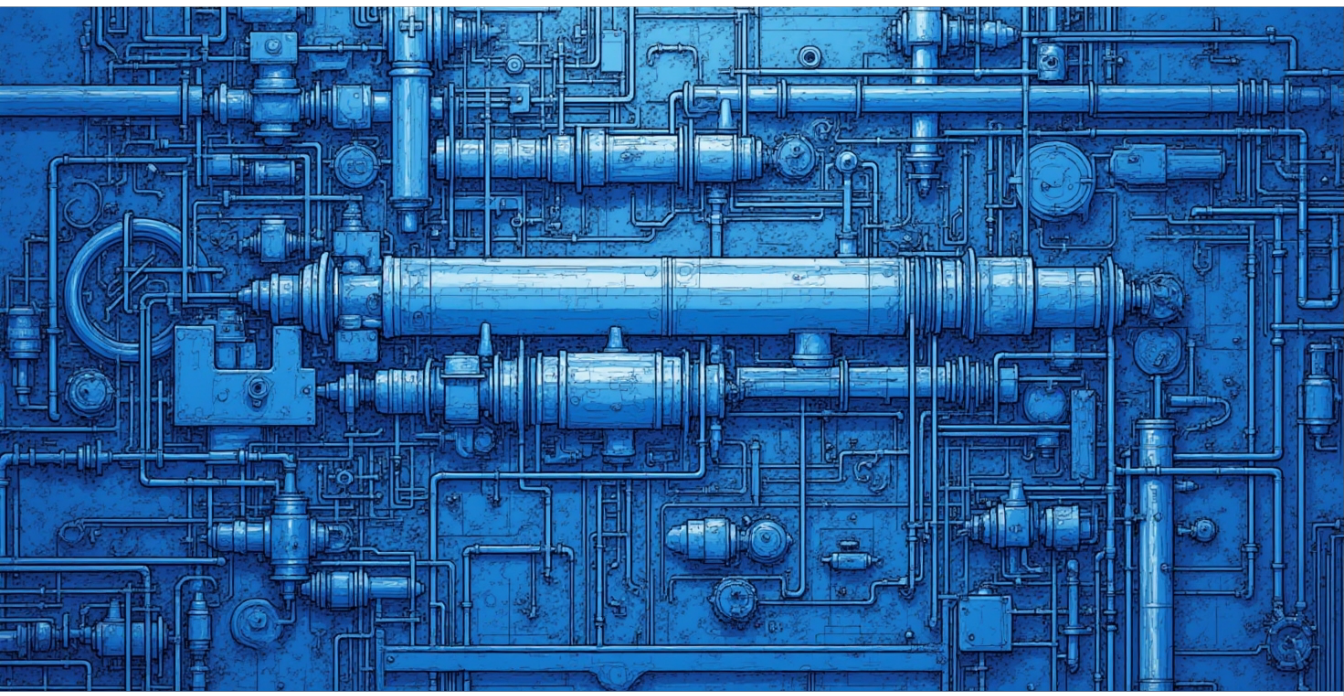
General-Purpose Visual Programming Language

Pipe

Pragmatic Approach to Visual Programming



Feature-rich visual language with highly detailed design all the way down to API level

Designed to supplement and enhance AI code generation

Maximizing productivity of AI-driven software development via visual programming

Oleg P. Kabanov

General-Purpose Visual Programming Language

Pipe

Pragmatic Approach to Visual Programming

Oleg P. Kabanov

For licenses and attributions of all fonts used in this book, please see the end of the book.

ISBN: 978-1-0696277-0-4

First Edition

Published by Oleg Pavlovitch Kabanov

Please feel free to leave your comments or review at https://www.pipelang.com/

Contents

This page is intentionally left blank

1. Introduction

1.1. Introduction

Visual programming makes it easier for people of different backgrounds and levels of expertise to create software. Visual languages differ from text-based (non-visual) languages as they are based on a graphical notation. However, text-based languages such as C# and Java are the most used and popular today. Despite many efforts to create a general-purpose visual language, there isn't any practical and widely used one. Text-based programming languages still dominate as a primary method of software production nowadays.

To address the need in visual programming tools, low-code platforms are gaining popularity these days. They provide a visual method of software construction. However, each of these platforms has its own non-generic graphical notation, which means the problem of defining a common visual language for all low-code platforms is a significant challenge for the software development industry.

AI code generation technologies pushed software development productivity to the next level, and it seems visual languages became irrelevant. However, visual programming can still play a significant role when combined with AI code generation. Even if AI is capable to comprehend complex technical and business specifications, there will always be ambiguities and gaps in prepared specifications for code generation, and number of potential incorrect assumptions made by AI during code generation are going to be increasing almost exponentially as complexity of a project grows. Providing detailed enough specifications for AI is a human responsibility and it is an extremely difficult task. Another words, complete and unambiguous explanation of requirements to AI becomes more and more difficult as complexity of a project increases. The solution of this problem is generating code only for base-level components easily explainable to AI, completing the rest of application via manual programming. However, it defeats the purpose of using AI to eliminate human coding. This is where visual programming can be extremely useful as an alternative to text-based languages, boosting AI-driven software development productivity much further.

The next stage of AI-assisted visual programming is probably going to be a direct generation of visual flowcharts. This will significantly simplify human job of verification and understanding generated logic, also making it very easy to change generated visual workflow manually as a faster and easier alternative to resorting to full code re-generation every time modifications of logic are needed.

All these circumstances point to a need for a practical general-purpose visual programming language. Therefore, this book introduces a new visual language "Pipe". The book consists of two main parts: Chapter 2 "Language Specification" and Chapter 3

"Design Rationale and Future Development". Chapter 2 is a formal specification of the Pipe language. Chapter 3 provides summary of a Pipe language specification, also explaining underlying reasons of the language design decisions and describing new ideas for future development of the language. Example of Pipe diagram for a real business case is provided in the Figure 65.

All non-visual programming language examples are using C++. Definitions of new terms in Chapter 2 are highlighted by **underline bold**. Names of elements from diagrams, figures, tables, and source code fragments are shown by ***bold italic***. If a word or sentence needs to be highlighted for any other reason, then it is shown in **bold**. C++ source code and domain/object specifications (see 2.1.4.1.2 "Domain Specification" and 2.1.4.1.3 "Object Specification") use a `fixed-width` font.

Please use the following link to leave feedback related to this book or about visual language Pipe in general: https://www.pipelang.com/.

1.2. Language Overview

Pipe is a visual programming language where citizen developers create workflows from a set of visual building blocks by combining and connecting them together. Pipe has the following features:

- ✓ **Open visual language.** Developers are not constrained by a fixed set of prebuilt components with limited customizations, and they can create or modify visual components exactly to their requirements and specifications.
- ✓ **General-purpose visual language.** The language contains only general-purpose abstract elements. There are no elements representing narrow domain-specific concepts or notions, making Pipe a truly general-purpose visual language.
- ✓ **Compact but powerful language.** Pipe provides relatively few elements and concepts, but this small element base contains expressive and versatile visual building blocks for implementing a wide variety of algorithm.
- ✓ **Practical visual language.** Pipe does not replace non-visual programming languages but rather complements them, leaving lower-level component development to traditional languages and providing visual methods to combine programmed components into workflows.
- ✓ **API for integration with non-visual languages.** Complete API specification for integration with non-visual languages is an essential part of Pipe language specification. This API can be consumed by any object-oriented non-visual programming language used for integration with Pipe workflows.
- ✓ **Comprehensive and detailed language specification.** Comprehensive and detailed language specification makes building a complete virtual machine for Pipe flowchart execution a straightforward task, as precise and unambiguous description is provided for all language elements.

- ✓ **Statically typed visual language.** Pipe is a statically typed visual language similar to top-tier non-visual programming languages such as Java, C#, C/C++, etc.
- ✓ **Levels of usage.** It is not required to know the complete Pipe specification for software development as multiple levels of usage make it possible to start building Pipe workflows with just a partial skill set.
- ✓ **Integration with AI code generation tools.** Source code produced by AI code generation tools can be placed inside visual components for Pipe integration. Therefore, visual workflow builder can play a role of a composition and integration layer for AI-generated code converted into reusable visual components.
- ✓ **Next generation of low-code platforms.** Pipe usage as integration layer of visual components encapsulating AI-generated code can inspire the next generation of low-code platforms where users are not constrained by prebuilt components with limited customizations anymore as they can generate new components using AI, integrating them via visual language Pipe.
- ✓ **Long-term vision.** While the current version of Pipe language already provides a great number of features, there are lots of new ideas and features planned for future versions of the language.

2. Language Specification

2.1. Concepts

2.1.1. Pipeline

Pipeline is a directed graph connecting pipeline **inputs** with **outputs** directly or via intermediary **components** represented by graph nodes (see Figure 1). Each component may have any number of its own inputs and outputs called collectively **pins**.

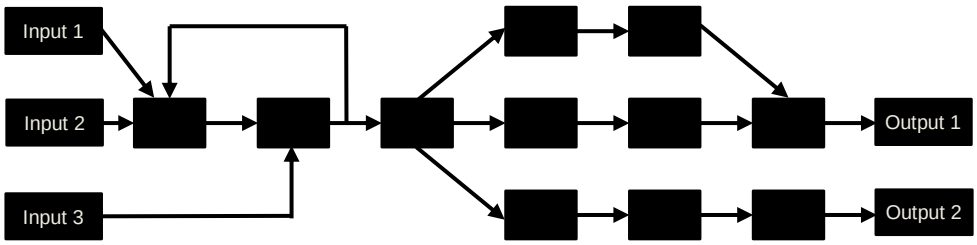


Figure 1. Pipeline as a directed graph.

2.1.2. Components

2.1.2.1. Component Taxonomy

The following top-level components are defined in Pipe language (see Figure 2):

- **Runlet** – data processing unit implementing some functionality.
- **Memlet** – data storage component.

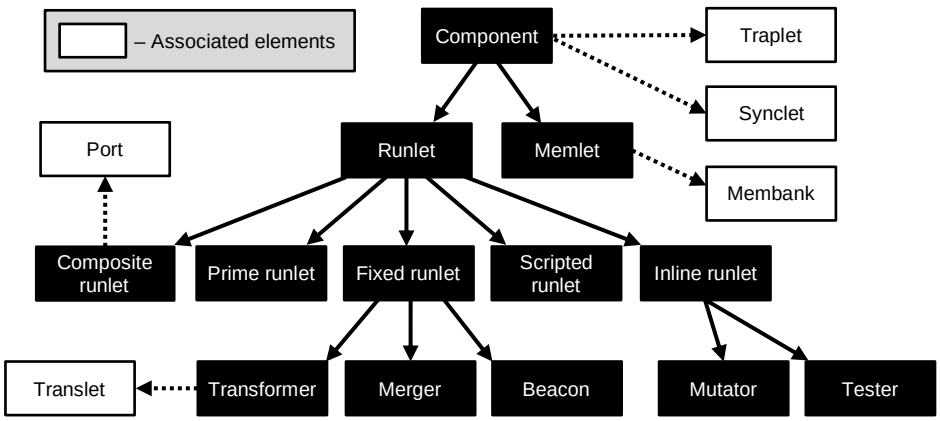


Figure 2. Component taxonomy.

2.1.2.2. Runlets

2.1.2.2.1. Runlet Types

Runlet represents a data processing unit. It may have any number of inputs and outputs (except inline runlets that have a fixed number of inputs and outputs – see "Inline runlet"). The following runlet types are defined in Pipe (see Figure 2):

- Fixed runlets.
- Prime runlets.
- Composite runlets.
- Scripted runlets.
- Inline runlets.

Fixed runlets have predefined behavior as native built-in elements of the language. They are described in 2.1.2.2.2 "Fixed Runlets".

Prime runlets contain executable assets created using any text-based programming language (except scripting programming language Python) called a **prime language**. The source code used to create executable assets inside of a prime runlet is called a **prime source code**.

Prime runlet development would normally happen outside of Pipe development environment because Pipe implementations do not have to provide integrated support of any prime language except allowing import and usage of prime runlets already containing executable assets inside.

Prime runlet is the most difficult type of runlets to create, as it usually requires professional software development skills. However, prime runlets provide many advantages such as better performance and access to a large variety of system and low-level APIs (network, multithreading, etc.).

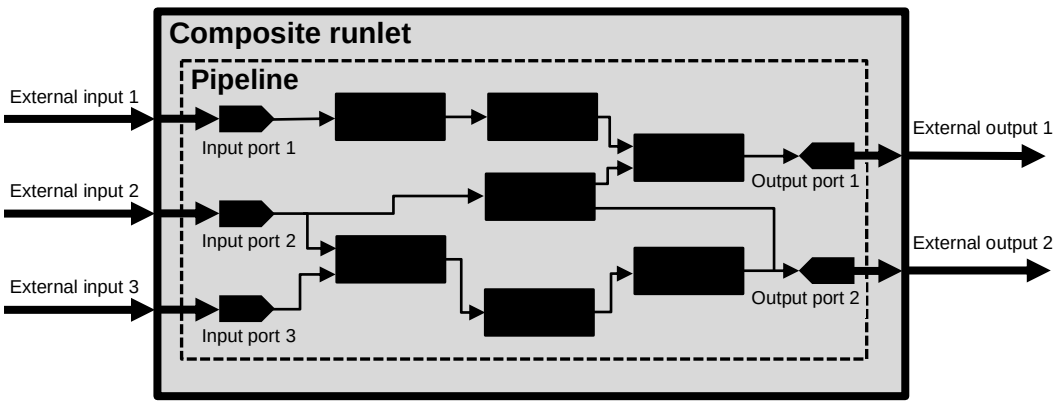


Figure 3. Input and output ports.

Composite runlet contains a pipeline inside. It also contains **input ports** and **output ports** (called collectively **ports**) – connection points inside of a composite runlet for connecting internal components with external inputs and outputs of the composite

runlet. All connections of internal components with external inputs/outputs must be done via ports. Each input port must be connected to exactly one external input, and each output port – to exactly one external output. However, a port may be connected to any number of internal components inside of a composite runlet (see Figure 3).

Scripted runlets are the same as prime runlets, except they are created using scripting programming language Python. Scripted runlets are a middle ground between composite runlets created using a user-friendly visual workflow builder, and much more complex prime runlets requiring professional software development skills. Scripted runlets can be programmed by citizen developers in cases when new composite runlet cannot be created due to absence of required prime runlet building blocks, while creation of a new prime runlets is also out of reach due to the lack of professional software development skills. Python is easy to learn/use and therefore it was included as a part of Pipe for use by advanced citizen developer, giving them an ability to make Python-based scripted runlets which are much easier to program than prime runlets. Python is an essential and integral part of the Pipe language and therefore, Python programming tools must be integrated in each Pipe implementation.

Inline runlet is a component placed at a specific point of a connection and processing data from all signals passing through that point (see Figure 4). Inline runlets are special kind of scripted runlets and therefore, they also contain Python code. There are the following types of inline runlets:

- **Mutator** – inline runlet with one input and one output. Mutator must always produce at least one output signal for each received input signal. Mutator cannot change input data structure, but it may change input data contents.
- **Tester** – inline runlet with one input, and one or two outputs. Tester outputs are called **positive output** and **negative output**. Tester must always produce exactly one signal on one of its outputs for each input signal, but it is allowed not to produce an output signal if the tester has one output (positive or negative), playing the role of a data filter in such case. If tester produces output signal, its data structure and data contents should be identical to data structure/contents of the received input signal. Tester cannot change neither input data structure nor input data contents, but it can read input data contents to direct input signal to the proper output.

Inline runlets represent a simpler alternative to scripted runlets. They are a preferred runlet type for small Python code fragments that can be quickly and easily converted into an inline runlet instead of a more complex process of wrapping the code into a scripted runlet and explicitly assigning domains to all its inputs and outputs (see 2.1.4.1.21.1 “Domain Assignment Scope” and 2.1.4.1.21.2 “Domain Assignment Methods”). Inline runlet is the easiest way to integrate Python code fragments into a pipeline.

Runlet types containing Python or prime code inside – prime, scripted, and inline runlets, are called collectively **coded runlets**.

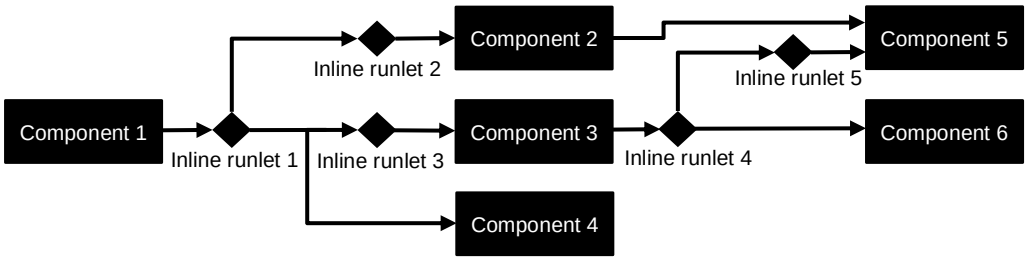


Figure 4. Inline runlets.

2.1.2.2.2. Fixed Runlets

The following fixed runlets are defined in Pipe:

- Merger.
- Transformer.
- Beacon.

Merger is a component for merging multiple data objects together (see 2.1.4.2.1 “Mergers”).

Transformer is a component for transforming structure of a single data object (see 2.1.4.2.2 “Transformers”).

Beacon is a component for generating startup and shutdown signals. It has no inputs and one output. A beacon produces (fires) a signal when an application (see 2.1.2.2.4 “Runlet Tree") containing that beacon is starting up or shutting down. There are the following beacon types:

- **Startup beacon** – beacon firing a signal during application startup.
- **Shutdown beacon** – beacon firing a signal during application shutdown.

The order of firing for regular beacons during application startup or shutdown is not defined. However, there is a special type of beacon called **priority beacon** with an integer number in a range from 0 to +32,767 defining priority of its firing inside of the pipeline (lower number indicates higher priority). If pipeline contains both regular and priority beacons, then regular beacons start firing only after all priority beacons are fired first. Several beacons in a pipeline may have the same priority and the order of their firing is not defined. The order of firing between beacons located in different pipelines is also not defined. No non-beacon signal should start processing anywhere in application before signals from all startup beacons are processed during application startup, and after firing of the first signal from a shutdown beacon during application shutdown.

2.1.2.2.3. Template/Instance Runlets

The same runlet can be inserted into the same and/or different pipelines any number of times by a user. New copy of a runlet (called **instance runlet**) is created every time a new runlet is added to a pipeline during workflow construction. Each newly created instance of a composite runlet automatically gets its own copies of all its own membanks (see 2.1.2.3.1 “Memlets and Membanks”) and all membanks for all its direct and indirect

children. An original runlet used to create instance runlets is called a **template runlet**. It never gets inserted into a pipeline by itself, but it is used as a template for generating identical copies of instance runlets added to pipelines. Exceptions from this rule are inline and fixed runlets that cannot be reused and therefore, they do not have a template runlet, existing only as a single instance at a location where they were placed. To reuse inline or fixed runlet, it must be wrapped into a composite runlet. All runlet types except inline and fixed runlets are template-based and they are collectively called **reusable runlet types**: prime, scripted, and composite runlets. All instance runlets created from the same template runlet share the same **template parameters** with their corresponding template runlet: pin names, domains assigned to pins (see 2.1.4.1 “Domains”) and internal contents (code for prime/scripted runlets, or pipeline graph for composite runlets). Template parameters cannot be changed independently for a specific instance runlet and the only way to change template parameters is by modifying its template runlet with automatic propagation of the change to all instance runlets.

2.1.2.2.4. Runlet Tree

As a composite runlet contain other runlets inside, it forms a tree-like structure with non-composite (prime, scripted, fixed, and inline) runlets as tree leaves, and composite runlets as non-leaf or leaf (in case it does not contain any runlets inside) nodes. A tree containing all pipelines nested within one another, starting from the lowest tree leaves and extending all the way up to the top-most (root) level is called a **runlet tree**. The root of a runlet tree is called a **root runlet** – a runlet generated automatically during creation of a project to hold a top-level pipeline of a runlet tree, or code of a scripted root runlet in case of a purely coded runlet tree. A single runlet tree available for modifications is called a **project**. A set of projects packaged together is called a **solution**. The following project types within a solution are defined in Pipe:

- **Library** – project type whose root runlets are available for reuse in the same solution. Library projects can also be made available for reuse in other solutions after **exporting** such library projects by labeling them as exported. Adding a solution into another solution for reuse is called library **import**. Only root runlets of library projects to be reused in the same or other solutions. However, a composite or scripted non-root **in-place runlet** can be defined anywhere inside of a runlet tree while being used at that location at the same time. In-place runlets can be used only in places where they are defined, and not anywhere else.
- **Application** – project type where one project of such type can be designated as a **default project** meaning its root runlet becomes an entry point of the solution when it is loaded into Pipe virtual machine for execution as an application. Root runlet of a project of application type is also available for reuse in the same solution and in other solutions after labeling such projects as available for reuse, similar to library project type.

The same solution can be used both as an application (runlet tree of default project becomes an entry point of application), or as a library that can be imported into another solution for reuse (all runlets labeled as exported in the library can be reused in other solutions).

Table 1. Examples of application types and their connectivity.

Application type	System input ports	System output ports
Web	HTTP request	HTTP response
Desktop	Input event (mouse, keyboard, etc.).	Flag indicating if the event was processed
Console	Command-line parameters	Termination status code

2.1.2.2.5. External Connectivity

Root runlet of application project type cannot use regular ports inside, but the following **system ports** are available only inside of a root runlet of application project for connectivity with a virtual machine (VM) running Pipe application:

- **System input ports** – ports supplying input signals coming from VM.
- **System output ports** – ports for sending response signals back to VM.

Application connectivity with VM can be established by connecting root runlet inputs with system input ports, and root runlet outputs – with system output ports. **Application type** determines what specific system input/output ports are available for connectivity with root runlet of application project, and data structures for each of them. The list of such application types is outside of the scope of this specification and Pipe implementations are free to define their own application types – examples are provided in Table 1. System inputs and outputs get connected to corresponding automatically synthesized inputs and outputs in case a project of application type is reused.

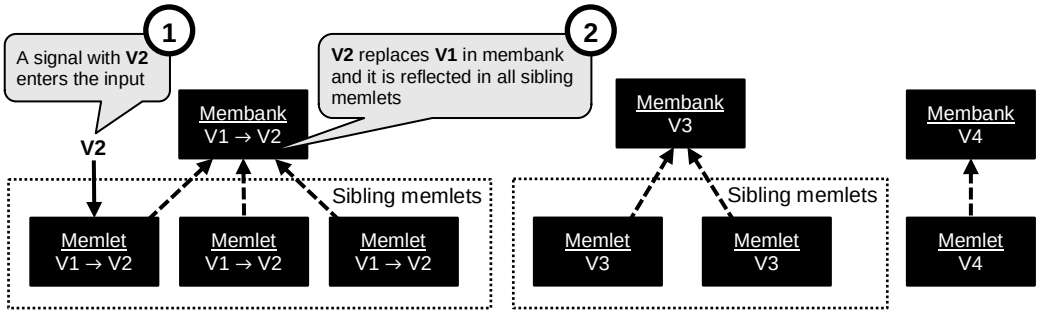


Figure 5. Memlets and membanks.

2.1.2.3. Memlets

2.1.2.3.1. Memlets and Membanks

Memlet is a component for data storage. It has one input and one output. Memlet does not store data by itself, because it is backed by a data-storing element called **membank** which does not have its own inputs/outputs, but it provides shared data storage space for all connected memlets. All membank data operations must be done only via its connected memlets, but the data itself is stored in the membank.

Different memlets of the same pipeline might share the same membank and therefore, different parts of a pipeline can read/write the same data. However, membanks cannot be shared between different pipelines. Memlets sharing the same membank are called **sibling memlets**. Sibling memlets share the same data content as they are connected to the same membank. Any data change in one memlet should be reflected in a connected membank and that change must be instantly propagated in all its other sibling memlets, i.e., memlets must always reflect up-to-date content of their connected membanks (see Figure 5).

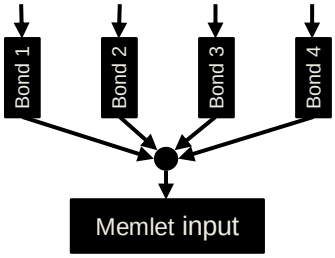


Figure 6. Bond separation

2.1.2.3.2. Memlet Bond Attributes

Bond is a point where connection enters a component pin. Memlet response to an input signal depends on **bond attributes** assigned to a bond delivering signals to that input. Even though memlet has only one input, all connections entering the same input may go through different bonds on their way to the input and therefore, memlet response to signals coming to the same input via different bonds varies depending on attributes assigned to the bonds. This feature is called **bond separation** (see Figure 6). A connection cannot enter the same input via different bonds with different attributes – this rule is valid for input attribute of any type, not just for a bond attribute.

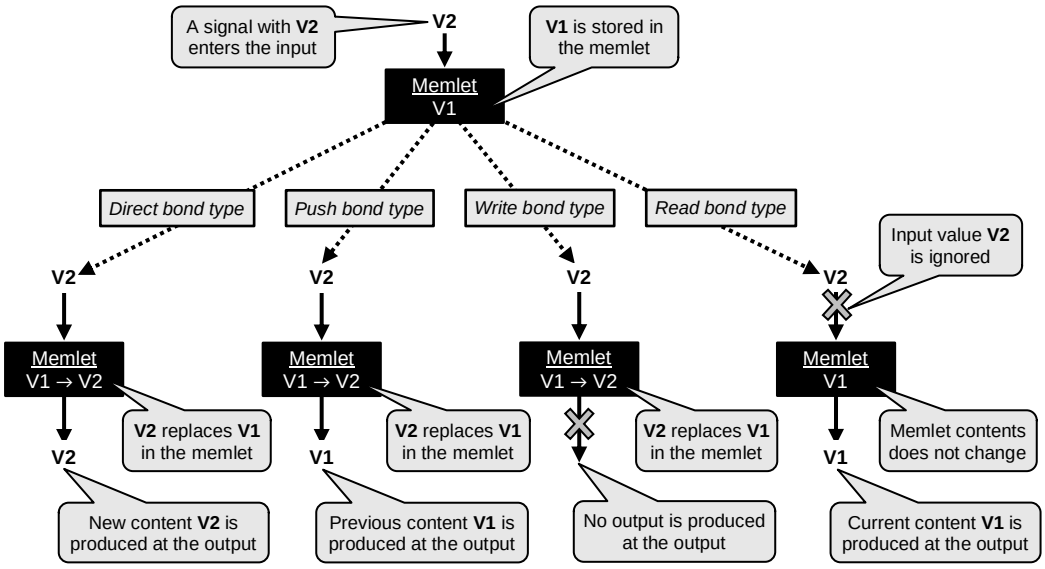


Figure 7. Memlet bond types.

The following memlet bond attributes (see Figure 7 and Table 2) indicating **memlet bond type** are available for memlet input bonds (exactly one of these attributes must be assigned to every memlet input bond to indicate its type):

- **Direct attribute** – value from arrived signal overwrites memlet content, and output is generated with a new content. Direct attribute represents "First write then read" semantics when a value gets written to a memlet and its new content is produced at the output.
- **Push attribute** – value from arrived signal overwrites memlet content, and output is generated with a previous content. Push attribute represents "First read then write" semantics when new value gets written to a memlet and its previous content is produced at the output.
- **Write attribute** – value from arrived signal overwrites memlets content and no output is produced. Write attribute represents "Write-only" semantics when input value gets written to the memlet without producing any output.
- **Read attribute** – value from arrived signal is ignored, and output is generated with the current content. Read attribute represents "Read-only" semantics when input value pushes the current content to the output without writing anything to the memlet.

Example shown in the Figure 47 shows how different memlet bonds work together.

Table 2. Bond type summary.

Bond type	Input action	Output action	Semantics
Direct	Input value overwrites memlet content	New content is produced at the output	First write then read
Push		Previous content is produced at the output	First read then write
Write		No output is produced	Write-only
Read	Input value is ignored	Current content is produced at the output	Read-only

2.1.2.3.3. Broadcast Bond Attribute

A signal entering a memlet produces output only from that memlet and not from any other sibling memlet. However, a memlet input bond can have a **broadcast bond attribute** assigned to it, meaning that a signal entering an input via bond with such attribute triggers output production not only from the memlet receiving the signal, but also from all its sibling memlets (see Figure 8). Such signal is called **broadcast-triggering signal**. A bond with a broadcast attribute triggering output production from all sibling memlets is called a **triggering bond**. A memlet that directly received a signal via triggering bond and invoked response in all siblings memlets is called **triggering memlet**. All other sibling memlets which did not receive any signal directly but produced an output as a response to a signal entered the triggering memlet are called **triggered memlets**. All signals produced as a result of a signal entering memlet input via a bond

with a broadcast attribute (including output signal produced directly from a triggering memlet) are called **broadcast-triggered signals**.

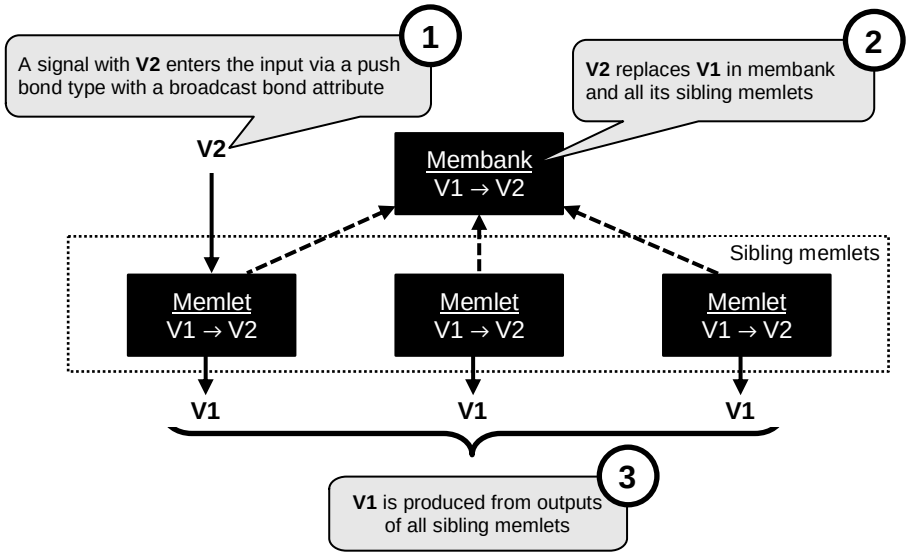


Figure 8. Broadcast bond attribute.

Broadcast bond attribute is optional and applicable only to memlet input bonds. It is compatible with all memlet bond types. Broadcast bond must be combined with exactly one memlet bond type attribute.

All sibling memlets produce the same response according to the bond type of a triggering bond – for example, the previous content of each memlet is produced in case of a signal entering the triggering memlet via a push bond type. Broadcast bond attribute has no effect if used together with a write bond attribute as such combination does not produce any output.

The order of output production is first from the triggering memlet and then for all triggered memlets. The order of output production between triggered memlets is not defined. No new signal arrived after broadcast-triggering signal should start processing in a pipeline until all broadcast-triggered signals are processed first.

2.1.2.3.4. Read-Only Flag

Special read-only flag may be assigned to a memlet or membank. **Read-only memlet** allows strictly read bond type for its input. **Read-only membank** must be connected solely to read-only memlets.

2.1.3. Connectivity

2.1.3.1. Signal

Signal is produced by a component output or by an internal event inside of a component, passing data between components and triggering component execution after entering

input of a component, or after starting to handle an internal event produced inside of a component. A signal may optionally carry an attached **data object** containing data passed between components. A component that received a signal is going to be executed regardless of presence or absence of an attached data object in that signal. A signal with attached data object is called **carrier signal**, and signal without an attached data object – **blank signal**. There are the following signal types:

- **External signal** – signal produced from an output of the same or other component and entered a component input.
- **Internal signal** – signal produced inside of a component by an internal event not related to processing of any input signal. Internal signals are not allowed for root runlet of an application. They are produced in the following cases:
    - **Memlets** – output signals are produced by triggered memlets via a broadcast bond attribute and enqueued at the parent composite runlet to be processed after output signal of triggering memlet is processed.
    - **Prime and scripted runlets** – output signals produced by a prime or scripted runlet and enqueued at the parent composite runlet after an asynchronous operation completes, an external software/hardware event is handled by the code, or in any other asynchronous case where output signal production was not a direct result of an input signal processing.
    - **Composite runlets** – internal signals produced inside of a composite runlet by memlets, prime and scripted runlets during internal signal events described above.
    - **Inline runlets** – internal signals are not allowed for inline runlets.

2.1.3.2. Connection

Connection is a directional relation for signal propagation from M outputs to N inputs, where $M > 0$ and $N > 0$. It is allowed to connect an output and input of the same component together in a loop-like configuration. Initial (starting) points of a connection are called **source points**. Terminal (ending) points of a connection are called **destination points**. Each source point must be connected to a component output, and each destination point – to a component input. Source and destination points are called collectively **endpoints**.

2.1.3.3. Signal Distribution and Processing Rules

Signal distribution and processing (SDP) must follow these rules:

Rule SDP-1. If a signal is produced from a source point of a connection with multiple destinations, then it is delivered to all destination inputs (see Figure 9). Signals entering inputs of different components are processed in parallel, independently from each other. Signals entering input(s) of the same component are processed sequentially one-by-one according to the rule SDP-4.

Rule SDP-2. If several signals are produced from multiple source points at the same time, they are enqueued at the connection and fetched from the queue one-by-one for delivery to destinations: the next signal is fetched from the queue only after the current signal is delivered to all destinations for processing according to the rule SDP-1 (see Figure 10).

Rule SDP-3. Signal processing inside of a component is triggered by an external signal entering the component input, or by an internal signal produced inside of the component. Processing of a signal may produce side effects (such as printing output, changing database, etc.). Zero or more output signals can be generated from any outputs as a result of signal processing.

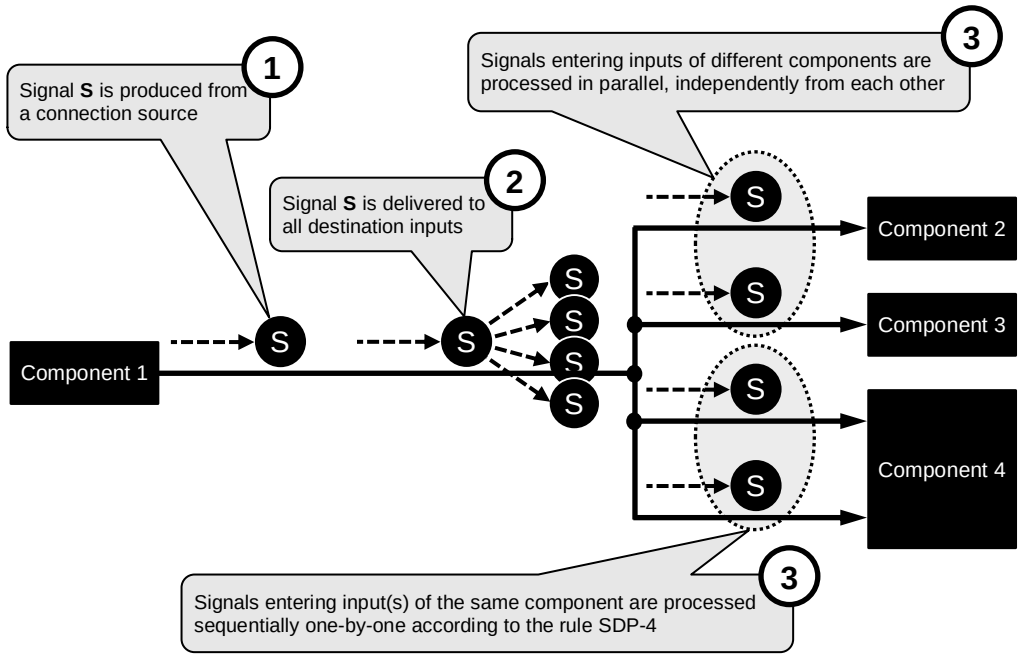


Figure 9. Signal distribution and processing rule SDP-1.

Rule SDP-4. All signals are processed inside of a component one-by-one in the order of arrival. A component may process no more than one input signal at any given moment. Processing of the next input signal starts only after the current signal is fully processed and all output signals are produced. Output signals must become available on external outputs immediately after their production without waiting for completion of input signal processing. Signals waiting at inputs of a merger (see 2.1.4.2.1 “Mergers”) inside of a composite runlet are considered fully processed, thus not blocking fetching the next input signal for processing in that composite runlet.

An exception from the rule SDP-4 is a merger (see 2.1.4.2.1 “Mergers”), which waits for arrival of signals to all its inputs and then starts processing all signals together.

Rule SDP-5. All internal signals produced inside of a component are waiting for processing in the same common queue (called **input queue**) together with the external

input signals arrived at all inputs of the component. However, propagation of internal signal during its processing inside of a component starts from an internal component's output which produced the internal signal before it was enqueued at the parent component, while propagation of external input signal starts from a port connected to a corresponding input the signal has arrived at. An exception from this rule is a merger (see 2.1.4.2.1 "Mergers") that has separate input queues per each input.

Rule SDP-6. Several signals cannot be processed together until their data objects are explicitly merged. No implicit signal merge is allowed. Merger (see 2.1.4.2.1 "Mergers") should be used to merge two or more signals together by combining their data objects.

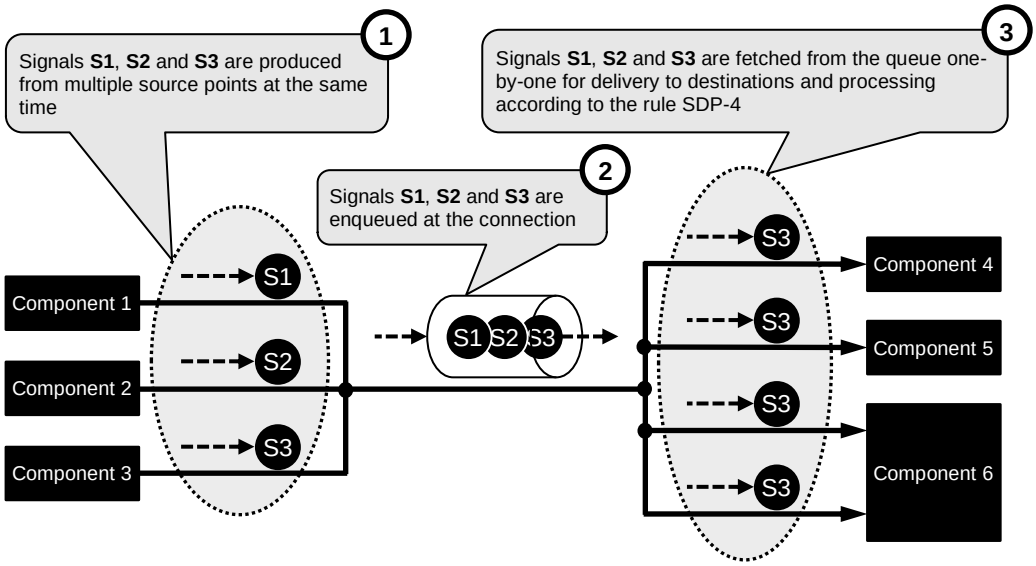


Figure 10. Signal distribution and processing rule SDP-2.

2.1.3.4. Signal Multiplication

One notable side effect of rules SDP-1 and SDP-6 is that splitting one connection into **N** branches and merging them back together will turn one source signal into **N** identical copies of the signal at a destination (see Figure 11).

2.1.3.5. Synchronization

Synchronization is a capability to process incoming signals one-by-one in the order of arrival, when processing of the next input signal starts only after the current input signal is fully processed and all outgoing signals are produced.

Synclet is a convex two-dimensional shape defining boundaries of the synchronization coverage where incoming signals entering the synclet over connections crossing the synclet boundary inward are processed one-by-one in the order of arrival. Processing of the next signal starts only after the current signal inside of the synclet is fully processed and all outgoing signals departing the synclet over connections crossing synclet boundary outward are produced.

Internal signals produced by a component inside of a synclet are enqueued at the boundary line of the synclet in the same common input queue along with all signals coming from outside of the synclet, but propagation of internal signal inside of a synclet during its processing starts from an output of a component that generated the signal before it was enqueued at synclet boundary. Signals waiting at merger inputs inside of a synclet are considered fully processed, thus not blocking fetching of the next input signal for processing in that synclet.

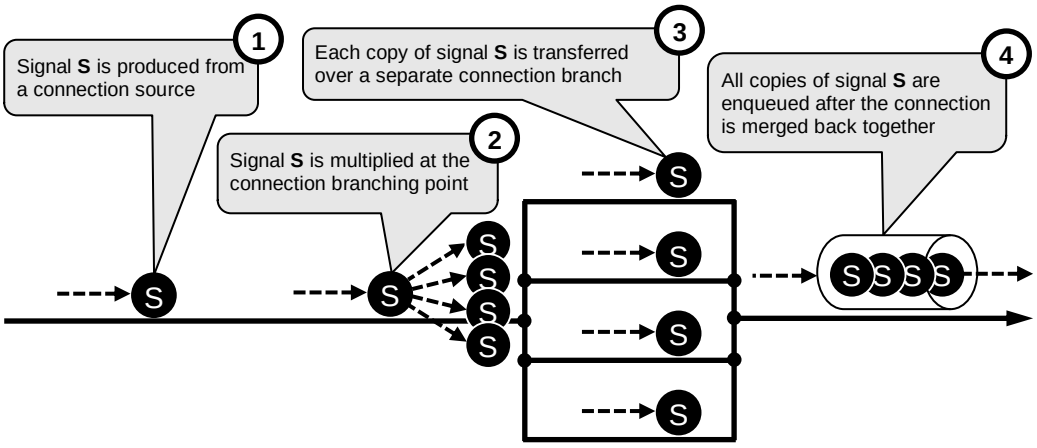


Figure 11. Signal Multiplication.

Another interpretation of a signal processing rule SDP-4 (see 2.1.3.3 “Signal Distribution and Processing”) is that all components (except merger – see 2.1.4.2.1 “Mergers”) must use synchronization for signal processing. To satisfy this requirement, all components except merger have permanently attached **implicit synclet** covering the component area edge to edge. There are cases when a part of a pipeline needs to be synchronized – for example, a loop generating a range of consecutive integer numbers that must not be interrupted by any new incoming signal until sequence generation is completed. To avoid forcing a part of a pipeline to be moved inside of a runlet just to ensure synchronization, a rectangular synclet can be created and placed directly on a pipeline surface, providing synchronization coverage for all components inside of that rectangle. Synclets have the following placement rules:

- No two synclets can intersect partially, but a synclet may contain other synclets inside.
- A component cannot be covered by a synclet partially – it must be completely inside or outside of the synclet.
- All connections crossing synclet boundary inward represent **synclet inputs**, and connection crossing synclet boundary outwards – **synclet outputs**.

If internal signal is generated by a component located inside of a synclet, the signal will be enqueued at the boundary of the synclet (innermost parent synclet relative to the component generated internal signal in case of nested synclets), not at the parent component containing both the synclet and the component generated internal signal.

2.1.4. Data Concepts

2.1.4.1. Domains

2.1.4.1.1. Domain Definition

Domain is a single-root tree data structure called **domain tree**, where tree nodes are called **domain nodes**. Each domain node can contain the following optional **node slices**:

- **Value slice** – value of a certain data type defined for a node.
- **Children slice** – container for a zero or more child nodes.

A node containing value slice is called **storage node**. It might or might not have children slice. Storage node without children slice is called **storage-bound node**. A node without a value slice is called **non-storage node**. It might or might not have children slice.

A node with children slice is called **group node**. It might or might not have a value slice. Group node without a value slice is called **group-bound node**. A node without children slice is called **non-group node**. It might or might not have a value slice. Node containing neither value nor children slice is called **void node**. Node containing both value and child slices is called **complete node**.

Summary of domain node types is provided in Table 3.

Table 3. Domain node types.

Node type	Node has value slice	Node has children slice	Definition
Storage	Yes	Yes/No	A node containing value slice
Storage-bound	Yes	No	Storage node without children slice
Non-storage	No	Yes/No	A node without value slice
Group	Yes/No	Yes	A node containing children slice
Group-bound	No	Yes	Group node without a value slice
Non-group	Yes/No	No	A node without children slice
Void	No	No	Node containing neither value nor children slice
Complete	Yes	Yes	Node containing both value and child slices

Domain containing only a root storage-bound node (i.e., domain containing root node with a value slice and without a children slice) is called **scalar node**.

Each node in the domain tree (except root node) should have a name assigned to it, unique among all its sibling nodes (for node naming rules see 2.1.4.11.3.1 “Naming Rules”). Root node has a reserved name – “at” character (“@”).

Domain path is a sequence of node names in the order of traversal down the tree (from parent to child) starting from an arbitrary start node to some end node, with a slash character (“/”) as a divider between node names. A path from a root node starts with a

name of a root node (“@”). Otherwise, the path must start with a dot character (“.”). For example:

```
@/Account/Balance
./Components/Categories/Names
```

A path is called a **full path** if it starts from a root node. Otherwise, it is called **relative path**. Each domain must have a name that should be unique in the solution. A domain can be labeled as exported for distribution and reuse in other solutions. If an exported domain refers other domains (as a reference domain – see “Domain data type”, or a base domain – see 2.1.4.1.22 “Domain Inheritance”), then referred domains are automatically exported too, but they are not independently available for reuse in a solution they are imported into, unless such domains are explicitly labeled as exported too. All domain structure modifications are allowed only in a solution defining the domain, and they must propagate to all places where the domain is used within a solution.

All domain specifications in this book utilize textual format, not visual notation. This is because Pipe domain are simple, static, and regular tree data structures. Therefore, it is more efficient to use textual representation in case of domains. However, specific Pipe implementations may provide tools for domain structure visualization and for visual domain modifications.

2.1.4.1.2. Domain Specification

Domain specification is a format of a domain structure description as a list of node names (one node per line starting from the root node) in the order of a depth-first tree traversal from parents to children with indentation of child nodes relative to their parent. The first line may optionally contain domain name if it is specified for the domain, or underscore character (“_”) if domain is anonymous (see 2.1.4.11.3.4 “Anonymous Domains”). All storage nodes must have their data types (see 2.1.4.1.4 “Data Types”) shown after arrow character combination (“->”) placed after node name. A comment can also be added starting after a number character (“#”) and running till the end of line. For example:

```
Account                         # Domain specification for a domain “Account”.
@                               # Root node.
   UserInfo                     # User info.
      FirstName -> string       # First name of the user.
      MiddleName -> string      # Middle name of the user.
      LastName -> string        # Last name of the user.
```

Domain node is considered a storage node (i.e., node with a value slice) if data type is specified for that node in domain specification. Domain node is considered a group node (i.e., node with a children slice) if any child is specified for that node in domain specification.

2.1.4.1.3. Object Specification

Object specification is a format of a content description for a specific data object as a list of node names (one node per line starting from the root) in the order of depth-first tree

traversal from parents to children with indentation of child nodes relative to their parent. The first line must contain the name of a corresponding domain, or underscore character ("_") if domain is anonymous (see 2.1.4.11.3.4 "Anonymous Domains"). Comments are allowed with the same syntax as for domain specifications. All nodes of a respective domain must be included in object specification (except missing nodes with optional attribute – see below) and all storage nodes must have their values specified. The value of storage node is shown after equality symbol ("=") placed after node name. For example:

```
User      # Object specification for a domain "User".
@
   Account
      UserInfo
            FirstName = "John"
            MiddleName = ""
            LastName = "Smith"
```

2.1.4.1.4. Data Types

Value slice of a domain node holds some data and therefore, each value slice should have a certain **data type** specifying what kind of data it can accept.

Storage node of any data type may have a special **null value** meaning the node has a value slice, but it does not hold any data in a specific data object. **Null object** is a special case of null value where the whole data object is replaced with null. It indicates null object in place of a data object stored in membank, or null object in place of a data object attached to a carrier (not blank!) signal. Null values and null objects are indicated by a keyword **null** in domain and object specifications. For example:

```
Inventory
@
   Manufacturer = "Nikon"
   Model = "D800E"
   Notes = null         # Null value of the node "Notes".
```

Null value can be replaced with a non-null value via assignment. A different **none value** is used only for non-storage nodes to indicate they cannot hold any data as they do not have a value slice. None value cannot be replaced with any other value via assignment unless domain specification for that node is changed to add value slice.

Each data type has a **default type value** – initial value assigned to a domain node having value slice of that data type in all newly created data objects provided no nullable attribute, no constant value and no default value are assigned to the node (see 2.1.4.1.5 "Node Attributes"). Actual initial value of a node with value slice after applying full combination of attributes is called **default node value** – it might be the same as its default type value or different, depending on assigned attributes.

Pipe defines the following **data type categories**:

- Primitive data types.
- Domain data types.
- Collection data types.

Names of all primitive data types are reserved, and they cannot be used as domain or other names. The list of **primitive data types** is provided in Table 4.

Table 4. Primitive data types.

Name	Value example(s)	Default type value	Description
string	“Hello”	“”	String of any length in double quotes.
bool	True	False	Boolean (true/false) values.
int	25	0	Positive and negative integer numbers.
float	25.47	0.0	Positive and negative real numbers in format IEEE 754 binary64 (double precision, float64)
datetime	|2019-05-10 12:54:13|	|0000-01-01 00:00:00|	Date/time value in format ISO-8601 enclosed in vertical lines (“|”).
binary	\B5A2008F\	\\	Binary data block of arbitrary size. The content of a binary block is shown as pairs of hexadecimal numbers (one pair per byte) enclosed in backslash characters (“\\”).
any	“Hello”, 25, True	\\	Value of any data type. The default type value is an empty binary data block.

Domain data type uses domain (called **reference domain**) as data type, meaning that data must conform to the structure of the reference domain. The default type value of a domain data type is a default data object of the reference domain (see 2.1.4.1.9 “Default Data Object”). The reference domain name is shown in domain specifications enclosed in curly brackets (“{}”). For example:

```
@
   Account
      UserInfo
         PersonName -> {Name}   # Reference domain “Name” of a domain data type.
```

Values of storage nodes with domain data type are shown in object and domain specifications as **domain-expanded** – expanded recursively all the way down to the level where there are no more storage nodes with domain data type to be expanded. Domain name is not shown inside of a domain-expanded block which must be enclosed in curly brackets (“{}”). For example:

```
User
@
 Account
    UserInfo
       PersonName = { # Domain-expanded node of the
       @              # reference domain “Name”
          FirstName = “John”
          MiddleName = “”
          LastName = “Smith”
       }
```

Domain specification for domain “Name”

```
Name
@
   FirstName -> string
   MiddleName -> string
   LastName -> string
```

Storage node with domain data type can participate in a **crossover path** – a domain path continuing below a domain data type node with nodes from the reference domain. First part of a path with nodes from a main (parent) domain and second its part with nodes of a reference domain must be divided by a path element with a root name (“@”), i.e., root node name is used inside of a path to indicate where a node of a domain data type ends and reference domain starts. For example:

```
@/Address/@/StreetName  # @/Address – parent domain, @/StreetName – reference domain.
```

Collection data type is an array of elements of the same data type (called **base type**). Any data type can be used as a base type for the array, including another collection data type (representing nested arrays). The default type value of a collection data type is an empty array. Collection data types are shown in domain specifications as a base data type name in square brackets (“[]”). If base type for a collection is a domain data type (called **base domain type**), then the name of the base domain type should be enclosed in curly brackets (“{}”) before being enclosed in square brackets (“[]”). For example:

```
Article
@
   Authors -> [string]          # String base type of a collection data type.
   Title -> string
   Subtitle -> string
   Body -> string
   References -> [{Reference}]  # Domain base type of a collection data type.
```

The structure of the base domain type for a collection data type can be defined in-place in domain specifications by enclosing base domain type definition in curly brackets (“{}”) and then in square brackets (“[]”). For example:

```
Order
@
   CustomerInfo -> {Customer}
   Items -> [{                  # Base domain type for the collection defined in-place.
   @
      ProductInfo -> {Product}
      Quantity -> int
      Price -> float
   }]
```

Values of a collection with domain base type are shown in object and domain specifications enclosed in square brackets (“[]”), and with each value shown domain-expanded and enclosed in curly brackets (“{}”) with comma character (“,”) as divider between domain-expanded values. For example:

```
Property
@
   NumberOfRooms = 2
   RoomDimensions = [ # Domain-expanded values of a domain base type of a collection.
      {
      @
         Length = 12
         Width = 12
      },
      {
      @
         Length = 19
         Width = 11
      }
   ]
```

Values of a primitive base type collection are shown in object and domain specifications enclosed in square brackets (“[]”) with element values divided by comma character (“,”). For example:

```
Tournament
@
   Country = “USA”
   City = “Phoenix”
   PlayingCountries = [“USA”, “France”, “UK”]   # Values of primitive base type.
```

2.1.4.1.5. Node Attributes

The following **node attributes** can be assigned to a domain node:

- Default attribute.
- Constant attribute.
- Nullable attribute.
- Optional attribute.
- Mandatory attribute.
- Forbidden attribute.
- Injection attribute.
- Rejection attribute.

All assigned attributes must be shown in domain and object specifications, except default values (see Default attribute) that should not be shown in object specifications, unless default value is an actual node value in a specific data object.

Default attribute specifies default node value assigned to a storage node in all newly created data objects instead of a default type value of a corresponding data type. Optionally, default attribute can be indicated as a capital letter "D" in parentheses after node name. This attribute can be assigned to storage nodes of any data type, but actual values must be specified directly only for storage nodes with primitive data types or primitive base type collections. Default value of a storage node with a domain data type must be domain-expanded first, and then the default values must be specified for all storage nodes of a primitive data types or primitive base type collections shown for the domain after its full expansion. The default value is shown after equality character ("=") placed after node type name. Default value must be a single literal, i.e., no expressions are allowed. For example:

```
AccountCls
@
   Code -> int
   Name -> string
```

Domain specification for domain "AccountCls"

```
User
@
   Account
      AccountInfo
         Status(D) -> string = "Open"
         Class -> {AccountCls} = { # Domain-expanded default value for node "Class"
         @
            Code = 100
            Name = "Regular Account"
         }
         Categories -> [string] = ["User", "Client"] # Default values of collection.
```

Constant attribute is the same as default attribute, except the node value never changes: all input values pushed into the node are ignored and the same constant value is produced by such nodes as a response to any input signal. The constant attribute is indicated as a capital letter "C" in parentheses after node name. An optional constant value is shown after equality character ("=") placed after node type name. If no constant value is specified for a storage node explicitly, then it is assumed to be the default type

value of a corresponding data type, or null value if a nullable attribute is specified for a node (see Nullable attribute). For example:

```
User
@
   Account
      UserInfo
         Name
            FirstNameI -> string = “John” # Constant value for this node is “John”.
            MiddleNameI -> string      # Constant value for this node is “”.
            LastNameI(N) -> string     # Constant value for this node is null.
      AccountType -> string
      Status -> string
```

Nullable attribute indicates that the storage node may contain null value. The default node value of a storage node with nullable attribute is a null value unless non-null default or constant value is specified for the node explicitly. Nullable attribute is shown as a capital letter “N” in parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         Name
            FirstName -> string
            MiddleName(N) -> string    # Nullable attribute for a node “MiddleName”.
            LastName -> string
      AccountType -> string
      Status -> string
```

Storage node of a domain data type must be nullable with null default node value to be able to use its direct or indirect parent domain as a reference domain, except cases of a base domain type for a collection. For example:

```
Employee  # Direct parent container domain for “Manager”, “CoWorker” and “HrReps” nodes.
@
   Manager(N) -> {Employee}     # Allowed: “Employee” is a parent container domain,
                                # and node “Manager” is nullable.
   CoWorker -> {Employee}       # Not allowed: “Employee” is a parent container domain
                                # so node “CoWorker” must be nullable, but it is not.
   HrReps -> [Employee]         # Not nullable is fine here as this is a collection.
```

Optional attribute indicates that the node may be absent from a data object. If node with an optional attribute is missing from a data object, then it is going to be automatically created when first value is assigned to that node, or first child is added to the node. If all nodes are missing in an outgoing data object due to optional attribute assigned to them all, then null object is produced from a component output. Please note that optional node is not the same as a storage-bound node with null value. Optional attribute is shown as a capital letter “O” in parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         FirstName -> string
         MiddleName(O) -> string       # Optional attribute for a node “MiddleName”.
         LastName -> string
      AccountType -> string
      Status -> string
```

Mandatory attribute indicates that the node must always participate in any overlap (see 2.1.4.1.11 “Overlap”). Mandatory attribute is shown as a capital letter “M” in

parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         Name
            FirstName -> string
            MiddleName(M) -> string     # Mandatory attribute for a node “MiddleName”.
            LastName -> string
      AccountType -> string
      Status -> string
```

Forbidden attribute indicates that the node must never participate in any overlap (see 2.1.4.1.11 “Overlap”). Forbidden attribute is shown as a capital letter “F” in parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         Name
            FirstName -> string
            MiddleName(F) -> string     # Forbidden attribute for a node “MiddleName”.
            LastName -> string
      AccountType -> string
      Status -> string
```

Injection attribute indicates that the node will be automatically included in every overlap (see 2.1.4.1.11 “Overlap”). Injection attribute is shown as a capital letter “I” in parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         FirstName -> string
         MiddleName(I) -> string        # Injection attribute for a node “MiddleName”.
         LastName -> string
   AccountType -> string
   Status -> string
```

Rejection attribute indicates that the node will be automatically excluded from any overlap (see 2.1.4.1.11 “Overlap”). Rejection attribute is shown as a capital letter “R” in parentheses after node name. For example:

```
User
@
   Account
      UserInfo
         FirstName -> string
         MiddleNameI -> string  # Rejection attribute for a node “MiddleName”.
         LastName -> string
   AccountType -> string
   Status -> string
```

2.1.4.1.6. Attribute Grouping

Node attributes may participate in **attribute grouping** – an attribute flag making the attribute applicable not only to a specific node, but also to all its direct and indirect child nodes. Attribute grouping is indicated by plus character (“+”) after attribute name in domain specification. Explicit default attribute indication (capital letter “D”) should be used in case of attribute grouping applied to a default attribute.

Attributes inherited by child nodes implicitly by descent from a parent node via attribute grouping should not be shown for that node in domain specifications – only directly assigned attributes should be shown.

Explicitly specified default or constant value for a node with attribute grouping will also be applied to all direct and indirect storage child nodes, provided all impacted storage nodes have data types compatible with the data type of the inherited default or constant value. Constant and default values assigned to a node by descent should not be shown for child nodes. For example:

```
User
@
   Account(N+)  # Nullable attribute applied to all child nodes.
      UserInfo  # Nullable attribute received by descent from parent node is not shown.
      Name(D+) = “None” # Explicit default value assigned to a group node is applied
                        # to all child nodes.
            FirstName -> string  # Nullable and default attributes received by descent
            MiddleName -> string # should not be shown in any child node.
            LastName -> string
      Address(M+)               # Mandatory attribute is applied to all child nodes.
         UnitNo -> int          # Mandatory and nullable attributes should not be
         StreetNo -> int        # shown in any child node.
         StreetName -> string
```

2.1.4.1.7. Multiple Node Attributes

A node may have multiple attributes assigned to it. There are two possible ways for a node to have multiple attributes:

- **Direct multiple assignment** – multiple attributes are assigned to the node directly.
- **Indirect multiple assignment** – one or more attributes are assigned to the node indirectly by descent from parent node via an attribute grouping feature.

Only directly assigned multiple attributes must be shown for a node. They must be shown after node name next to each other in separate pairs of parentheses. For example:

```
User
@
   Account
      UserInfo(N+)(M+)  # Nullable and mandatory attributes are assigned to the node.
         Name(I)(O+)    # Injection and optional attributes are assigned to the node.
            FirstName -> string  # None of attributes N, M or O received by descent
            MiddleName -> string # should be shown in any child node.
            LastName -> string
      AccountType -> string
      Status -> string
```

2.1.4.1.8. Attribute Priorities

Some pairs of attributes have opposite effect if applied to the same node, so one of these attributes must override another. The following **attribute priorities** are defined in Pipe:

- Injection attribute overrides rejection attribute.
- Mandatory attribute overrides forbidden attribute.
- Optional attribute overrides constant and default attributes.

2.1.4.1.9. Default Data Object

Default data object is a newly created data object with none of its nodes changed yet. Each storage node of a default data object has value assigned according to these rules:

- **If an optional attribute is defined for a node**, then the node is not present in data object.
- **Otherwise, if a default or constant attribute is defined for the node**, then the specified constant or default value is assigned to the node. Default node value is assigned to the node if no explicit value is specified for the node with a constant or default attribute.
- **Otherwise, if nullable attribute is defined for the node**, then the null value is assigned to the node.
- **Otherwise**, the default type value of a corresponding data type is assigned to the node.

2.1.4.1.10. Domain Assignment Types

Domains can be assigned to membanks and component pins (see 2.1.4.1.21 “Domain Assignment”). There are two types of domain assignment:

- **Nullable** – null objects are allowed for a membank or a component pin:
  - Component pin with nullable assignment can accept null objects.
  - Membank with nullable assignment can store null objects.
  - Initial value of a membank with nullable assignment is a null object.
- **Non-nullable** – null objects are not allowed for a membank or component pin:
  - Component pin with non-nullable assignment cannot accept null objects.
  - Membank with non-nullable assignment cannot store null objects.
  - Initial value of a membank with non-nullable assignment is a default data object.

Assignment type is an attribute of a specific membank or component pin, and it is not part of a domain specification. Nullable assignment is shown in object specifications as a capital letter “N” in parentheses after domain name. For example:

```
Invoice(N) # Nullable assignment to object of domain “Invoice” in object specification.
null       # Null object.

_(N)       # Nullable assignment to an object of a domain without name.
@          # Non-null object, but it can be null as assignment is nullable.
   Data
      Value -> string
```

If domain has a root node with an optional attribute, then all component pins and membanks with such domain must have nullable domain assignment as missing root node is equivalent to null object.

2.1.4.1.11. Overlap

Component inputs and outputs have domains assigned to them (see 2.1.4.1.21 “Domain Assignment”). A signal transfers data from output to input only between pairs of domain nodes that have the same domain path in both output domain (called **source domain**) and input domain (called **destination domain**). A pair of nodes (one from source domain and other – from destination domain) having the same full path are called **twin nodes**. A node that has a twin node is called **cross node**. The process of producing a list of all twin nodes for a pair of a source and destination domains of a connection is called **domain intersection**. This list represents a new domain called **overlap** produced as a result of intersecting the source and destination domains. Overlap is a domain of a data object transporting data from source to destination point. Overlap defines a subset of nodes from a source domain that have twin nodes in a destination domain, and only values for those nodes will be transmitted from a source to destination point. Neither source nor destination domain get changed as a result of an overlap. The following algorithm is used for transferring data from a source to destination domain using overlap:

- All source and destination domain nodes with rejection attribute are excluded.
- Overlap is calculated.
- All source and destination domain nodes with injection attribute and not present in overlap are added to the overlap with optional attribute assigned to them.
- Values are transferred between each pair of twin nodes from source to destination domain, except the following cases:
    - If there is a pair of twin nodes where one of them is a non-storage node and other is storage node, then the value does not get transferred from a source to destination point. If storage node is on destination side, then it takes a default node value.
    - If a node in a source or destination domain for a pair of twin nodes has an optional attribute for any of them and value is missing in one or both sides, then then data is not transferred. If the node is missing only in a source domain, then the destination domain node takes a default node value.
    - If destination domain node has a constant attribute, then it is not filled with value from a twin source domain node, thus keeping its constant value intact.
- Nodes of a destination domain without corresponding twin nodes in a source domain (i.e., destination domain nodes outside of an overlap) take default node values.

- If a source and destination domains do not have any twin nodes (i.e., their overlap is empty), then blank signal travels from the source to destination point and all storage nodes of the destination domain take their default node values.

An example of overlap is provided in Table 5. Overlap concept allows connections between component inputs and outputs with different domains, providing flexibility of workflow composition and type safety at the same time.

Table 5. Overlap.

Source domain (nodes participating in overlap are shown in **bold**)	Destination domain (nodes participating in overlap are shown in **bold**)
User @ **Account** **UserInfo** **Name** FirstNameI -> string **LastName -> string** **Address -> {AddressInfo}** Email -> string **Phone -> string** Contacts -> {ContactInfo}	Customer @ **Account** **UserInfo** **Name** FirstName -> string MiddleName -> string **LastName -> string** **Address -> {AddressInfo}** **Phone -> string** WorkHistory -> {EmploymentHistory}
Twin nodes excluding node *"@/Account/UserInfo/Name/FirstName"* with rejection attribute	
@/Account/UserInfo/Name/LastName @/Account/UserInfo/Address @/Account/UserInfo/Phone	
Overlap	
@ Account UserInfo Name LastName -> string Address -> {AddressInfo} Phone -> string	

2.1.4.1.12. Data Object Transfer Algorithm

The following algorithm is used to transfer non-null data object from a source to destination point after an overlap is calculated for a connection:

1. **This step gets executed at the source before sending overlap data object to destination:**
   - **1.1.** New overlap domain data object (called overlap data object below) is created.
     - **1.1.1.** Each overlap node takes data type from a twin source domain node present in data object.

1.1.2. Each overlap node inherits only nullable attribute from a twin source domain node if nullable attribute is present for the node in a source domain.

1.2. All nodes of the overlap data object created in **1.1** are populated with values from the twin source domain nodes.

1.3. Overlap data object populated in **1.2** is transmitted to a destination.

2. **This step gets executed at the destination after arrival of an overlap data object:**

2.1. New destination domain data object (called destination data object below) is created.

2.2. Values from all nodes of the arrived overlap data object are transferred into the twin nodes of the destination data object created in **2.1**, in accordance with overlap data transfer rules covering nodes with optional and constant attributes, pairs of storage and non-storage nodes, etc.

2.3. Destination data object populated in **2.2** is submitted to the destination.

2.1.4.1.13. Overlap Validation

There are certain conditions that must be met to be able to make a connection between input and output. If these conditions are met, then such pair of input and output is called **connectable**. One of the conditions is that output domain must be **convertible** to input domain. No connections must be allowed between inputs and outputs that are not connectable.

An output (connected to a source point) and input (connected to a destination point) produce a **valid overlap** if the following conditions are met (Conditions 1-5 can also be used to test if a source domain is convertible to destination domain in general, irrelevant of a domain assignment to specific input and output):

- **Condition 1.** All storage cross nodes have data types in source domain convertible to data types of their twin nodes in destination domain.
- **Condition 2.** All nodes with applicable mandatory (forbidden) attribute are (not) present in overlap.
- **Condition 3.** All cross nodes with a nullable attribute in source domain correspond to twin nodes with a nullable attribute in destination domain.
- **Condition 4.** All cross nodes with optional attribute in source domain correspond to twin nodes without an applicable mandatory attribute in destination domain.
- **Condition 5.** All cross nodes with applicable injection attribute correspond to twin nodes without applicable forbidden rejection attribute.
- **Condition 6.** If source point has a nullable domain assignment, then destination point must also have a nullable domain assignment.

Input and output are connectable only if their overlap is valid. Overlap validation is performed interactively during workflow construction. An application must begin execution only when overlaps for all its connections are valid. If input and output are not connectable, a transformer (see 2.1.4.2.2 “Transformers”) can be placed between them with source domain modifications to make it convertible to destination domain. Transformer can fix violations of conditions 1-5, but violation of condition 6 can only be fixed by changing domain assignment type of a source or destination point.

Condition 1. All storage cross nodes have data types in source domain convertible to data types of their twin nodes in destination domain. Each storage cross node in source domain must have a data type convertible to a data type of a corresponding twin storage node in destination domain. This rule is applicable only for a pair of twin storage nodes. Convertibility rules for different pairs of data type categories are provided in Table 6 and convertibility rules for primitive data types – in Table 7.

Condition 2. All nodes with applicable mandatory (forbidden) attribute are (not) present in overlap. This condition requires for all nodes of a domain labeled as mandatory (forbidden) in domain specification where such attribute is applicable according to a modifier, (not) to participate in the overlap. For example, the mandatory node ***Address*** (see Table 8) from the destination domain should participate in an overlap according to the destination attribute modifier but it does not and therefore, the source domain ***User*** is not convertible to the destination domain ***Customer***.

Table 6. Convertibility rules for different combinations of data type categories.

Source domain data type category	Destination domain data type category	Convertibility condition
Primitive	Primitive	Source primitive data type is convertible to the destination primitive data type (see Table 7 for convertability between primitive data types).
Primitive	Domain	Source primitive data type is convertible to the data type of the root node of destination domain (the value goes into the root node of the destination domain).
Primitive	Collection	Source domain data type is convertible to destination collection base type (primitive source value becomes the only element of the destination collection).
Domain	Primitive	Data type of the root node of the source domain is convertible to the destination primitive data type (the value is taken from the root node of the source domain).
Domain	Domain	Source domain is convertible to the destination domain (i.e., their overlap is valid).

Source domain data type category	Destination domain data type category	Convertibility condition
Domain	Collection	Source domain is convertible to destination collection base type (domain source value becomes the only element of the destination collection).
Collection	Primitive	Not convertible.
Collection	Domain	Not convertible.
Collection	Collection	Base data type of a source collection is convertible to the base data type of a destination collection.

Condition 3. All cross nodes with a nullable attribute in source domain correspond to twin nodes with a nullable attribute in destination domain. If a cross node has a nullable attribute in a source domain, then its twin node in destination domain must also have nullable attribute. For example, the node ***FirstName*** (see Table 8) has nullable attribute in a source domain but not in destination domain.

Table 7. Convertible primitive data types.

Source data type	Convertible to the following destination data types
string	string, any
bool	bool, int, float, string, any
int	int, float, string, any
float	float, string, any
datetime	datetime, int, float (Epoch time is used to represent **datetime** as **int** or **float**), any
binary	binary, any
any	any

Condition 4. All cross nodes with optional attribute in source domain correspond to twin nodes without an applicable mandatory attribute in destination domain. If a cross node has an optional attribute in a source domain, then its twin node in a destination domain should not have an applicable mandatory attribute. For example, node ***MiddleName*** (see Table 8) has optional attribute in source domain, but it also has an applicable mandatory attribute in destination domain.

Condition 5. All cross nodes with applicable injection attribute correspond to twin nodes without applicable rejection attribute. If a cross node has applicable injection attribute on one side of a connection, then it cannot have an applicable rejection attribute on the other side. For example, node ***LastName*** (see Table 8) has an injection attribute in source domain and rejection attribute in destination domain.

Table 8. Violations of a valid overlap conditions.

Source domain	Destination domain

`User(N)`
`@`
`  Account`
`  UserInfo`
`  Name`
`  FirstName(N) -> string`
`  MiddleName(O) -> string`
`  LastName(I) -> string`
`  Employment -> {EmploymentRec}`
	`Customer`
	`@`
	`  Account`
	`  UserInfo`
	`  Name`
	`  FirstName -> string`
	`  MiddleName(M) -> string`
	`  LastName(R) -> string`
	`  Address(>M) -> {AddressInfo}`

Paths for twin nodes	

`@/Account/UserInfo/Name/FirstName`
`@/Account/UserInfo/Name/MiddleName`
`@/Account/UserInfo/Name/LastName`	

Destination domain node with applicable mandatory attribute is not present in the overlap (violation of the Condition 2)	
`@/Account/UserInfo/Address`	
Source domain node with a nullable attribute has a twin destination domain node without a nullable attribute (violation of the Condition 3)	
`@/Account/UserInfo/Name/FirstName`	
Source domain node with an optional attribute has a twin destination domain node with a mandatory attribute (violation of the Condition 4)	
`@/Account/UserInfo/Name/MiddleName`	
Source domain node with an injection attribute has a twin destination domain node with a rejection attribute (violation of the Condition 5)	
`@/Account/UserInfo/Name/LastName`	
Source domain has a nullable assignment while destination domain does not have a nullable assignment (violation of the Condition 6)	
`User(N) -> Customer`	

Condition 6. If source point has a nullable domain assignment, then destination point must also have a nullable domain assignment. If source point has a nullable domain assignment, then destination point must also have nullable assignments. For example, source domain ***User*** (see Table 8) has a nullable assignment, while destination domain ***Customer*** does not.

2.1.4.1.14. Domain Merge

Domain merge is an operation used in the following cases:

- Merging source domains for calculating multipoint overlap (see 2.1.4.1.15 “Multipoint Overlap Calculation Algorithm”).
- Combining data objects inside of a merger (see 2.1.4.2.1 “Mergers”).
- Implicit merge of input domains in default domain assignment method (see 2.1.4.1.21.2 “Domain Assignment Methods”).

- Domain merge during multiple inheritance (see 2.1.4.1.22 “Domain Inheritance”).

Unlike overlap, domain merge creates a new domain based on structures of two input domains. The following algorithm is used for creating a new merged domain:

- All nodes from the first source domain without twin nodes in the second source domain and all nodes from the second source domain without twin nodes in first source domain (i.e., all nodes outside of an overlap between two source domains, or non-cross nodes) are added to the merged domain without any modifications, except optional attribute added to them in the merged domain.
- All cross nodes from both source domains are processed using the following algorithm:
    - If both source nodes have primitive data types, then the data type with a higher rank goes into the merged node (see Table 9). If both primitive data types have the same rank, then the rank going into the merged node is one step higher, or the same rank if it is already the highest rank.
    - If both source nodes have domain data types, then their domains get merged using this algorithm.
    - If both source nodes have different data type categories (primitive/domain, primitive/collection, or domain/collection), then the merged node gets data type **any**.
    - If both source nodes have collection data types with different ranks (number of dimensions), then the merged node gets collection data type with the lower rank among two and base data type **any**. If collections have the same rank, then the resulting data type is also a collection with a base data type produced as a result of base data type merge from both source collections (data type merge rules used in this case are specified above for different data type categories).
    - Attributes are merged according to the following rules:
        - All node attributes from both source domains are added together in a target node of a merged domain.
        - Any explicitly defined constant or default values defined for one or both cross nodes are getting removed, resetting them to default node values. An exception is the same constant or default value in both cross nodes – explicitly defined common value is not getting removed in this case.
    - If both source domains use inheritance (see 2.1.4.1.22 “Domain Inheritance”), then their base domain lists and inheritance overrides are merging.

Table 9. Data type ranking for domain merge.

Rank	Data type	Notes
0	none	This type represents case of an absent value in a non-storage node.
1	datetime	
1	bool	
2	int	
3	float	
4	string	
5	binary	
6	any	

2.1.4.1.15. Multipoint Overlap Calculation Algorithm

The following algorithm is used to calculate overlap of a multipoint connection containing ***M*** source points and ***N*** destination points:

- Source domains of all ***M*** source points are getting merged into one domain.
- Resulted merged domain is intersected separately with domains of each ***N*** destination points. If any of ***N*** resulting overlaps is not valid, then the whole multipoint overlap becomes invalid.

2.1.4.1.16. Domainless Component Pins

It is not required for a component pin to have an assigned domain. Component pins without domains assigned to them are called **domainless**.

Domainless output produces blank signals. Such output creates valid overlap with any domain not containing nodes with applicable mandatory attribute. Domainless input creates valid overlap with any domain, stripping data objects from incoming signals. Memlet inputs with a read bond type are always domainless (as read bond type does not write data to memlet) and therefore, output with any domain is allowed to be connected to a memlet input via read bond type. Beacon outputs are also always domainless. Domainless targets are indicated in comments as a pair of curly empty brackets (“{}”). Please note that carrier signal with attached null object is not the same as a blank signal, because the carrier signal is still associated with a domain while blank signal is not.

2.1.4.1.17. Mandatory vs Injection Attribute

Injection and mandatory attributes seem to be playing almost identical roles – forcing domain node to participate in overlap. Such similarity is also seeming to be present between forbidden and rejection attributes. However, there is a crucial difference in both cases. Mandatory (forbidden) attribute requires respective domain node (not) to be present in overlap. However, overlap calculation algorithm does not do anything to

enforce these conditions when intersecting domains – the resulting overlap just becomes invalid if the condition is false. On the other hand, overlap calculation algorithm automatically adds (injects) nodes with injection attribute, and removes (rejects) nodes with rejection attributes. Therefore, it is impossible to make overlap invalid based on presence of one among injection or rejection attributes, but usage of mandatory or forbidden attribute may impact overlap validity.

2.1.4.1.18. Injection vs Rejection Attribute

Generally, injection attribute has an opposite effect to rejection attribute. There are priorities set between injection and rejection attributes according to attribute prioritization rules. However, these priorities have effect only when applied to the same node via direct or indirect attribute assignment in a domain, or when both attributes brought together to the same node as a result of a domain merge. A pair of both injection and rejection attributes coming from different twin nodes during overlap calculation do conflict with each other, and this makes overlap invalid.

2.1.4.1.19. Mandatory vs Optional Attribute

Mandatory and optional attributes do not have opposite effects if applied to the same node via attribute assignment in a domain, or when both attributes brought together to the same node as a result of a domain merge. This is because mandatory node itself is allowed to be not present in data object (i.e., to be an optional at the same time).

However, mandatory and optional attributes are not compatible in case of an overlap with twin nodes having optional attribute in a source domain and mandatory attribute in a destination domain. Such combination results in an invalid overlap because mandatory and optional attributes are coming from different nodes. Mandatory attribute in destination domain requires value will be provided for the node in any data transfer operation, but optional attribute in a source domain makes it impossible.

2.1.4.1.20. Attribute Modifiers

Attribute modifier can be added to a node attribute, specifying additional conditions when the attribute is applicable to a node. The following attribute modifiers are defined in Pipe:

- **Source attribute modifier** – the corresponding node attribute is applicable only when the node is in a source domain of an overlap.
- **Destination attribute modifier** – the corresponding node attribute is applicable only when the node is in a destination domain of an overlap.

The node attribute is applicable to both source and destination domains if no modifier or both modifiers are specified at the same time. Attributes applicable on both sides of an overlap are called **bilateral attributes**, and applicable only to one side – **unilateral attributes**.

Modifiers can be applied to the following node attributes:

- **Mandatory (injection) attribute without modifier or with both modifiers** – a node must (will) be included in any overlap.
    - **Source attribute modifier is specified** – a node must (will) be included in any overlap if the node is in a source domain. As a result, the node value will always be read on a source side of data transfer.
    - **Destination attribute modifier is specified** – the node must (will) be included in any overlap if the node is in a destination domain. As a result, the node value will always be written on a destination side of data transfer.
- **Forbidden (rejection) attribute without modifier or with both modifiers** – a node must (will) be excluded from any overlap.
    - **Source attribute modifier is specified** – the node must (will) be excluded from any overlap if the node is in a source domain. As a result, the node will never be read on a source side of a data transfer.
    - **Destination attribute modifier is specified** – the node must (will) be excluded from any overlap if the node is in a destination domain. As a result, the node will never be written on a destination side of a data transfer.

Attribute modifier is indicated by prefixing attribute symbol in parentheses with a "less-than" character ("<") for a source attribute modifier, and a "greater-than" character (">") for a destination attribute modifier. For example:

```
User
@
   Account (>M+) # Mandatory attribute with source modifier applied to all child nodes.
      UserInfo
         Name
            FirstName(<M) -> string    # Mandatory attribute must be included in every
                                       # overlap as opposite attribute modifiers are
                                       # merged into bilateral attribute.
            LastName(M) -> string # Mandatory attribute must be included in any overlap
      Status(>R) -> string      # Node value will be excluded from overlap in case of a
                                # destination domain.
```

2.1.4.1.21. Domain Assignment

2.1.4.1.21.1. Domain Assignment Scope

Each component type has one of the following domain assignment scopes (see Table 10):

- **Component-level scope.** A domain is assigned to a component as a whole, and only that domain must be used for all its pins. The following component types use this scope:
    - Inline runlets.
    - Membanks.
    - Memlets.
    - Ports.

- **Pin-level scope.** Component pins may have different domains individually assigned to each of them explicitly or implicitly. The following component types use this scope:
    - Reusable runlet types.
    - Mergers.
    - Transformers.

Table 10. Domain assignment scope for different component types.

Component type	Domain assignment scope
Inline runlet	Component-level assignment scope
Membank	
Memlet	
Port	
Reusable runlet types	Pin-level assignment scope
Merger	
Transformer	

2.1.4.1.21.2. Domain Assignment Methods

The following domain assignment methods are used in Pipe (see Table 11):

- **Explicit assignment.** The domain is explicitly assigned to a pin or component. This method is used for the following component types:
    - Reusable runlet types.
    - Membanks.
- **Auto assignment.** The domain is taken from another component or inferred after applying domain transformations. This method is used for the following component types:
    - Memlet (its domain is taken from a connected membank).
    - Merger output (its domain is inferred after applying merge transformation combining all input domains into one output domain – see 2.1.4.2.1 “Mergers”).
    - Transformer output (its domain is inferred after applying all specified input domain transformations – see 2.1.4.2.2 “Transformers”).
    - Ports (it automatically takes domain of an associated pin from parent component or takes predefined domain in case of a system port).
    - Inline runlet outputs (input domain is used also as an output domain).
- **Default assignment.** If no domain is explicitly or auto assigned to an input of a component, then the **default domain** is used. Default domain is

applicable to inputs only, and it is produced as a merge of domains from all connections attached to an input. This method is used for the following component types:

- Inline runlets inputs.
- Transformer inputs.
- Merger inputs.

Table 11. Domain assignment methods availability.

Component type	Available domain assignment methods					
	Default assignment		Explicit assignment		Auto assignment	
	Inputs	Outputs	Inputs	Outputs	Inputs	Outputs
Reusable runlet types	No		**Yes**		No	
Inline runlet	**Yes**	No	No		No	**Yes**
Membank	No		**Yes**		No	
Memlet	No		No		**Yes**	
Merger	**Yes**	No	No	No	No	**Yes**
Transformer	**Yes**	No	No	No	No	**Yes**
Port	No		No		**Yes**	

2.1.4.1.22. Domain Inheritance

2.1.4.1.22.1. Introduction

Domain inheritance is a way to create a new domain by extending one or more existing domains after deriving their structures. New domain created by inheritance from other domains is called **inherited domain**. Existing domain used for inheritance is called **base domain**. Inherited domain may have more than one base domain. The list of all base domains for an inherited domain is called **inheritance list**. Domain inheritance is shown in domain specifications as an inheritance list above the name of the inherited domain. The name of each base domain in the list starts with a new line and it should be prefixed by the left arrow character combination ("<-"). For example:

```
<- Person    # Base domain "Person"
<- Customer  # Base domain "Customer"
User         # Domain "User" is inherited from domains "Customer" and "Person"
@
   Name
      FirstName -> string
      MiddleName -> string
      LastName -> string
```

2.1.4.1.22.2. Default Inheritance Rules

The following **default inheritance rules** are applied for a new domain created as a result of inheriting from one or more other domains:

- Inherited domain is created by merging all base domains together.

- Domain nodes can also be specified directly in an inherited domain, and they are divided into two categories:
    - Nodes without twin nodes in any base domain are staying in inherited domain exactly as they are specified there.
    - Nodes with one or more twin nodes in any base domain are getting merged with twin nodes from all base domains where they exist.

2.1.4.1.22.3. Inheritance Override

Inherited domain is created as a merge of all base domains. However, it might be required to use not all, but only some base domains for a merge of a specific node, or not to inherit the node at all. **Inheritance override** is used to override default inheritance rules for a specific node, allowing only specified base domains to be participating in a merge, or not to inherit a node at all (this is equivalent to using zero base domains for a merge, meaning the node is excluded from inherited domain).

To use inheritance override, full path to the name of the node from a base domain must be specified in inherited domain enclosed in parentheses, prefixed with two-colon character combination (“::”) and with the name of the base domain specified before two-colon divider. If more than one base domain must participate in the merge, the whole domain list must be enclosed in parentheses with comma character (“,”) as divider between names of base domains. If node must not be inherited at all, the list inside of parentheses must be empty. For example:

```
Vehicle::(@/Car/Type)                  # Node “@/Car/Type” must be inherited from
                                       # the domain “Vehicle” only.
(Steel, Copper)::(@/Metal/Price)       # Node “@/Metal/Price” must be created as merge
                                       # of base domains “Steel” and “Copper”.
()::(@/House/Location)                 # Node “@/House/Location” should not be
                                       # inherited at all.
```

All specified base domains must be part of inheritance list and path to the node must exist in all specified base domains.

2.1.4.1.22.4. Node Redefinition

A node can be defined directly in an inherited domain and this node will be merged with all twin nodes from base domains, if there is any. This is called **node redefinition**. Node defined such way is called **redefined node**. It does not have to have a twin node in any base domain. Node redefinition in inherited domain is declared the same way as regular nodes are declared in domain specifications: node name followed by attributes and data type with all parent nodes defined one per line above it and with decreasing indentation for each higher level of hierarchy all the way up to the root node. For example:

```
<- Person    # Base domain “Person”.
Employee     # Inherited domain “Employee”.
@
   Name  # Including parent node “Name” on the way from the root to “FirstName” node.
      FirstName(N) -> string # Redefinition of “FirstName” node with nullable attrib.
```

Node redefinition can be combined with inheritance overrides to control how the node defined in an inherited domain merges with twin nodes from base domains. The

same syntax is used: the name of the node must be prefixed by two-colon character combination ("::") and with names of base domains specified before two-colon divider. If more than one base domain participates in the merge, the whole domain list must be enclosed in parentheses with comma character (",") as divider between names of base domains. If node must not be inherited at all, the list inside of parentheses must be empty, meaning that the defined node overrides all twin nodes from base domains. For example:

```
Asset::Price -> int     # Node "Price" must be merged with base domain "Asset".
(Employee, User)::Person -> {Person}    # Node "Person" must be merged with base
                                        # domains "Employee" and "User".
()::Weight -> float     # Node "Weight" should not merge with any base domain.
```

All specified base domains must be part of inheritance list, and path to the node must exist in all specified base domains. A node from a base domain(s) must be mentioned no more than once anywhere in inherited domain as a part of inheritance override or node redefinition.

2.1.4.1.22.5. Forced Definition

Forced definition is used when inheritance override and node redefinition are not enough to override inherited domain. It has syntax of a node redefinition, but forced redefinition covers additional cases:

- Excluding an attribute defined for a node as a result of inheritance (**Forced attribute definition**).
- Forcing precise data type no matter what data types are coming from base domains (**Forced data type definition**).

Forced attribute definition is specified for an attribute by enclosing it in square brackets ("[]") instead of parentheses. Attribute grouping and source/destination modifiers are allowed in forced attribute definition. Forced attribute definition does not require for that attribute to be assigned to the node in base domain – attribute removal has no effect if the attribute is already missing. Forced data type is shown as data type in parentheses. Empty parentheses should be used to delete any data type from a node, removing value slice from the node and turning it into a non-storage node. For example:

```
Name[N] -> string       # Exclude nullable attribute from node "Name".
Length[<M] -> float     # Exclude mandatory attribute with source modifier. If the node
                        # has full mandatory attribute, this exclusion works like
                        # subtraction, leaving attribute ">M" after exclusion.
Weight[O+] -> float     # Exclude optional attribute from node "Weight"
                        # and all children wherever the optional attribute is present.
Amount -> (int)         # Force data type "int" for a node "Amount".
Type -> ()              # Remove value slice from node "Type" making it a non-storage
                        # node.
```

2.1.4.1.22.6. Inheritance in Object Specifications

Inheritance lists and inheritance attributes are never shown explicitly in object specifications because all objects there must be **inheritance-expanded** – nodes from all base domains should be shown as a part of an inherited domain according to default inheritance rules and after applying all inheritance overrides and redefinitions.

2.1.4.2. Mergers and Transformers

2.1.4.2.1. Mergers

2.1.4.2.1.1. Introduction

Merger waits for arrival of signals to all its inputs and then merges their data objects. It has two or more inputs, and one output.

2.1.4.2.1.2. Merger Types

If domains from two or more merger inputs have a non-empty overlap where both twin nodes are storage nodes, then these storage nodes conflict with each other because there are two node candidates to send its value to a single merged data object, but the merged node can hold only one value. Such situation is called **domain collision** and these storage nodes are called **collided nodes**. The problem of domain collision is that the two collided twin nodes contain different values coming from two different input data objects, and both values need to be somehow accommodated under a single storage node of the merged output data object. The following merger types are defined based on the method of domain collision resolution:

- **Naming merger** – names are used for domain collision resolution. Collided storage nodes are created as children of a destination node in a merged domain with resolution names (see 2.1.4.11.3.7 “Merger Resolution Names”) assigned as names of the child nodes (see Figure 12). The names of new child nodes must not exist under parent node of the merged domain.
- **Priority merger** – unique integer number in a range from 0 to +32,767 is assigned to each input indicating priority of domain collision resolution (lower number indicates higher priority). Storage node candidate from an input with higher priority will be selected during collision resolution. One or more inputs may have their priority omitted, meaning their storage node candidates for overlapping storage nodes will never go into a merged domain. No overlapping storage node goes into a merged domain if all inputs have priorities omitted.
- **Exclusion merger** – only storage domain nodes not overlapping with storage nodes of any other input domain and all non-storage twin nodes participate in the merge. All overlapping storage nodes are excluded. Exclusion merger is equivalent to priority merger with priorities for all inputs omitted.
- **Strict merger** – a connection cannot be attached to an input if it creates non-empty overlap with any other input domain with at least one instance where both twin nodes are storage nodes.

2.1.4.2.1.3. Merger Processing Algorithm

Each merger input has a queue and input signals are processed in the order of arrival. Merger starts processing data objects from all its inputs only when at least one signal is

waiting for processing on every input. The set of signals at the front of each input queue is called **merge vector**. The process of merge vector accumulation (i.e., waiting until all inputs have at least one input signal) before triggering merger execution is called **vector formation**. An event of triggering signal processing event after completion of vector formation is called **vector collapse**. Merge vector signals are getting removed from corresponding input queues during vector collapse to start their processing by merging respective data objects. The next merge vector formation event starts after the previous merge is completed and signal of a merged data object is produced at the output, unless the next merge vector is already filled (i.e., each input already has at least one signal waiting) – merge vector collapse happens immediately in such case for the next merge vector, skipping vector formation step.

Blank signals are counted as part of a merge vector, but they do not provide any domain nodes for a merged data object. Merger has a domainless output producing blank signals in case a merge vector contains only blank signals. Null object of a merged domain is produced at the merger output if merge vector contains combination of at least one null object and zero or more blank signals.

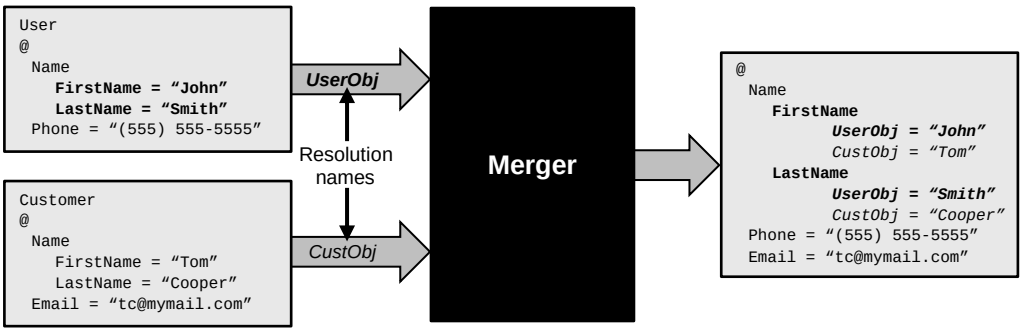


Figure 12. Domain collision resolution by a naming merger.

2.1.4.2.1.4. Merger Bond Attributes

The following bond attributes are available for merger inputs:

Immunity attribute – vector collapse happens with signals present on all merger inputs except on inputs with bonds having immunity attribute, i.e., vector collapse may happen without a signal present on inputs with immunity attribute. All or none of bonds for a single input must have immunity attribute specified, i.e., it is not allowed to have immunity attribute specified only for some bonds of the same input. Domain of an input with immunity attribute will have the optional attribute added to all their nodes in an output merged domain, meaning those nodes might not be present in output domain if corresponding inputs with immunity attribute do not have signals present at the moment of vector collapse.

Purge attribute – vector collapse starts when a signal arrives to any input via a bond with purge attribute, no matter if other merger inputs have signals. Domains of all merge inputs will have optional attribute specified for all their nodes in output domain if

purge attribute is present on any input bond, except case when only one input has a purge attribute specified – nodes specifically for that input will not have optional attribute added in an output merged domain.

Immunity and purge attributes are compatible with all other input bond attributes applicable to merger inputs. However, immunity and purge attributes have opposite effect and therefore, they cannot be used together for the same or different bonds of the same input.

2.1.4.2.2. Transformers

2.1.4.2.2.1. Introduction

Transformer modifies structure of a single data object (without impacting structure of a corresponding domain) to reconcile source and destination domains of an output and input that need to be connected, but their overlap is not properly structured, or it is invalid making connection impossible. Transformer can be placed between output and input, playing a role of a connection adapter by transforming the structure of a source domain so that it produces a valid and acceptable overlap with a destination domain. Transformer has one input and one output.

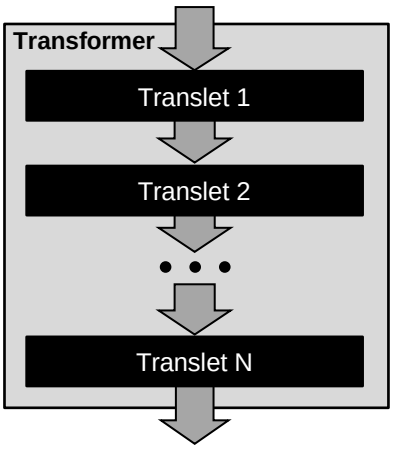


Figure 13. Internal structure of a transformer.

2.1.4.2.2.2. Translets

Transformer describes domain transformation as a chain of one or more **translets** – atomic transformation steps. An output of a translet connects to an input of the next one, or it becomes the final output of the transformer in case of the last translet in a chain (see Figure 13). Each translet contains the following parameters:

- **Target node** – full path to the target node of a transformation (crossover paths are allowed).
- **Operation** – transformation operation applied to the target node (see Table 12).

- **Arguments** – extra information for a specific operation. The structure of this parameter depends on operation (see Table 12).
- **Strict check flag** – setting this flag will cause a translet to throw an exception (see 2.1.4.3 “Handling Exceptions”) in case the target node is not found in incoming data object due to a null data object, or because of a target node with optional attribute is missing in a specific data object.

If translet operation changes location of a target node within a domain tree, or a new node gets created in a new location (for example, copy or move operation), then the name of a node in a new location must be unique among its new sibling nodes. All missing parent nodes along a path will be created automatically as non-storage nodes when a node gets created in a new location.

Table 12. Translet operations.

Operation	Description	Parameters
AddChange	Adding new node as a child of the target node if the node does not exist or changing an existing node. New node attribute list and new domain name override list fully replace all attributes and inheritance override domains for existing node. Node attribute list includes source/destination modifiers (for mandatory, forbidden, injection and rejection attributes) and attribute grouping flag for each attribute of the list.	• Path to the target node. • Node name. • Data type. • Node attributes list. • Base domains override list. • Forced definition (flag specifying if forced definition of data type is applied). • Default or constant value (for a node with a default or constant attribute).
Insert	Adding new node as a parent of the target node. The parent of the target node becomes the parent of a newly added node, and the target node become a child of a newly added node. Added node becomes a new root node if the target node is a root node.	See above (same parameters as for “AddChange” operation).
Rename	Renaming the target node. The target node cannot be a root node.	• Path to the target node. • New node name.
Delete	Deleting the target node and all its children. An output of the translet is domainless if the target node is a root node.	• Path to the target node.
Copy	Copying the target node and all its children to a new location.	• Path to the target node. • Path to the node that will be a new parent of the target node. • Name of target node in case the target node is a root node.

Operation	Description	Parameters
Move	Moving the target node and all its children to a new location. The target node cannot be a root node. If parent is direct or indirect child of target node, then it gets split into two copies: one gets moved, and other stays where it was before.	• Path to the target node. • Path to the node that will be a new parent of the target node.
DeleteOne	Deleting the target node only and moving all its children one level up under the parent of the deleted target node. New non-storage root node gets created if the target node is a root node.	• Path to the target node.
CopyOne	Copying the target node only, without its children.	• Path to the target node. • Path to the node that will be new parent of the target node. • Name of target node in case the target node is a root node, and new parent node is not a root node.
MoveOne	Moving the target node only and moving all its children one level up under the parent of the moved target node. New non-storage root node gets created if the target node is a root node.	• Path to the target node. • Path to the node that will be new parent of the target node.
OuterDelete	Deleting all nodes except the subtree starting from the target node, which becomes a new root node.	• Path to the target node.
Inherit	Fully replacing base domains with the list from parameters.	• Base domains list.

2.1.4.3. Handling Exceptions

2.1.4.3.1. Introduction

When error happens inside of a coded runlet, it produces (throws) an **exception** – system event propagating up the runlet tree until it reaches a level above the root runlet of application, where it is received by the system handler that terminates running application and prints details of the exception to a system output.

An exception might be intercepted on any level of a runlet tree, preventing application termination and handling error inside of the running application instead. Rectangular area called **traplet** can be placed directly on a pipeline surface with the following placement condition: no pipeline component can be covered by a traplet

partially – it must be completely inside or fully outside of any traplet. Also, traplet rectangles cannot intersect partially, but a traplet may contain other traplets inside. Execution of application after throwing an exception pauses until it is intercepted and processed, or the exception reaches the root runlet terminating the application. Traplet has one output with a nullable domain assignment. Traplet intercepts exceptions coming from any depth level of any runlet located inside of the traplet area, stopping further exception propagation, and producing **exception object** at the traplet output containing information about the exception. Traplet also has a nullable input with a domain of exception object. A signal entering traplet input produces the same exception object at the traplet output. Traplet input can be used to emulate exceptions for debugging and other purposes.

2.1.4.3.2. Exception Specification

Exception source is a coded runlet directly thrown an exception. Exception object is created by an exception source, propagating up the runlet tree until the exception object is either intercepted by a traplet on its way up, or it eventually escapes the applications causing its termination. If exception object is intercepted by a traplet, then it is produced at its output. Exception object must conform to the following domain:

```
Exception
@
   Code -> int # User-defined integer code of the exception in range from 0 to +32,767.
   Description -> string  # Details of the exception.
   EndpointPath -> string # Endpoint path to the exception source (see Endpoint path).
   DataObject(N) -> any # Data object caused the exception (null for a blank signal).
   Data(N) -> any       # Any additional data related to the exception (null if none
                        # exists).
```

2.1.4.3.3. Exception Codes

There are the following traplet types according to exception codes it can intercept:

- **Default traplet** – intercepts exceptions with any code.
- **Selective traplet** – intercepts only exceptions with certain codes. The list of accepted exception codes for the selective traplet is called **acceptance specification** and it contains exception codes divided by comma and enclosed in square brackets (“[]”). The list can contain both single codes as well as ranges shown as two inclusive minimum and maximum codes with a minus character (“-”) between them. The list should be sorted in ascending order. Minus character in square brackets is an acceptance specification of a default traplet (“[-]”). Empty brackets (“[]”) indicate traplet that does not accept any exception code. For example:

```
[2, 14-18, 24-35]   # Accepted codes are 2, 14-18, 24-35.
[-]                 # All codes are accepted.
[]                  # No codes are accepted.
```

Propagation of a thrown exception continues up the runlet tree until it is either intercepted by the first traplet accepting exception code specified in propagating exception object, or until it reaches level above the root runlet terminating application.

2.1.4.4. Component State

2.1.4.4.1. Stateful Components

Application state is a state of the root runlet of the application. Definition of a **component state** depends on component type. The component state is defined for the following **stateful components**:

- **Coded runlets** – contents of all variables inside of the runlet.
- **Membanks** – contents of the membank.
- **Memlets** – contents of a connected membank.
- **Synclets, traplets and composite runlets** – set of states for all stateful child components inside of the component.

2.1.4.4.2. Default Behavior

The default behavior of all stateful components is preservation of the state between processing of two successive input signals, or after intercepting an exception for traplets. Inline runlets are exception from this rule – their default behavior is resetting component state after finishing signal processing.

2.1.4.4.3. Component State Reset

Component state reset is an operation applicable to a component while it is not in the middle of processing any signal. It is defined for all stateful components as following (depending on component type):

- **Coded runlets** – resetting all variables inside of runlet to their initial values.
- **Membanks** – putting default data object to the membank.
- **Memlets** – resetting connected membank.
- **Synclets, traplets and composite runlets** – resetting all stateful child components inside of the component.

2.1.4.4.4. Component State Overrides

Pipe provides capabilities to override default behavior for all stateful components. The following **component state overrides** are provided for stateful components:

- **State retention** – preservation of state between processing of two successive input signals. This override is applicable to inline runlets.
- **Self-reset** – implicit state reset of a component after finishing signal processing, or after intercepting an exception for traplets. This override is applicable to reusable runlets, synclets and traplets.
- **Parent reset** – implicit state reset of a component before a new signal starts processing by a parent composite runlet containing a component with this override. The override is applicable to reusable runlets, memlets, synclets and traplets.

- **Top reset** – implicit state reset of a component before a new signal starts processing by an application containing the component with this override. The override is applicable to reusable runlets, memlets, synclets and traplets.
- **Explicit reset** – explicit state reset of a component when signal enters a special domainless **reset input** of the component. Reset input is provided by default to all stateful components allowing explicit reset. This override is applicable to inline runlets, reusable runlets, memlets, synclets and traplets.

All reset overrides apply to the whole component. Multiple state overrides can be applied to the same component. Applicability of state overrides to component types is provided in Table 13.

Table 13. State overrides applicability.

Component type	Applicable state overrides				
	State retention	Self-reset	Parent reset	Top reset	Explicit reset
Inline runlet	**Yes**	No	No	No	**Yes**
Reusable runlets	No	**Yes**	**Yes**	**Yes**	**Yes**
Memlet	No	No	**Yes**	**Yes**	**Yes**
Synclet	No	**Yes**	**Yes**	**Yes**	**Yes**
Traplet	No	**Yes**	**Yes**	**Yes**	**Yes**

Parent reset and top reset of a memlet or membank does not lead to production of a signal on any output other than reset output (see 2.1.4.4.5 “Reset Output”). Explicit reset of a memlet replaces the current memlet content with a default data object. All bond attributes except read attribute (as it does not allow for a memlet content change) are allowed for memlet explicit reset inputs, i.e., explicit reset of a memlet results in production of an output signal according to applied memlet bond attributes. For example, previous memlet content is produced at the output in case reset signal is sent to a memlet via a push bond type. Explicit reset signal has no effect in the following cases:

- **Inline runlet without state retention override** – inline runlets without state retention override do not retain the state by default, so there is nothing to reset explicitly.
- **Reusable runlet, synclet or traplet with self-reset override** – they will reset state implicitly after finishing signal processing, so an explicit reset signal will always be sent to a component already reset implicitly after finishing processing of the last signal. On the other hand, if it did not start processing any signal yet then it is already in initial state and therefore, it again does not make sense to reset component which is still in its initial state.

2.1.4.4.5. Reset Output

Reset output is a nameless output present by default in all components supporting component state. Reset output produces a signal at the moment the component reset is

triggered, and it has the following domain:

```
Reset    # Scalar domain.
@ -> int  # Code of the component reset reason:
             # 0 - implicit reset of inline runlet: default behavior for inline runlet.
             # 1 – self-reset.
             # 2 – parent reset.
             # 3 – top reset.
             # 4 – explicit reset.
```

Output reset port inside of a composite runlet is connected to reset output of the runlet. This port can be used when non-reset signals must be routed to external reset output of composite runlet to emulate reset. There is also **input reset port** inside of a composite runlet, but it is not connected to external reset input. It has the same domain as reset output and it can be used to process reset signal. Output and input reset ports produce signals when a composite runlet is reset for any reason (including explicit reset).

2.1.4.5. General Bond Attributes

The following **general bond attributes** are used to indicate **general bond type** (exactly one attribute must be specified for each input bond of applicable component types to indicate input bond type):

- **Regular attribute** – all new signals entering an input are enqueued and consumed one-by-one for processing. This bond type represents “All signals are processed” semantics when all signals arrived at the input are processed.
- **Front attribute** – every new signal entering an input is enqueued only if there are no signals waiting at that input and the component is not processing a signal from that input. Otherwise, the new signal is discarded. This bond type represents “First signal is processed” semantics when the first signal arrived at the input is processed.
- **Back attribute** – every new signal entering an input is getting enqueued, while all other signals waiting at that input are getting discarded. This bond type represents “Last signal is processed” semantics when the last signal arrived at the input is processed.
- **Reset attribute** – every new signal entering an input and all signals waiting at that input are getting discarded. This bond type represents “No signal is processed” semantics. Inputs with reset attribute are domainless.

Front and back attributes represent queue optimization methods. Front attribute is useful in cases when new signal is not needed if one is already enqueued (for example, when processing of a single signal is enough). Back attribute allows to discard previous signals when a new signal arrives (for example, arrival of new updated coordinates makes all enqueued coordinates obsolete). Reset attribute can be used for cleaning input queue. General bond attributes are applicable to inputs of the following component types:

- **All runlet types except beacons** – bond attributes are applicable to reusable runlets on instance level, i.e., runlet instances can have different attributes defined for the same bond.

- **Synclets** – bond attributes are applicable to points where connection crosses synclet boundary inwards (synclet inputs) and outwards (synclet outputs).
- **Traplets** – bond attributes are applicable only to explicit inputs and outputs.

2.1.4.6. Tear Bond Attribute

Tear bond attribute strips away data object from all signals arrived at input queue. Tear bond attribute is equivalent to a domainless input applied not the whole input, but only to input bonds with a tear attribute. Tear bond attribute is applicable to the same component types and has the same rules as for general bond attributes. Tear bond attribute is compatible with all general bond types and therefore, this attribute can be combined with any general bond type attribute.

2.1.4.7. Global Bond Attribute

Global bond attribute expands scope of general bond attribute from a single input to any input of a component. It is compatible with all general bond attributes except regular. Global bond attribute must be applied together with a general bond attribute. Global bond attribute modifies behavior of a general bond attribute in the following way (all behavior changes, compared to a version without a global attribute, are underlined):

- **Front attribute** – every new signal entering an input is enqueued only if there are no signals waiting at any component input and the component is not processing a signal from any component input. Otherwise, the new signal is discarded.
- **Back attribute** – every new signal entering an input is getting enqueued, while all other signals waiting at any component input are getting discarded.
- **Reset attribute** – every new signal entering an input and signals waiting at any component input are getting discarded.

2.1.4.8. Active Bond Attribute

Active bond attribute indicates that an output with such bond may produce internal signals to be enqueued at a parent of such component with active attribute. No component output is allowed to produce internal signals unless it has active attribute assigned. The purpose of active bond attribute is providing an indication that an output may produce internal signals and workflows must be prepared to handle such cases. Active bond attribute is applicable to outputs of the following component types:

- Prime runlets.
- Scripted runlets.
- Composite runlets.
- Memlets.

If any bond of an output has active attribute assigned, then all other bonds of the same output must have active attribute assigned too.

Active bond attribute is assigned to runlets on a template (not instance) level, meaning that all instances of a runlet have active bond attribute assigned automatically if this attribute is assigned to a corresponding input bond in a template runlet.

If output of any internal component of a composite runlet has active output, then it can be connected only to external outputs of the composite runlet that have active bond attribute assigned. If memlet output has an active bond attribute specified, then all its sibling memlets must have active bond attribute specified for its outputs too. Broadcast bond attribute implicitly assumes presence of active bond attribute.

2.1.4.9. Staging Bond Attribute

Rule SDP-4 states: *"Output signals must become available on external outputs immediately after their production without waiting for completion of input signal processing"*. However, there are situations when output signals must become available only after input signal processing is completed. **Staging bond attribute** is used for this purpose. Staging bond attribute should generally not be used for generating output signals while continuously processing one input signal in a loop, as no signal would ever appear on outputs with staging bond in this case until loop ends and signal processing finishes.

2.1.4.10. Signal Priorities

By default, signals are enqueued at an input of receiving component in the order of their arrival. However, priority integer number can be assigned to a signal in a range from -32,768 to +32,767, providing signal priority among all inputs (**global priority**) or within a single input (**local priority**), depending on a specified **priority type** (local or global). Lower number has a higher priority, meaning a signal with such number will be processed before processing signals with higher numbers relative to that signal.

No signal without explicitly assigned priority should start processing by a component until all signal with explicitly assigned priority are processed. Signal priority takes effect only if it enters component via a bond with a regular general attribute. Signal priority can be specified only programmatically inside of a coded runlet, before sending an external signal to an output or before firing an internal signal (see 2.5 "Runlet API").

2.1.4.11. Names

2.1.4.11.1. Name Classes

There are the following name classes in Pipe:

- **Pin names** – names of component inputs and outputs.
- **Namespace names** – prefix part of name for domains and template runlets.
- **Common names** – names used in the following cases:
  - Domain names.
  - Domain node names.
  - Template runlet names.

- Pipeline member names.
- Membank names.
- Merger resolution names.

2.1.4.11.2. Pin Names

2.1.4.11.2.1. Naming Rules

The following naming rules are used for pin names:

- Allowed characters are upper-case letters, digits, and underscore character ("_").
- The maximum name length is 24 characters.

2.1.4.11.2.2. Assignment Rules

The following assignment rules are used for pin names:

- All component types that can potentially have more than one input (output) must have names assigned to all its inputs (outputs), even in cases when a specific instance component of such type uses only one input (output). The only exceptions from this rule are reset input and reset output that should never have a name. Based on this requirement, the following component types must have input (output) names:
  - Input and outputs of reusable runlet types.
  - Merger inputs.
- Tester has the following fixed non-changeable names for its outputs:
  - Name **YES** for a positive output.
  - Name **NO** for a negative output.
- A name assigned to a component input (output) must be unique among all inputs (outputs) of the component.

2.1.4.11.2.3. Default Pin Names

If no name is assigned to a pin explicitly, the **default pin name** will be automatically assigned to the pin using the following format:

- For input pin: **IN*{X}*** (for example: **IN1**, **IN5**, **IN25**).
- For output pin: **OUT*{X}*** (for example: **OUT3**, **OUT7**, **OUT14**).

Parameter ***{X}*** is the lowest available number for input or output pin name starting from zero. For example, if a component has inputs with pin names **IN0**, **IN1** and **IN5** then the next available default pin name for an input is going to be **IN2**.

2.1.4.11.3. Common Names

2.1.4.11.3.1. Naming Rules

The following naming rules are used for common names:

- Names are case-sensitive.
- Allowed characters are letters, digits, and underscore character ("_").
- The maximum name length is 512 characters.

2.1.4.11.3.2. Member Names

All membanks, memlets, runlets, synclets, traplets and ports located in one pipeline are called collectively **pipeline members**. All pipeline members must have their own names called **member names** unique among all pipeline members, except ports whose names are either inherited from connected external pin for regular ports, or they have fixed pre-assigned names for system ports. Reset input and output ports do not have names. If no name is assigned to a pipeline member (except ports) explicitly, the **default member name** will be automatically assigned to the component using the following format:

- For membanks: **V*{X}*** (for example: **V8**, **V51**, **V89**).
- For memlets: **M*{X}*** (for example: **M3**, **M14**, **M52**).
- For runlets: **R*{X}*** (for example: **R0**, **R43**, **R78**).
- For synclets: **S*{X}*** (for example: **S2**, **S11**, **S35**).
- For traplets: **T*{X}*** (for example: **T5, T28, T62**).

Parameter ***{X}*** is the lowest available number for a member name starting from zero. For example, if pipeline contains runlets with names **R0**, **R2** and **R7** then the next available default member name for a runlet is going to be **R1**.

2.1.4.11.3.3. Component/Endpoint Path

Each component in a runlet tree can be uniquely identified by a full path to that component from the root runlet. Such path is called a **component path**. It must include member names of all direct and indirect parent composite runlets of the component up to the root level of a runlet tree. Slash character ("/") is a divider between names in the path. For example:

```
@/Processor/MainProc/VolumeCalculator   # Path to the runlet "VolumeCalculator"
```

Endpoint path is a path to the specific input or output of a component. Endpoint path is shown as a pin name prefixed by the path to a component containing that pin, with two colon characters ("::") as a divider between them. For example:

```
@/DataManager/Reader::INOBJ     # Path to input "INOBJ" of runlet "Reader"
```

2.1.4.11.3.4. Anonymous Domains

Anonymous domains stay assigned to one pin or membank without an ability to be reused,

as they do not have names to refer to. They can be inherited from one or more non-anonymous domains. Anonymous domains are used in the following cases:

- **In-place domain definition during domain assignment to a runlet pin or membank, or when defining new domain data type within another domain.** A new nameless domain can be created and inserted **in-place** during domain assignment to a runlet pin or membank, or when defining new domain data type within another domain. It is allowed to use inheritance during domain definition in-place, and this is the best method to adapt existing domain for a specific pin, membank or new domain data type without creating new domain name to avoid namespace pollution.
- **Overlap domains**. All overlap domains are anonymous.
- **Merger and transformer output domains**. Output domains of mergers and transformers are anonymous.
- **Paramlets** – paramlet domains can be anonymous (see 2.4.1 "Runlet Parametrization").

2.1.4.11.3.5. Port Names

Ports do not have their own names as they automatically take names of a parent composite runlet pins for regular ports, or fixed names pre-assigned to system ports in a specific Pipe implementation. No names are assigned to reset ports.

2.1.4.11.3.6. Membank Names

Each membank must be assigned a name unique among all membanks in a pipeline. A membank cannot have a memlet outside of a pipeline where the membank is defined. All sibling memlets refer to their shared membank by its name.

2.1.4.11.3.7. Merger Resolution Names

Inputs of a naming merger can have secondary names assigned to them along with a regular pin name. These names are called **resolution names** and they are used to resolve collided nodes for naming merger type (see Figure 12). If no resolution name is specified for a naming merger input, then its pin name is automatically assigned as resolution name.

2.1.4.11.4. Namespaces

Template runlets and domains may contain a **namespace** as a second (prefix) part of their names. Namespace is a part of the name that provides scope for the template runlet or domain to minimize probability of name collisions inside of their own solution, and after importing them to other solutions. The name without namespace part is called **base name**, the name prefixed by a namespace is called **full name**. Full name must be unique in the solution among other objects of the same type (template runlets or domains). Only base domain names must be shown in domain and object specifications. If base names from different namespaces collide in a domain or object specification, then a **local alias** must be assigned to one or both colliding domain names. Local alias is a synonym to

refer to the domain within that specification only. All local aliases must be placed at the beginning of a domain or object specification with a key word ***alias*** followed by name alias and original full name divided by an equal character ("="). For example:

```
alias ReaderNew = com.upsoft.Reader  # "ReaderNew" is an alias for "com.upsoft.Reader".
```

The namespace name is a sequence of **namespace segments** divided by a dot character ("."). The same dot divider is used also between namespace and base names. The naming rules for namespace segments are the same as for common names (see 2.1.4.11.3.1 "Naming Rules"). For example:

```
com.westsoft.Reader # "Reader" is the base name, "com.westsoft" is the namespace names.
```

2.1.4.11.5. Synonyms

Importing template runlets and domains into another solution might cause name collisions of imported objects with names of existing template runlets and domains in the solution, even if namespaces are used. To resolve this problem, imported objects with colliding names must be assigned **synonyms** in such case – alternative unique names assigned to a template runlet or domain for use only in the solution they are imported into. Original name must be saved after assigning a synonym so that it could be recovered later if necessary. Both manual and automatic synonym assignment methods must be available in Pipe implementations. Synonyms can only change base names, but they cannot change namespace part of imported template runlets or domains.

2.2. Visual Representation

2.2.1. Diagram

2.2.1.1. Colors

Pipe supports two diagram modes differing by colors only: one production diagram, and any number of non-production mode diagrams. **Production diagram mode** has strict coloring rules. Background color is white with a possibility of a background grid with grey lines. Elements are shown using black, white, and grey colors only strictly according to drawing specification for each element. Inversion of colors is allowed for implementing dark mode. All colors are allowed for a **non-production diagram mode**, and non-production diagrams can have different colors.

2.2.1.2. Orientation

The diagram orientation is vertical with a top-down direction of a general signal flow. All components must have input connections attached to a top edge, and output connections – to a bottom edge of a component shape (except inline runlets that can have inputs and outputs attached to any of its corners – see 2.2.2.1.3 "Inline Runlet").

Layout of two adjacent components does not have to be strictly one under another. For example, they can be on the same level horizontally for optimal space filling, as long as the general diagram flow goes from top to bottom.

2.2.1.3. Common Elements

2.2.1.3.1. Junctions

Connection junction is an intersection point of two connection lines where they are fused together into one connection using a connection junction point shown by a solid black dot at the intersection point. If two connection lines cross each other without a topological contact between them (i.e., they stay isolated from each other), then there should not be any shape at the intersection point (see Figure 14).

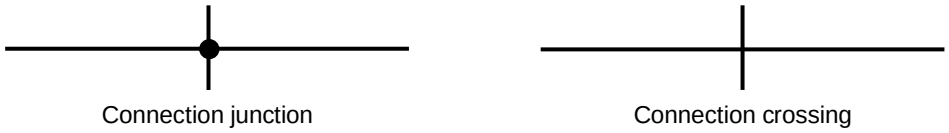


Figure 14. Connection junction and crossing.

Pin junction is an intersection point between connection and component boundary. Pin junction is a visual representation of a point where connection enters a component. Pin junction point is shown by a solid black dot with a center at the location where connection line is crossing component boundary (see Figure 15).

Figure 15. Pin junction.

2.2.1.3.2. Connections

Connection is shown as a black polyline with solid black arrows attached to all destination points (see Figure 16). Only horizontal and vertical polyline segments are allowed.

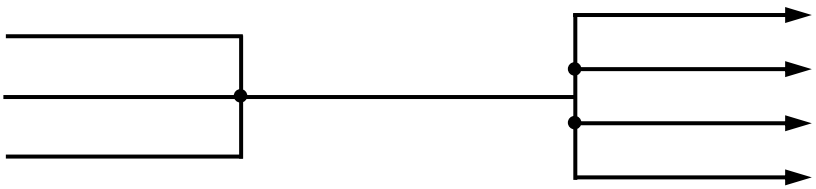


Figure 16. Connection.

2.2.1.3.3. Copy Counters

There are cases when more than one copy of the same element can be shown on diagram. **Copy counters** are used in such cases to identify each copy, also providing a total number of copies.

Copy counter is a string suffix automatically added at the end of element name in case more than one copy of the element exists on a diagram. The format of a copy

counter is two numbers divided by slash character (“/”) and enclosed in parentheses. The first number before slash character indicates unique ordinal number of a copy starting from zero. The second number after slash character shows total number of copies. For example, an element with name “MyElement” will have the following suffixes appended to names for each of component copies:

```
MyElement(0/2)
MyElement(1/2)
```

2.2.1.3.4. Input/Output Separation

When different connections enter the same component input, they are forced to merge to be able to enter the shared input. For example, Figure 17 assumes output of the component ***Source_1*** must be connected to input of ***Destination_1*** only. However, it must merge to connection going out from output of ***Source_2***, as these two connections enter the same input of ***Destination_1***. As a result, signals coming out from ***Source_1*** are entering not only ***Destination_1***, but also ***Destination_2*** in the merged connection, contrary to the original intent. Similar situation is for output of ***Source_3*** that goes only to input of ***Destination_2***, but it is forced to merge with another connection going from ***Source_2*** to the same input of ***Destination_2***.

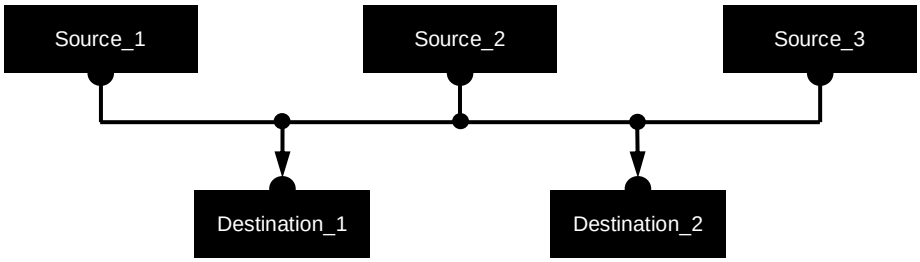


Figure 17. Single multipoint connection.

Multiple connections can enter the same input separately to avoid merge by entering input at different angles, but diagonal line segments are not allowed for Pipe connections. Also, such solution would quickly overwhelm a single input connection point if too many connections are entering it (see Figure 18).

Figure 18. Multiple diagonal connection lines entering the same input.

The solution of this problem is **input separation** – entering the same input via different connection points (see inputs of ***Destination_1*** and ***Destination_2*** in the Figure 19). As a result, ***Source_1*** can connect to ***Destination_1*** without the need to merge with

other connections entering the same input. ***Source_3*** is also able to connect only to ***Destination_2***, avoiding any forced connection merges.

Pipe also provides **output separation** feature that allows to isolate connections going out from the same output (see output of ***Source_2*** in the Figure 19). This feature can be useful when destination domains of one of more inputs not connectable with a source domain of an output. Single multipoint connection from a one output to multiple inputs will be rejected as a whole, even when only one of inputs is not connectable. However, connection separation localizes invalid problematic destination domain within a single one-to-one connection, not impacting others. For example, a single multipoint connection going out from ***Source_2*** to ***Destination_1*** and ***Destination_2*** in the Figure 17 will be invalid as a whole, even when only one of destination domains for inputs of ***Destination_1*** or ***Destination_2*** is not connectable with a source domain for the output of ***Source_2***. However, separating a single multipoint connection into two distinct connections ***Source_2*** -> ***Destination_1*** and ***Source_2*** -> ***Destination_2*** via output separation feature prevents destination domain problem in one of the inputs from impacting other input (see Figure 19).

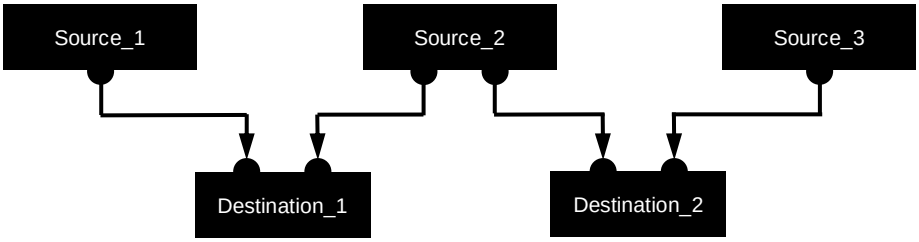


Figure 19. Input and output separation.

2.2.1.3.5. Named Pins

Some components must have names assigned to their pins. The names of input (output) pins are shown at the top (bottom) edge inside of a component. It is allowed to omit pins that do not have any connections attached to them in a specific component instance.

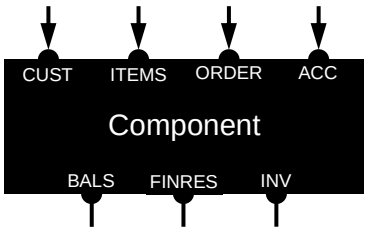


Figure 20. Named pins.

Pin junctions are shown directly above (for inputs) or directly below (for outputs) pin name with centers of the corresponding pin names horizontally aligned with centers of pin junctions. The width of the component should be auto adjusted to accommodate

the longest row of pin name with a reasonable horizontal gap between pin names (see Figure 20).

2.2.1.3.6. Pin Repetition

The same pin can be repeated any number of times on input or output row, both for nameless and named inputs and outputs. If named pin is repeated more than once on the same row, then copy counters are appended to the corresponding pin names (see Figure 21). It is not required for repeated pins to be next to each other. Pin repetition is a visual implementation of a bond separation, input separation and output separation features.

Please note that the case of the same connection entering the same input ***N*** times (where ***N*** > 1) via pin repetition will multiply count of each input signal ***N*** times according to rules SDP-1 and SDP-6, meaning ***N*** copies of the same input signal will be enqueued at the input.

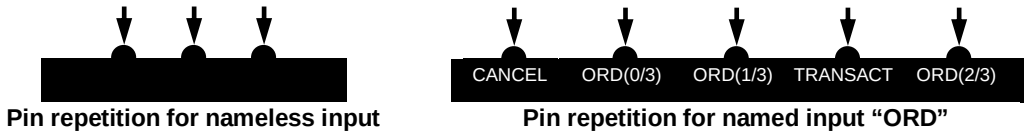


Figure 21. Pin repetition.

2.2.1.3.7. Pin Extenders

Rhombus vertices of inline runlets (see 2.2.2.1.3 “Inline Runlet”), arrow vertices of ports (see 2.2.2.4 “Ports”) and triangle vertices of beacons (see 2.2.2.1.1.3 “Beacon”) have a very limited space for attaching multiple connections, so **pin extender** can be used for attaching multiple connections to an inline runlet, beacon or port. Pin extender is a horizontal or vertical thick line (no less than double of the regular connection line thickness) going through a component vertex. All connections can be attached to extender line from the same or opposite side where component body is located (see the second example from the left on Figure 22). Extender lines should not intersect with each other, and with bottom name rectangle. Pin extender line does not necessarily need to be centered with a component, as long as it touches a vertex of the component body. Pin extender is not required in case of multiple connections attached to the bottom part of a member name rectangle (see the second example from the right in the Figure 22), but it can be attached to the bottom of the member name rectangle if it is too narrow to accommodate all connections (see the left-most example in the Figure 22).

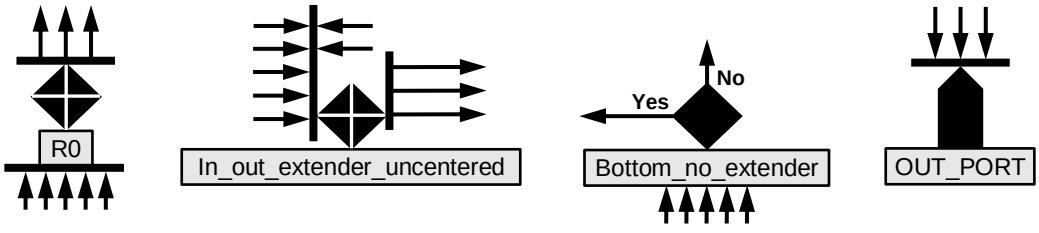


Figure 22. Pin extender.

2.2.1.3.8. Bond Attributes

Bond attributes for component pins are shown as rectangles with a black border and light-grey fill color (see Figure 23). All bond attribute letters are shown combined together inside of a rectangle (for example, “BW” combination of “B” and “W” letters). Bond attribute rectangle is not shown in case no letter is assigned to a bond attribute.

Letters used for encoding bond attributes are provided in Table 14. No letter is assigned to a bond attribute if a table cell in the column “Letter” of the table is empty. Bond scope means that a pin is allowed to have the attribute specified only for some bonds of a pin, while pin scope means the attribute must be applied to all bonds of a pin.

Table 14. Bond attribute encoding.

Attribute	Letter	Notes	Scope	Applicable to pin type
Memlet bond attributes				
Direct		Exactly one attribute must be specified for each bond of a pin.	Bond	Input
Push	P			
Write	W			
Read	R			
Broadcast	B	This attribute implicitly assumes presence of active bond attribute applied to the same bond. Broadcast attribute must also be combined with exactly one memlet bond attribute.		
Merger bond attributes				
Immunity	M	Combination of these attributes cannot be used together for the same pin.	Pin	Input
Purge	G		Bond	
General bond attributes				
Regular		Exactly one attribute must be specified for each bond of a pin.	Bond	Input
Front	F			
Back	K			
Clear	C			
Other bond attributes				
Global	L	This attribute must be combined with exactly one general bond attribute, excluding regular attribute.	Bond	Input
Tear	T		Bond	Input
Active	A	Internal signals can be produced only from outputs with this attribute.	Pin	Output
Staging	S	Caution is needed when attribute is used while continuously processing a single input signal in a loop.	Pin	Output

Bond attributes for inputs of outputs of components with a solid body and pin junctions are shown fully outside and adjacent to component body, between connection

and the body and with junction circles removed (see the left example in the Figure 23). Bond attributes for components without pin junctions are shown between vertex of the pin and connection line (see bond attribute “K” in the right example in the Figure 23). Bond attributes on pin extenders are shown fully outside and adjacent to extender line, between extender and connection lines (see bond attribute “C” in the right example in the Figure 23). Bond attributes for pins connected to name rectangles are shown fully outside and adjacent to name rectangle, between the bottom edge of the name rectangle and connection line (see bond attribute “F” in the right example in the Figure 23). Bond attribute rectangles for synclets and traplets are shown with their centers placed on boundary lines and connection lines going through behind bond attribute rectangles in case of synclets (see bond attributes “T” and “F” in the center example in the Figure 23). Traplet inputs and outputs should have pin junctions removed if bond attributes are used.

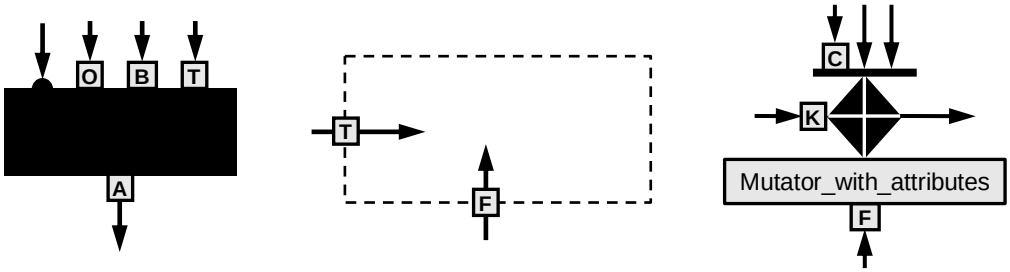


Figure 23. Bond attributes.

2.2.1.3.9. Badge

Components may have extra attributes shown as a letter combination inside of a rectangle with black border and light-grey fill color attached to the top-left corner of a component (see Figure 24).

Figure 24. Badge in top-left corner of a component.

Badges for inline runlets are shown as black rectangles with white text inside and attached to the left side of the component name rectangle (see Figure 25).

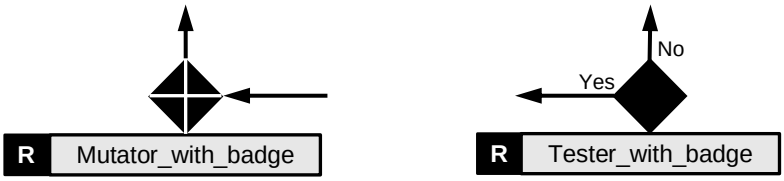


Figure 25. Badge for inline runlets.

Empty badges without letters are not shown. If multiple different attributes should be shown for a component in a badge, then letters must be shown in a multi-section badge with one letter per section, where each section is divided internally by vertical lines (see Figure 24). Multi-section badges are not used for inline runlets – all attribute letters are getting combined into a single letter combination in such case.

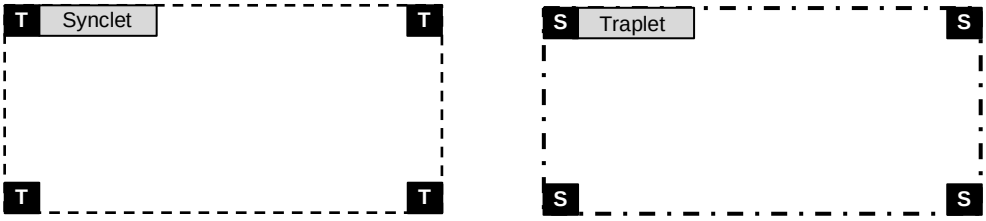


Figure 26. Badges for synclets and traplets.

Badges for synclets and traplets are shown as black rectangles with white text inside. Four copies of a badge should be placed in each corner of a synclet or traplet boundary rectangle. Component name label in top-left corner of a rectangle must be shifted to the right to accommodate horizontal space for a badge (see Figure 26).

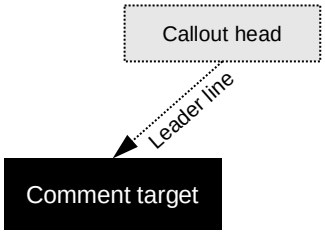


Figure 27. Comment elements.

2.2.1.3.10. Comments

2.2.1.3.10.1. Visual Representation

Comment is a descriptive text attached to visual element of a pipeline. Rectangular text container must be resized to fully accommodate text block plus reasonable margins (i.e., no horizontal or vertical scrolling are allowed inside of a rectangular text container).

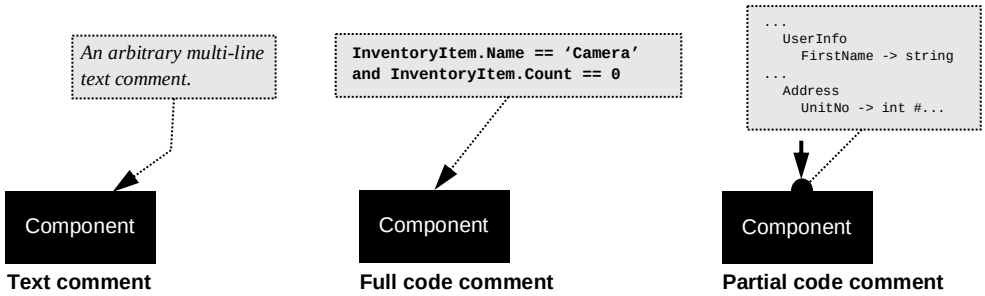


Figure 28. Comment types.

A comment consists of the following elements: **leader line**, **callout head** and **comment target**. Leader line is shown as a dotted line with arrow on the end attached to a specific location of a comment target and with a callout head on the other end. The callout head is a rectangular text container with a dotted border line and light-grey fill color (see Figure 27).

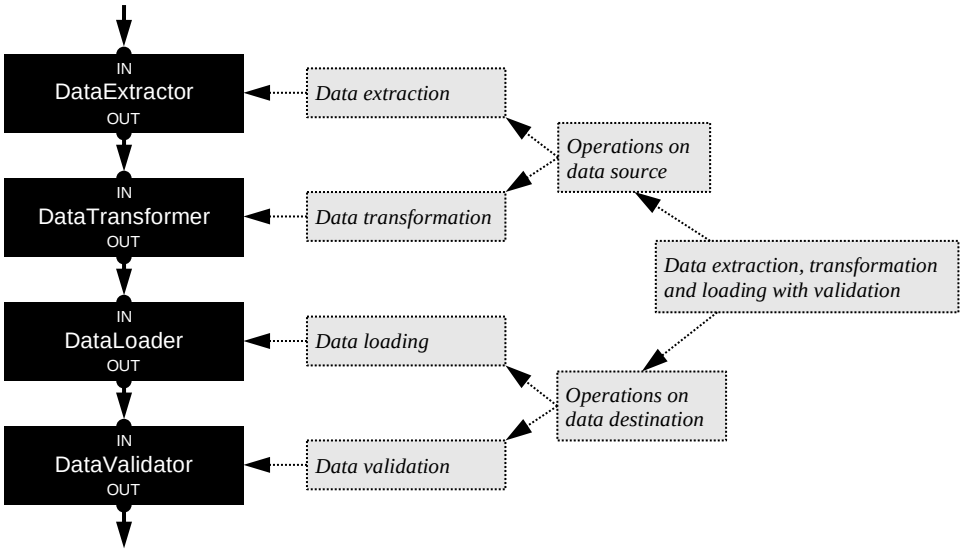


Figure 29. Multilevel comments.

2.2.1.3.10.2. Comment Target Types

The following comment targets are valid for attaching to a comment:

- Pipeline members.
- Component pins.
- Connections.
- Comments.

2.2.1.3.10.3. Comment Types

There are the following comment types (see Figure 28):

- **Text comment** – arbitrary text that must be shown using *Italic* font.
- **Full code comment** – complete Python source code fragment, traplet acceptance specification or a domain specification automatically extracted from a comment target and shown using `fixed-width` font (for example, Courier font). Traplet acceptance specifications and domain specifications are shown using regular version of the font, while Python code fragments – **bold** version.
- **Partial code comment** – same as full code comment, but it contains arbitrary partial fragments of a Python source code or a domain specification. Partial code comment may include only selected lines of code. Parts of a single line

can be excluded too. Skipped code lines above or below of a selected line and skipped parts inside of the selected lines must be shown by ellipses ("…") in all locations where code fragments are skipped.

Full and partial code comments are collectively called **code comments**. Code comments should not be available for direct editing, and they must be automatically extracted from the contents of the comment target. However, users should be allowed to select what lines and line fragments to show for a code comment. It is allowed to fallback from partial to full code comments if it gets impossible to determine technically what fragments should be skipped in partial comment after code change in a comment target.

Multiple comment targets per one comment are allowed for text comments, and for code comments referring to a common domain. Text comments can refer to other comments without limitations on number of levels of reference, but self-references are not allowed. Multiple comment targets feature and ability to reference other comments makes it possible to have multilevel comments (see Figure 29).

Leader line should not have arrow in case of a component pin comment. Pin junctions must be used to attach pin comments for component that support pin junctions. All components without pin junctions (inline runlets, ports and beacons) are allowed to have pin comments attached directly to the connection point: vertex of a component shape, or connection point on a pin extender (see Figure 30). It is allowed to attach component comments to a name rectangle of a component.

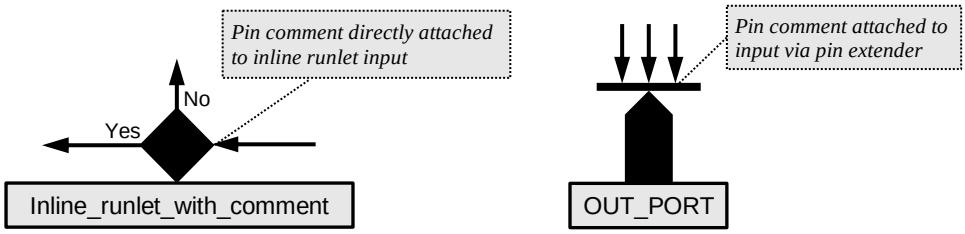


Figure 30. Pin comments usage without pin junctions.

2.2.1.3.10.4. Code Comment Targets

Text comment can be attached to any eligible comment target type. However, code comments are limited to the following targets (a choice to show Python code or domain specification should be provided in case of inline runlets):

- **Prime runlets** – source code is extracted for a comment if it is possible, or manifest of executable artifact otherwise.
- **Scripted and inline runlets** – Python code is extracted for a comment.
- **Memlets** – domain specification of memlet is extracted for a comment.
- **Connections** – domain specification of overlap is extracted for a comment.
- **Component pins** – domain specification of a domain attached to the component explicitly, derived from other domains or auto-assigned to a pin is extracted for a comment.

- **Inline runlets and ports** – domain specification of a domain defined for a component is extracted for a comment. Comment for inline runlet can contain either Python code or domain specification.
- **Traplets** – acceptance specification is extracted for a comment.

2.2.2. Components

2.2.2.1. Runlets

2.2.2.1.1. Fixed Runlets

2.2.2.1.1.1. Merger

Merger component is shown as a black upside-down trapezoid with white internal text. The component has vertically and horizontally centered name inside, a row of named inputs along the top edge and nameless output at the bottom edge (see Figure 31).

Merger type is shown inside of a badge. The following letters are used for indicating merger type:

- **Naming merger** – letter “N”.
- **Priority merger** – letter “P”.
- **Exclusion merger** – letter “E”.
- **Strict merger** – letter “S”.

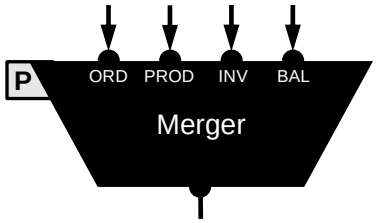


Figure 31. Merger.

2.2.2.1.1.2. Transformer

Transformer component is shown as a black horizontally elongated hexagon with white internal text. The component has vertically and horizontally centered name inside, nameless input at the top edge and nameless output at the bottom edge (see Figure 32).

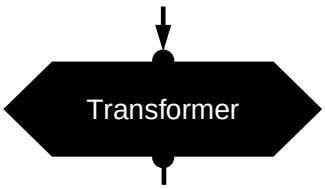


Figure 32. Transformer.

2.2.2.1.1.3. Beacon

Startup beacon is shown as a black upside-down rectangle with a nameless output attached to the bottom vertex of the rectangle. Shutdown beacon is shown as an upside rectangle with a nameless output at the center of the base of the rectangle. Pin junctions are not used for beacons (see Figure 33).

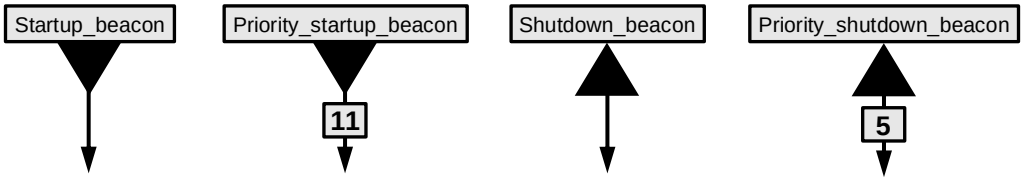


Figure 33. Startup and shutdown beacon.

Member name is shown as a black text in a rectangle with a black border and light-grey fill color. The name rectangle is located at the top of the beacon horizontally centered with the component shape. If beacon has a priority, then it is shown in a rectangle with a black border and light-grey fill color under the beacon on an output connection line, horizontally centered with the connection line (see Figure 33).

2.2.2.1.2. Reusable Runlet

Reusable runlet is shown as a black rectangle with white internal text. The component has vertically and horizontally centered name inside, a row of named inputs along the top edge and row of named outputs along the bottom edge (see Figure 34).

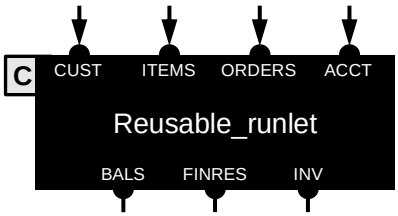


Figure 34. Reusable runlet.

Reusable runlet type is shown inside of a badge. The following letters are used for indicating runlet type:

- **Composite runlet** – letter “C”.
- **Scripted runlet** – letter “S”.
- **Prime runlet** – letter “P”.

2.2.2.1.3. Inline Runlet

Inline runlet is shown as a black rhombus. Mutator also has two white lines inside connecting opposite vertices. Inline runlets are allowed to have an input and an output connected to any vertex. Each vertex of tester must play a single role of only an input or

output, i.e., mixing inputs and outputs in one tester vertex are not allowed. Input/output separation rules are applicable to inputs and outputs connected to the same or different vertices of an inline runlet. Pin junctions are not used for inline runlets (see Figure 35).

Member name is shown as a black text in a rectangle with a black border and light-grey fill color, located at the bottom of the inline runlet centered horizontally with the component shape. Inputs and outputs can be connected to an inline runlet via a bottom side of a member name rectangle (see Figure 35).

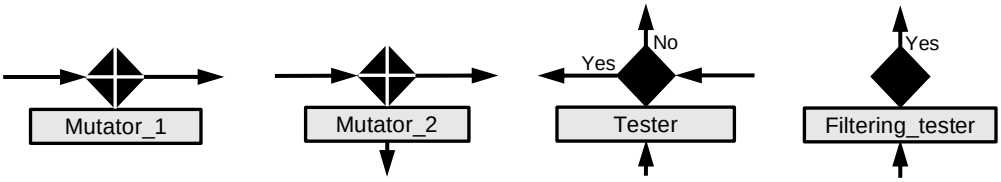


Figure 35. Inline runlet.

Mutator's inputs and outputs are nameless. Tester has nameless input and one or two named outputs with labels **YES** and **NO** shown near corresponding vertex outside of the shape (see Figure 35).

2.2.2.2. Memlet

Memlet is shown as a black parallelogram with a positive skew angle (angle of a top-left corner is greater than 90 degrees) and white text inside. The component has a nameless input at the top edge and nameless output at the bottom edge. Its internal area has a horizontal white line dividing the memlet shape into two equal parts. The top part contains member name of the memlet. The bottom part contains the name of a membank connected to the memlet with a font size smaller than the font size of the memlet name at the top part. Regular writable memlet has no badge, unless a writable memlet has some other letters to be shown in a badge – letter "W" is shown in the left-most section of the multi-section badge in such case (see the left-most example in the Figure 24). Read-only memlet has a badge with a capital "R" letter. Two-letter combination "RR" is shown for a memlet connected to read-only membank. Membank name must use copy counters in case more than one memlet is present on diagram for the same membank (see Figure 36).

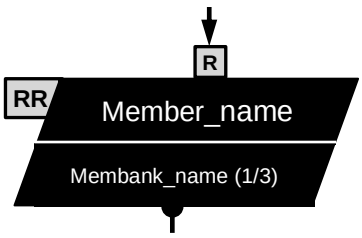


Figure 36. Memlet.

2.2.2.3. Membank

Membank is shown as parallelogram with the same corner angles as for memlet shapes. It has black border, light-grey fill color and black text inside displaying membank name.

No inputs and output connections are allowed. Only membanks with more than one memlet in a pipeline should be shown on diagrams. Membank badge uses black fill color and white text. The only attribute that can be shown for membanks is a read-only attribute as letter “R”. Memlets are connected to a membank using a mid-grey zigzag line called **memlink** (see Figure 37). Memlink lines are not constrained by only vertical and horizontal segments like connections, i.e., memlinks can have diagonal segments.

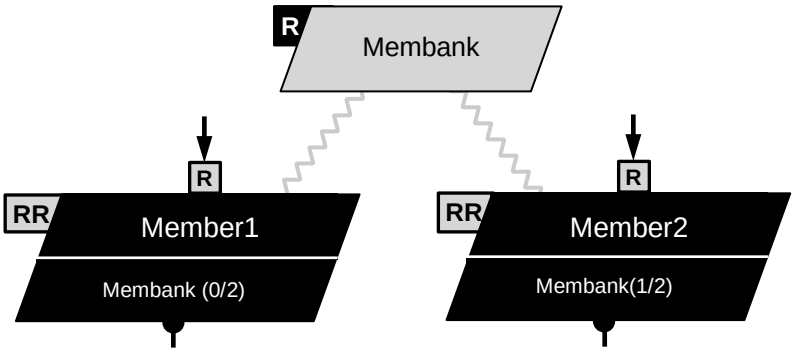


Figure 37. Membank.

2.2.2.4. Ports

Ports are shown as a pentagonal black arrow shape. Input ports are shown with arrow direction down, and output ports – with arrow direction up. Port inputs and outputs are located at the tip of the arrow shape. Pin junctions are not used for ports. Port name is shown as a black text with a black border and light-grey fill color in a rectangle horizontally centered with the component shape at the top of shape for input ports, and at the bottom – for output ports (see Figure 38). More than one copy of the same port may exist in one pipeline, so copy counters must be used for ports in such case (see the left example in the Figure 38). If output port is connected to active output, then a bond attribute rectangle with letter “A” is shown for such output (see the right example in the Figure 38).

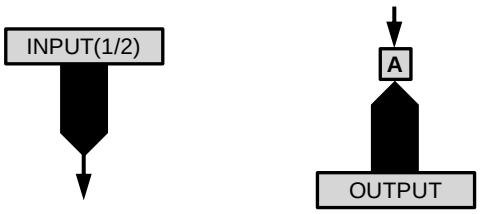


Figure 38. Input and output ports.

Port name and assigned domain cannot be changed for ports directly. The name or domain of a regular port can be changed by modifying name or domain of a corresponding pin connected to that port. Change rules for system port name/domain are outside of the scope of this specification and each Pipe implementation is free to introduce its own rules regarding system ports.

Reset input and output ports are shown with the same shapes as corresponding non-reset port types, but with white lines connecting diagonal corners of the port shape. Port names are not shown for input and output reset ports (see Figure 39).

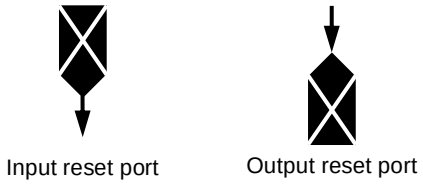


Figure 39. Input and output reset ports.

2.2.2.5. Synclet

Synclet is shown as a rectangle with a black dash line and no background color. Synclet name must be shown at the top-left corner in a rectangle with a black solid border, black text, and light-grey fill color (see Figure 40).

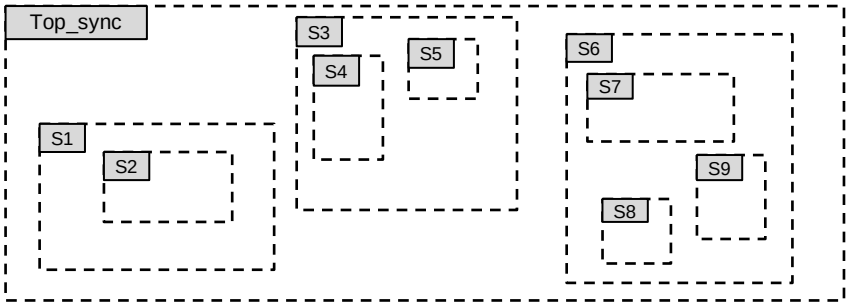


Figure 40. Nested synclets.

2.2.2.6. Traplet

Traplet is shown as a rectangle with dash-dot line border and no background color. Traplet inputs and outputs are shown as junction black circles placed on any side of a traplet rectangle with circle centers at the boundary line. Traplet name must be shown at the top-left corner in a rectangle with a black solid border, black text, and light-grey fill color (see Figure 41).

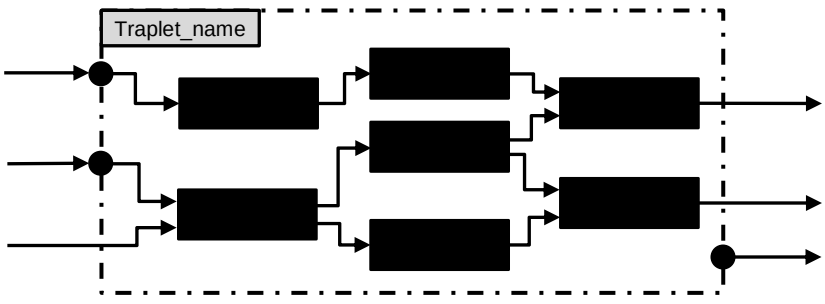


Figure 41. Traplet.

2.2.2.7. Connector

Connector is a visual symbol that creates a break in the depiction of a connection or memlink within a diagram, while preserving the underlying logical (topological) continuity between elements. Each point where the connection or memlink is broken is called **anchor**. Connector is a set of two or more anchors sharing the same identifier.

Connector links distant locations within a single pipeline. Connectors are labeled by matching identifiers assigned to each of their anchor automatically without allowing for modifications. Connector identifier is prefixed by letter "C" followed by a unique number of a connector within a pipeline starting from zero (for example: **C5**, **C8**, **C23**).

Connector is shown as two black circles touching each other with white text inside, where text in one circle contains connector identifier and text in other circle – copy counter (see Figure 42). Circles can be arranged vertically one under another – top circle shows connector identifier in this case (see the left-most example in Figure 42). Circles can also be arranged horizontally with left circle showing connector identifier (see the second example from the left in Figure 42).

Any number of inbound and/or outbound connections or memlinks are allowed for a connector. Connections can enter or exit a connector from the top, left, bottom and right side of any circle, except a point where one circle touches another. Memlinks can be attached to connector circles anywhere except a point where one circle touches another (see the second example from the right in Figure 42). Pin extenders can be attached to top, left, bottom and right side of any circle except a point where one circle touches another (see the right-most example in Figure 42).

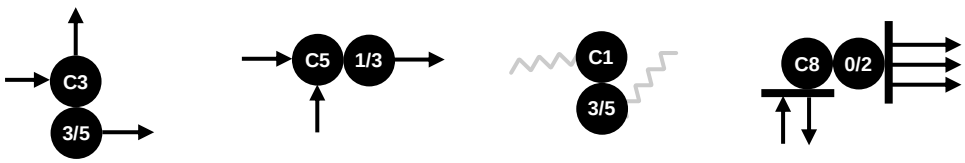


Figure 42. Connector.

2.2.2.8. Component State

2.2.2.8.1. Component State Overrides

Badges with the following letters are used to indicate different component state overrides for respective components:

- **State retention** – letter "R".
- **Self-reset** – letter "S".
- **Parent reset** – letter "P".
- **Top reset** – letter "T".

Letter combinations are used if more than one state override is applied to a component. For example, letter combination "PS" is used for indicating that both parent reset and self-reset state overrides are applied to the component.

As memlets, mergers and runlets already use badges to indicate their types, component state override uses a new second badge section to the right of the component type section (see Figure 24).

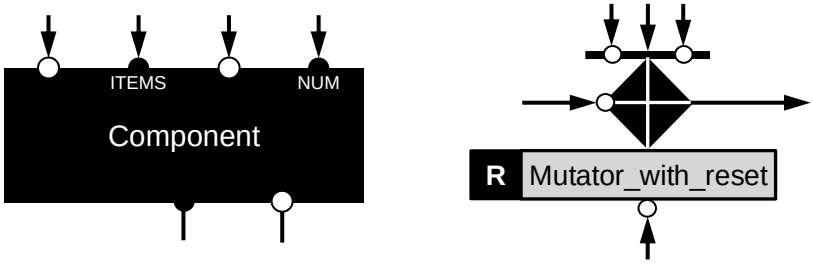


Figure 43. Reset pins.

2.2.2.8.2. Reset Pins

Reset inputs and outputs represent explicit state reset override and they are called collectively **reset pins**. Reset pin is shown as a white circle with black border replacing pin junction at the same location and with the same radius as replaced pin junction. Repetition of a reset pin is allowed. Reset pins are always nameless, even for components that have names assigned for all other pins (see the left example in the Figure 43).

Reset pins can be attached to vertices, pin extenders and name boxes of inline runlets. If reset pin is attached to a vertex or pin extender, then the circle of the reset input is centered on vertex or pin extender line respectively. If reset pin is attached to name box, then the reset pin circle should be shown fully outside of the box, only touching the name rectangle (see the right example in the Figure 43).

If reset pin is used together with bond attributes, then reset pin circle stays in the same place and the bond attribute rectangle is placed between the circle and connection line (see Figure 44).

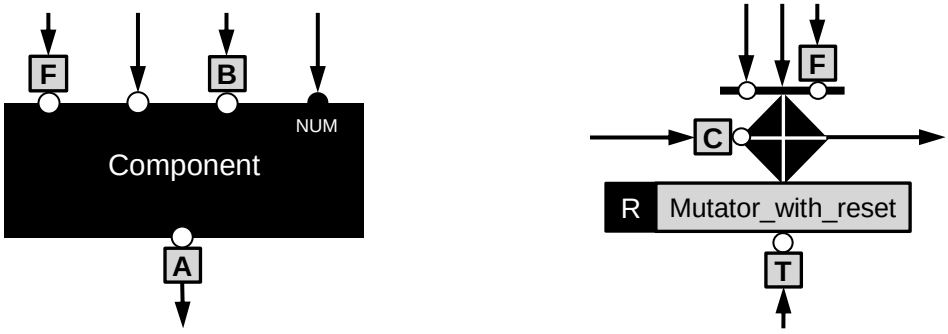


Figure 44. Reset pin with bond attributes.

2.3. Levels of Usage

2.3.1. Levels of Usage

Pipe language is designed in such a way that citizen developers do not need to comprehend the complete Pipe specification to start development, because the language provides several **levels of usage**. The lower level of usage is, the lower Pipe skill set requirements are. However, language constraints and limitations are increasing with each lower level of usage due to absence of certain Pipe features: the lower level of usage is, the more features become unavailable. There are the following levels of usage in Pipe:

- Basic level.
- Intermediate level.
- Advanced level.
- Professional level.

2.3.2. Basic Level of Usage

Basic level of usage has the following Pipe language skill set requirements:

- Usage of the following elements:
  - Reusable runlets.
  - Mergers (with default option not requiring knowing details of domain merge process).
  - Memlets and membanks.
  - Ports.
  - Beacons.
  - Synclets.
  - Traplets.
- Creating pipelines by adding and connecting components.
- Creating libraries and applications.

Basic level skill set does not require knowledge of Pipe domains or Python scripting language. As a result, citizen developers will not be able to reconcile component pins if they are not connectable and therefore, developers will not be able to connect components requiring domain transformations, especially components from different providers where the need of domain reconciliations appears very often. However, basic level still allows connecting components of the same provider if detailed documentation is provided explaining correct cases of connections between the components: non-connectable component pins would mean they are not supposed to be connected by design, but connectable pins might not automatically mean the connection is logically correct (runtime exception would probably need to be thrown in case of incorrectly connected pins).

Domains can still be referred on basic level by simple names, without revealing internal structures of domains. For example, domains “Account” can be referred as output

of a runlet or memlet, assuming that outputs with domain “Account” can be connected only with inputs with domain “Account”. Merging rules can also be established for domains. For example, a rule can be created stating that merging domains “PersonInfo” and “Balance” produces domain “Account”, and output of such mergers can be connected only to inputs expecting “Account” domain.

Another limitation of this level of usage is that the citizen developers are not able to create coded runlets.

2.3.3. Intermediate Level of Usage

Intermediate level of usage requires knowledge of Pipe domains on top of a skill set for a basic level of usage. Citizen developers for this level of usage can use transformers to reconcile domains of non-connectable pins. As a result, they can connect components from different providers where frequent reconciliations of domains from different providers are needed. However, citizen developers of this level do not have skills to create coded runlets.

2.3.4. Advanced Level of Usage

Advanced level of usage requires knowledge of a Python language on top of a skill set for an intermediate level of usage. Citizen developers of this level of usage can create Python-based runlets (scripted and inline runlets), but they still cannot create prime runlets. Advanced level of usage assumes knowledge of the complete Pipe specification.

2.3.5. Professional Level of Usage

Professional level of usage requires prime runlet development skills on top of a skill set for an advanced level of usage. It is enough to know just one prime language to qualify for this level of usage. Professional level combines roles of a citizen and professional developers.

2.3.6. Non-Coded Testers

Tester component in Pipe represents conditional execution and it is a key component for implementing almost any algorithm. However, tester components require knowledge of Python programming language and therefore, testers are not available for basic and intermediate levels of usage. It does not mean algorithm construction abilities are significantly limited for these levels of usage due to the coding nature of testers in Pipe. It is possible to create a generic reusable runlet with two outputs (one representing “Yes” output and another – “No” output) and with input domain describing boolean expression tree. This idea of a non-coded tester with input domain representing arbitrary boolean expression tree is discussed as a part of Pipe math library (see 3.2.3.18.4 “Math”).

Unfortunately, non-coded testers are available for users starting from intermediate level because it requires knowledge of internal domain structures. However, non-generic testers can be created as runlets with a single input and two outputs – “Yes” and “No”.

For example, reusable runlet can be created checking if account is open, taking named domain “Account” as input and generating result on one of outputs depending on account status (open or closed). Basic level users will be able to consume such non-generic testers to verify account statuses.

2.4. Coded Runlets

2.4.1. Runlet Parametrization

Runlet parametrization is a feature allowing a template runlet to have associated set of parameters by attaching a domain to a template runlet. Data objects representing that domain are automatically attached to all instance runlets created from the template runlet, but values of data objects are assigned separately for each instance runlet. The data object is modifiable only during design-time and available for reading inside of the instance runlet during runtime. One example where such feature might be useful is the case of a template runlet representing UI components (Label, TextBox, DropDown, List, etc.). Domain nodes would correspond to visual properties of UI components (font name/size, color, etc.) and component’s positional coordinates on a dialog form. Special drag-and-drop visual UI designer may provide abilities to change visual properties of UI components, and those properties are going to be synched up with data objects of the corresponding instance runlets representing UI component.

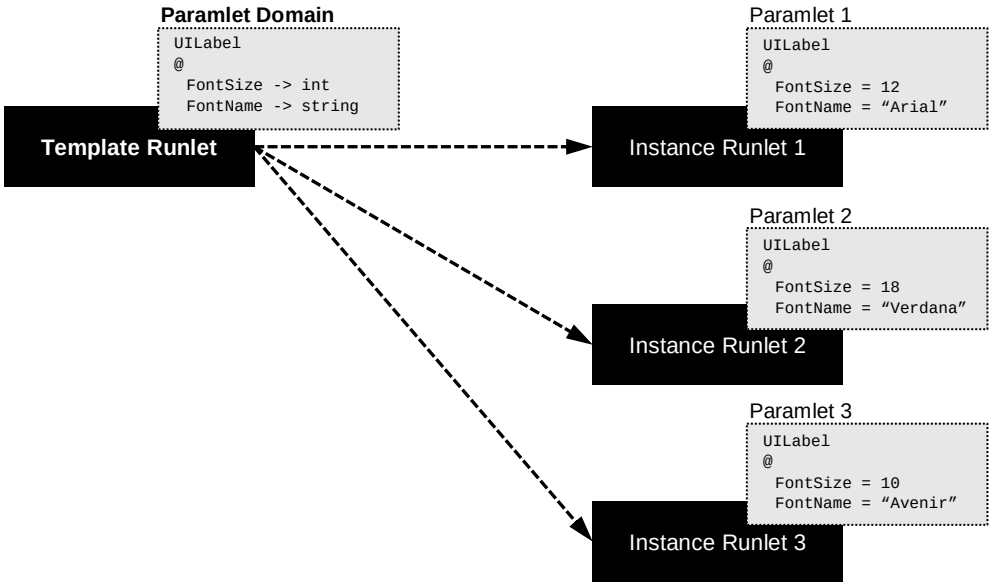


Figure 45. Runlet parametrization.

Runlet parametrization feature is implemented in Pipe by assigning a domain (called **paramlet domain**) to a coded template runlet. Paramlet domain can be defined in-place, or reference to an existing domain can be used for this purpose. Only nullable, default,

constant and optional node attributes are allowed to be used in a paramlet domains. Runlet parametrization usage must follow these rules (see Figure 45):

- Data object (called **paramlet**) gets created automatically from a paramlet domain and getting attached to an instance runlet during its instantiation from a template runlet.
- Paramlet values for storage nodes with a primitive data type or primitive base type collection can be modified during pipeline construction in a special visual designer inside of Pipe development environment individually for each instance runlet. Storage nodes with domain data type must be domain-expanded when paramlet is open for editing in a visual designer.
- The code inside of a runlet can read paramlet data during runtime, but it cannot modify the structure of the paramlet or its data contents.
- Adding paramlet domain to a template runlet, modifications of a paramlet domain structure and paramlet domain removal from a template runlet should automatically propagate to all corresponding instance runlets. New type of code comments (object specifications of attached paramlets) must be used to show paramlet contents on diagrams.

2.4.2. Exception Handling

When non-visual language **native exception** is thrown inside of the code and not caught inside of the same prime/scripted runlet, it always terminates Pipe application no matter what traplets are placed in the application. This is because native exceptions cannot be intercepted by Pipe. **Pipe exceptions** are completely different from native exceptions of any programming language. Only Pipe exceptions can be intercepted by traplets. To intercept native exceptions using traplets, native exception needs to be caught in the same prime or scripted runlet where it is thrown, converted to a Pipe exception, and then re-thrown as Pipe exception using runlet API (see 2.5 “Runlet API”).

2.4.3. Global Context

Python runlets cannot expose any global declarations (classes, methods, global variables, etc.) outside of a runlet containing the declaration. Only a special dedicated Python code area called **global context** attached to a solution may contain declarations available anywhere in the solution. Global context must contain only Python declarations available to all scripts inside of a solution and it is implemented as a special Python module with name ***global-context-declarations***. This module is an integral part of the solution used by internal Python code components and therefore, it also must be imported to other solutions as part of imported library. However, the global context declarations should not be accessible by Python scripts from any component outside of an imported solution.

2.5. Runlet API

2.5.1. Introduction

Runlet API is a programming interface that must be used by coded runlets. The interface provides interaction capabilities between a coded runlet and Pipe environment, such as receiving incoming input signals for processing and sending output signals back.

The specification provides only Python version of API, but an equivalent interface must be provided by any programming language brought to Pipe environment as a prime language.

2.5.2. API Specification

2.5.2.1. Enums

Enum **NodeAttributes**

Description

This enumeration provides flags to describe full combination of attributes assigned to a domain node.

Specification

```
class NodeAttributes(Flag):
    DEFAULT = 1             # Default attribute.
    CONSTANT = 2            # Constant attribute.
    NULLABLE = 4            # Nullable attribute.
    OPTIONAL = 8            # Optional attribute.
    MANDATORY = 16          # Mandatory attribute.
    MANDATORY_SRC = 32      # Mandatory attribute with source
                            # modifier.
    MANDATORY_DEST = 64     # Mandatory attribute with destination
                            # modifier.
    FORBIDDEN = 128         # Forbidden attribute.
    FORBIDDEN_SRC = 256     # Forbidden attribute with source
                            # modifier.
    FORBIDDEN_DEST = 512    # Forbidden attribute with destination
                            # modifier.
    INJECTION = 1024        # Injection attribute.
    INJECTION_SRC = 2048    # Injection attribute with source
                            # modifier.
    INJECTION_DEST = 4096   # Injection attribute with destination
                            # modifier.
    REJECTION = 8192        # Rejection attribute.
    REJECTION_SRC = 16384   # Rejection attribute with source
                            # modifier.
    REJECTION_DEST = 32768  # Rejection attribute with destination
                            # modifier.
```

Enum **PrimitiveDataType**

Description

This enumeration provides constants to specify a primitive data type for a storage node value.

Specification

```
class PrimitiveDataType(Enum):
    STRING = 0              # String data type
    BOOL = 1                # Bool data type
    INT = 2                 # Integer data type
    FLOAT = 3               # Float data type
    DATETIME = 4            # Datetime data type
    BINARY = 5              # Binary data type
    ANY = 6                 # Any data type
```

2.5.2.2. Domains

Class **Domain**

Description

This class represents a domain created in the current solution or imported to the solution. The class provides read-only access to the domain, i.e., it is not possible to create or change domain programmatically. Any attempt to create an instance of the class must throw an exception. Static method **get_domain** of this class must be used to get instances of the class.

Specification

Properties

- **original_name: str**
  - **Access:** Read-only
  - **Description:** Original full name of the domain if a synonym is assigned to the domain, or **None** in case of an anonymous domain or when no synonym is assigned to the domain.
- **name: str**
  - **Access:** Read-only
  - **Description:** Original full name of the domain, or the full name of a synonym if it is assigned to the domain. The property contains **None** in case of an anonymous domain.

Class **Domain**

Methods

- **get_domain(name: str) -> Domain**
  - **Type:** Static
  - **Parameters:**
    - ✓ **name** – full name of the domain or full name of a synonym if it is assigned to the domain.
  - **Description:** The method returns object of class **Domain** representing the domain with the name specified in the parameter **name**. The method returns **None** if there is no domain with a name specified in the parameter **name**, or the parameter **name** contains invalid value.
- **get_domain_node(path: str) -> DomainNode**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **path** – full domain path to the node.
  - **Description:** The method returns object of class **DomainNode** representing the domain node at the domain path specified in the parameter **path**. The method returns **None** if there is no domain node at the path specified in the parameter **path**, or the parameter **path** contains a malformed path.
- **create_data_object() -> DataObject**
  - **Type:** Instance
  - **Parameters:** None
  - **Description:** The method creates and returns an object of class **DataObject** representing new default data object conforming to the domain specification of the current domain. Returned instance of class **DataObject** is a default data object contains all nodes according to the corresponding domain specification, including nodes with optional attribute (property **missing** is set to **True** for such nodes).

Class **DomainNode**

Description

This class represents node as a part of a domain specification. The class provides read-only access to a domain node, i.e., it is not possible to create or change domain node programmatically. Any attempt to create an instance of the class must throw an exception. Method **get_domain_node** of class **Domain** must be used to get instances of this class. If this node comes from any base domain of a corresponding domain, then it is assumed to be a node redefinition or inheritance override. If data type is also defined for such node, then it is assumed to be a forced node redefinition for the data type.

Specification

Properties

- **name: str**
  - **Access:** Read-only
  - **Description:** Name of the domain node, or full node path in case of an inheritance override.

Class **DomainNode**

- **parent: DomainNode**
  - **Access:** Read-only
  - **Description:** Parent domain node. The property contains **None** in case of root node.
- **domain: Domain**
  - **Access:** Read-only
  - **Description:** Parent domain containing this node directly or via inheritance.
- **base_domains: list**
  - **Access:** Read-only
  - **Description:** List of base domains the node is inherited from. The list contains instances of class **Domain**.
- **attributes: NodeAttributes**
  - **Access:** Read-only
  - **Description:** A combination of attributes directly or indirectly assigned to the node.
- **forced_attributes: NodeAttributes**
  - **Access:** Read-only
  - **Description:** A combination of attributes excluded from the node if it comes from a base domain.
- **init_value: object**
  - **Access:** Read-only
  - **Description:** Default or constant value if the node has default or constant value, or **None** otherwise.
- **inheritance_override: list**
  - **Access:** Read-only
  - **Description:** List of domains from the base domain list participating in inheritance merge for this node, or **None** if inheritance override is not used. The list contains instances of class **Domain**.
- **children: list**
  - **Access:** Read-only
  - **Description:** Collection containing instances of class **DomainNode** representing children slice.
- **data_type: PrimitiveDataType**
  - **Access:** Read-only
  - **Description:** Primitive data type assigned to the storage node, or **None** in case of a non-storage node, or when the node comes from a base domain and there is no forced redefinition for a data type.
- **ref_domain: Domain**
  - **Access:** Read-only
  - **Description:** Reference domain of a domain data type, or **None** if this is not a domain data type.
- **dim_count: int**
  - **Access:** Read-only
    Description: The number of array dimensions (number of nested array levels plus one) starting from **1 (one)** for a one-dimensional non-nested array, or **0 (zero)** for non-collection data type.

2.5.2.3. Data Objects

Class **DataObject**

Description

This class represents a data object conforming to the structure of some domain. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Method **create_data_object** of the class **Domain** must be used to create new instances of this class.

Specification

Properties

- **domain: Domain**
  - **Access:** Read-only
  - **Description:** Domain specification this data object conforms to.

Methods

- **get_node(path: str) -> DataObjectNode**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **path** – full domain path to the data object node.
  - **Description:** The method returns instance of class **DataObjectNode.** Value **None** is returned if the path specified in a parameter **path** does not match any node in corresponding domain specification, or the parameter **path** contains a malformed path.

Class **DataObjectNode**

Description

This class represents a domain node instance in a context of specific data object. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Method **get_node** of the class **DataObject** must be used to create new instances of this class.

Specification

Properties

- **domain_node: DomainNode**
  - **Access:** Read-only
  - **Description:** Corresponding node from domain specification.
- **parent: DataObjectNode**
  - **Access:** Read-only
  - **Description:** Parent domain node instance containing this node. The property contains **None** in case of a root node.

Class **DataObjectNode**

- **data_object: DataObject**
  - **Access:** Read-only
  - **Description:** Parent data object containing this node.
- **children: list**
  - **Access:** Read-only
  - **Description:** Collection containing instances of the class **DataObjectNode** representing children slice of the node.
- **missing: bool**
  - **Access:** Read-write
  - **Description:** Flag indicating if the domain node instance with optional attribute is missing in a data object. An exception is thrown when assigning value **True** to the property of a node that does not have an optional attribute assigned to this node in domain specification.

Methods

- **get_value() -> object**
  - **Type:** Instance
  - **Parameters:** None
  - **Description:** The method returns value of the node with a Python data type corresponding to a data type of the node. The method returns instance of the class **NullObject** if value is null. The method returns **None** if the node does not have a value slice, or property **missing** is **True.**
- **set_value(value: object) -> bool**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **value** – new value of the storage node. An argument for this parameter must have proper data type according to the data type of the storage node (for mapping between Pipe primitive data types and Python data types see 2.5.3 “Primitive Data Type Mapping”). Value **NullObject** must be provided to set value as a null object or null value. Value **None** is not allowed. Property **missing** sets to **False** automatically after successful call of this method.
  - **Description:** The method sets value specified in a parameter **value**, returning **True** if the value was changed successfully, or **False** otherwise.

2.5.2.4. Signal Processing

Class **Runlet**

Description

This class represents a prime, scripted, or inline instance runlet containing currently running code. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Static method **get_runlet_object** of this class must be used to get a shared instance of the class.

Class **Runlet**

Specification

Properties

- **original_name: str**
  - **Access:** Read-only
  - **Description:** Original full name of the template runlet if a synonym is assigned to the runlet, or **None** otherwise.
- **name: str**
  - **Access:** Read-only
  - **Description:** Original full name of the template runlet, or full name of a synonym if it is assigned to the runlet.
- **member_name: str**
  - **Access:** Read-only
  - **Description:** Member name of this runlet in a pipeline.
- **path: str**
  - **Access:** Read-only
  - **Description:** Full runlet path not including this runlet member name.
- **inputs: list**
  - **Access:** Read-only
  - **Description:** Collection containing instances of the class **RunletPin** representing input pins of this runlet.
- **outputs: list**
  - **Access:** Read-only
  - **Description:** Collection containing instances of the class **RunletPin** representing output pins of this runlet.
- **paramlet: DataObject**
  - **Access:** Read-only
  - **Description:** Paramlet of the instance runlet. The property contains **None** in case the corresponding template runlet is not parametrized.

Methods

- **get_runlet_object()**
  - **Type:** Static
  - **Parameters:** None
  - **Description:** The method returns a shared instance of class **Runlet** representing current runlet.

Class **Runlet**

- **get_input_signal(return_immediately: bool = True) -> InputSignal**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **return_immediately** – flag indicating if the method should return immediately if there is no input signal waiting for processing (**True**), or the method must keep waiting for an input signal until one arrives (**False**). The default value is **True**.
  - **Description:** The method fetches the next input signal from the queue if there is any input signal waiting for processing and returns an instance of class **InputSignal**. The method waits for arrival of an input signal if the parameter **return_immediately** is **False,** or it returns **None** immediately if the parameter **return_immediately** is **True** and no input signal is waiting for processing. The method returns **None** immediately and irrelevant of the value of parameter **return_immediately** in case the application is not currently running. If this method returns an instance of class **InputSignal** then it opens a new runlet-scoped **transaction**. It means the second call of this method would throw an exception if method **finish_processing** of a returned object of the class **InputSignal** is not called first to close already open transaction before calling this method to open a new transaction for this runlet.
- **fire_internal_signal(data_object: DataObject, output: RunletPin, priority: int = None, tag: str = None)**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **data_object** – prepared data object to be attached to an enqueued internal signal. Instance of the class **NullObject** must be provided to fire a null object. Value **None** must be provided in case of a blank signal.
    - ✓ **output** – output to fire internal signal from. The output must have active attribute specified.
    - ✓ **priority** – optional priority value to fire the signal with.
    - ✓ **tag** – optional tag value to distinguish between different sources of data objects with the same domain: the tag helps to determine where propagation of internal signal begins inside of the parent runlet.
  - **Description:** The method fires new internal signal with a data object specified in the parameter **data_object**. Created internal signal gets enqueued in the parent composite runlet, waiting for processing until it is fetched from the queue after calling the method **get_input_signal** of the parent runlet. This method is not allowed for inline runlets and for root runlets, and any attempt to call the method in context of an inline runlet or root runlet should throw an exception.

Class **RunletPin**

Description

This class represents an input or output pin of the runlet containing currently running code. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Properties **inputs** or **outputs** of the class **Runlet** must be used to get input or output pins respectively.

Class **RunletPin**

Specification

Properties

- **name: str**
  - **Access:** Read-only
  - **Description:** Name of the pin. The property contains **None** if the pin is nameless.
- **domain: Domain**
  - **Access:** Read-only
  - **Description:** Domain assigned to this pin. The property contains **None** if the pin is domainless.
- **nullable: bool**
  - **Access:** Read-only
  - **Description:** Flag indicating if domain assignment to this pin is nullable.
- **active: bool**
  - **Access:** Read-only
  - **Description:** Flag indicating if this pin is an output with active attribute.

Class **InputSignal**

Description

This class represents a signal accepted by the runlet for processing. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Method **get_input_signal** of the class **Runlet** must be used to get instances of this class.

Specification

Properties

- **input: RunletPin**
  - **Access:** Read-only
  - **Description:** Input pin accepted this signal. The property contains **None** in case of an internal signal.
- **data_object: DataObject**
  - **Access:** Read-only
  - **Description:** Data object attached to the arrived signal. The property contains instance of the class **NullObject** if data object is null. The property contains **None** if the signal is blank.
- **internal: bool**
  - **Access:** Read-only
  - **Description:** Flag indicating if this is an internal signal.
- **tag: str**
  - **Access:** Read-only
  - **Description:** Tag value passed to the method **fire_internal_signal** of the class **Runlet** in case of an internal signal, or **None** if tag is not provided or this signal is not an internal signal.

Class **InputSignal**

Methods

- **send_output(output: RunletPin, data_object: DataObject, priority: int = None) -> bool**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **output** – output pin receiving the data object specified in the parameter **data_object**. The pin must be an output pin belonging to the runlet executing this code.
    - ✓ **data_object** – data object to be sent to an output pin specified in the parameter **output**. Value **None** must be provided for this parameter in case a blank signal has to be sent to the output pin. Instance of class **NullObject** must be provided in case null object has be sent to the output pin (the pin must have a nullable domain assignment to be able to accept null objects). Data object must conform to domain of the output pin specified in the parameter **output**. Sending the same data object to the same or different output pins multiple times is allowed, and it is equivalent to sending multiple separate data objects to outputs, except a tester that must send no more than one output signal for each input signal. Therefore, a tester must throw exception when the same data object is sent more than one time.
    - ✓ **priority** – optional priority value to send the signal to the output with.
  - **Description:** The method sends an output signal with a data object specified in the parameter **data_object** to an output pin specified in the parameter **output**. The method returns **True** if an output signal was sent successfully, or **False** otherwise.
- **finish_processing()**
  - **Type:** Instance
  - **Parameters:** None
  - **Description:** The method closes a transaction that was open when this object was returned from the method **get_input_signal** of the class **Runlet**. Input signal object must not be used after calling this method. Any attempt to access any property or method of the input signal object after calling this method must throw an exception.

Class **NullObject**

Description

This class represents a null object associated with a domain. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Static method **create_null_object** of this class must be used to create new instances of the class.

Specification

Properties

- **domain: Domain**
  - **Access:** Read-only
  - **Description:** Domain associated with a null object. The value for this property cannot be **None**.

Class **NullObject**

Methods

- **create_null_object(domain: Domain) -> NullObject**
  - **Type:** Static
  - **Parameters:**
    - ✓ **domain** – domain of the data object represented by this null object.
  - **Description:** The method returns new instance of a null object associated with the domain specified in the parameter **domain**.

2.5.2.5. Execution Context

Class **EntryPoint**

Description

This class contains two methods - exactly one of them must be overridden and implemented in inherited class for each coded runlet. The class must be inherited only once for each coded runlet and exactly one method should be overridden. Error should be thrown on application startup if both or none of the methods are overridden in inherited class. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw an exception. Two methods of the class define an entry point of a coded runlet execution on different levels: with an internal custom-coded run loop inside (method **run**), or with a system-provided external run loop (method **process**).

Specification

Methods

- **run()**
  - **Type:** Instance
  - **Parameters:** None
  - **Description:** This method gets called at the application startup if it is overridden. The method restarts automatically after returning if the application is still in a running state, except cases when the method threw an exception – the application gets terminated in such case. The method is responsible for fetching input signals by calling method **get_input_signal** of a class **Runlet** and it must implement a run loop inside, fetching input signals and processing them. Property **running** of the object of class **Application** must be used inside of a run loop to check if the application is still running, exiting the run loop and performing cleanup if the application is not running anymore.
- **process(input_signal: InputSignal)**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **input_signal** – a fetched input signal ready for processing.
  - **Description:** This method is called every time after an input signal is fetched for processing by an external code if it is overridden. The method should not fetch input signals directly by calling method **get_input_signal** of a class **Runlet** (because external system run loop does that) and any attempt to make such call inside of this method should result in an exception thrown. All attempts to close input signal

Class **EntryPoint**

transaction by calling method **finish_processing** for an object passed in a parameter **input_signal** inside of this method must be ignored, as the call will be made in external run loop after the method returns. Running the method for a long time would block processing of input signals for the runlet. Therefore, Pipe virtual machine may abort execution of this method and throw an exception if processing time of an input signal exceeds a certain time threshold.

Class **Application**

Description

This class represents current application. Instances of this class cannot be created directly and any attempt to create an instance of this class must throw a native non-Pipe exception. Static method **get_application** of this class must be used to get shared instance of the class.

Specification

Properties

- **running: bool**
  - **Access:** Read-only
  - **Description:** The flag indicating if the application is currently running.

Methods

- **get_application()**
  - **Type:** Static
  - **Parameters:** None
  - **Description:** This method returns the shared instance of the class **Application**.
- **terminate()**
  - **Type:** Instance
  - **Parameters:** None
  - **Description:** This method terminates currently running application.
- **throw_exception(code: int, description: str, data: object)**
  - **Type:** Instance
  - **Parameters:**
    - ✓ **code** – integer code of the exception.
    - ✓ **description** – details about the exception.
    - ✓ **data** – any additional data.
  - **Description:** This method throws Pipe exception that can be intercepted by traplets. The method throws a native non-Pipe exception indicating incorrect method usage if the application is not in running state. The method also throws native non-Pipe error exception indicating incorrect method usage if the runlet is not processing an input signal, i.e. a transaction is not open (a new transaction gets opened after returning **InputSignal** object from the method **get_input_signal** of the class **Runlet** and it is closed by calling method **finish_processing** of the class **InputSignal**). Calling this method closes open transaction automatically, so the runlet can call method **get_input_signal** of the class **Runlet** later without the need to close previous transaction first if thrown exception is caught and processed by a traplet.

2.5.3. Primitive Data Type Mapping

The mapping between Pipe primitive data types and Python data types is provided in Table 15.

Table 15. Mapping between Pipe primitive data types and Python data types.

Pipe primitive type name	Python data type
string	str
bool	bool
int	int
float	float
datetime	datetime
binary	bytearray
any	object

3. Design Rationale and Future Development

3.1. Key Problems of Visual Language Design

There are several reasons why no widely adopted general-purpose visual programming language exists today (paragraphs in **bold** explain how Pipe solves corresponding issues).

One reason is that visual programming languages can be less expressive than traditional text-based programming languages. This means that it may be more difficult to write complex programs or to implement certain algorithms using a visual programming language.

Pipe addresses this issue in many ways, such as using only generic visual programming elements. Another solution mitigating this issue is avoiding explicit depiction of data field mappings, which requires considerable efforts when creating them in visual languages. Pipe employs implicit name matching method (see 2.1.4.1.11 "Overlap") to automatically connect source and destination data fields with the same path in corresponding tree structures.

Another reason is that visual programming languages can have a steep learning curve for those who are not familiar with programming concepts. This can make it more difficult for beginners to learn how to program using a visual programming language as they may need to learn the programming concepts, specific syntax, and structure of the language.

Pipe mitigates this issue by minimizing the number of language elements and by introducing a concept of level of usage (see 2.3 "Levels of Usage").

Additionally, visual programming languages may not be as well suited for certain types of tasks as traditional text-based languages. For example, some programming tasks may require the use of complex algorithms or data structures that are difficult to represent visually.

One of the key design principles of the visual language Pipe is that it does not replace but complements text-based languages, recognizing the fact there are tasks that are best left to text-based programming. As a result of such approach, any non-visual programming language can be integrated seamlessly with Pipe via a common API. The popular and easy-to-use scripting language Python was even made an integral part of Pipe.

3.2. Language Design

3.2.1. Introduction

Pipe was created with an objective to make visual programming a usable method of software production. The key design goal was to make this language as practical as

possible. Therefore, Pipe provides only simple atomic general-purpose elements to be able to cover the broadest spectrum of potential algorithms. Also, highly detailed Pipe language specification provides comprehensive coverage of inner workings for every element.

Pipe was created with an idea of openness, meaning that citizen developers are not constrained by prebuilt visual blocks. They have visual workflow builder side-by-side with integrated non-visual programming tools available for creating new visual components using a simple scripting language Python. Common API allows integration of the Pipe workflows with almost any non-visual object-oriented programming language, significantly expanding possibilities of Pipe components customization.

To make learning the language easier and accessible for a broader range of citizen developers, Pipe introduces several levels of usage where most of the usage levels require knowledge of just a part of the specification.

Data mapping in Pipe is implemented in such a way that field matching between source and destination is done automatically based on full node paths in tree data structures. It saves significant amount of time during diagram construction.

3.2.2. General Design Aspects

3.2.2.1. Terminology

Basic concepts for most of Pipe elements were borrowed from traditional non-visual languages. However, majority of the elements were renamed – that is why Pipe contains so many new terms, even though most of them have obvious counterparts in non-visual languages. The main reason for renaming was because those elements still have some differences in a visual environment compared to traditional non-visual context, so keeping old names would be confusing and misleading. Also, new names were designed to be short and easy for pronunciation – for example, that is why there are so many terms in Pipe ending with suffix **-let** (runlet, memlet, synclet, translet, etc.).

3.2.2.2. Diagrams

3.2.2.2.1. Colors

To avoid diagram readability issues due to coloring, Pipe has strict color requirements where only white is allowed for background (with a possibility of a grid with grey lines) and all elements are shown using only black, grey, and white colors strictly according to drawing specification for each element. On the other hand, developers may need colors to mark and highlight different parts of diagrams for internal purposes. To reconcile these two conflicting requirements, Pipe introduces **production** and **non-production** diagram modes. Production mode has strict color requirements, while non-production mode allows free coloring (see 2.2.1.1 “Colors”).

3.2.2.2.2. Orientation

There are two possible flow orientations available for a diagram: horizontal and vertical. Horizontal orientation looks more natural for diagrams in general, and that is how they are drawn in documents where such orientation also saves page space. On the other hand, vertical orientation is preferable due to the following reasons:

- Vertical-only scrolling is used in two most popular categories of software – text editors and web browsers, so it is a very intuitive gesture for computer users.
- Rectangular elements with text inside cause huge waste of space if they are drawn horizontally. The issue might be resolved by rotating the text inside of elements 90 degrees, but this is bad for diagram readability. Vertical orientation produces significantly smaller areas of wasted space (see Figure 46), resulting in much more compact diagrams.

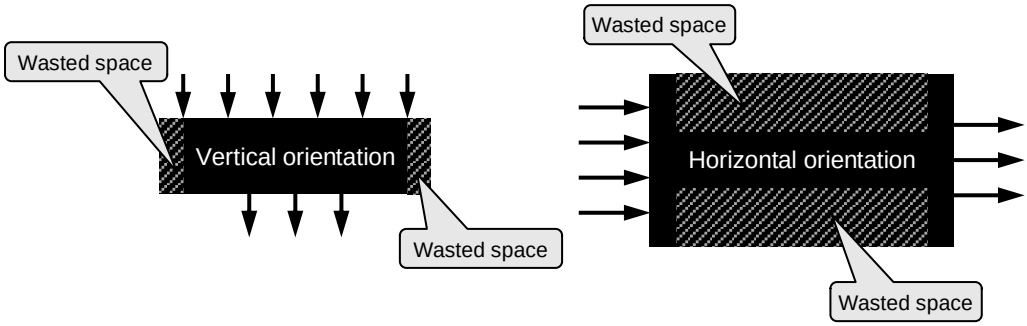


Figure 46. Vertical and horizontal orientation of components.

Therefore, Pipe uses vertical general flow orientation with inputs on top and outputs at the bottom of a component. However, it is not required for a layout of two adjacent components to be strictly one under another – for example, they can be on the same level horizontally (see 2.2.1.2 “Orientation”), as long as the overall diagram flow is from top to bottom.

3.2.3. Language Design Details

3.2.3.1. Components

Components are building blocks for composing **pipelines** – Pipe workflows implemented as directed graphs (see 2.1.1 “Pipeline”). Both pipelines and components have their own **inputs** and **outputs**. Component inputs and outputs are called collectively **pins**. The second level of the component taxonomy tree consists of two basic component types – **runlets** representing data processing units, and **memlets** implementing memory storage (see 2.1.2.1 “Component ”). All other component types are either subtypes or associated elements of these two basic component types.

There are nine elements in Pipe directly usable for workflow construction – they are shown as leaves of the component taxonomy tree (see Figure 2). This small number of elements still allows implementation of a wide range of algorithms, because each element implements a general-purpose concept. This is a design approach replicated from general-purpose non-visual languages, where relatively few concepts (interface, class, method, variable, etc.) still allows implementation of a wide variety of algorithms due to atomic and broadly general nature of those concepts.

3.2.3.2. Runlets

3.2.3.2.1. Introduction

Runlets are data processing units. They are similar to classes in non-visual object-oriented languages, where runlet inputs correspond to class methods (one method per input), and runlet outputs – to method return values, except there is no 1:1 relation between inputs (methods) and outputs (return values), because a single input signal in Pipe can produce signals on any combination of outputs. There are several runlet types defined in Pipe language – they are described below.

3.2.3.2.2. Prime and Scripted Runlets

Prime and **scripted runlets** are used for integrating non-visual programming languages with Pipe. Prime runlets can use any non-visual language except Python, and scripted runlets use Python. The reason why Python-based runlets were moved into a separate category from prime runlets was because Python is a first-class citizen in Pipe compared to other non-visual languages. Therefore, all Pipe implementations must have integrated Python support. This gives an opportunity for advanced citizen developers to use Python for making necessary runlets by themselves, without resorting to professional software development help. Python is a scripting language which is easy to learn and use. Special runlet API must be used for any non-visual language integration with Pipe. Runlet API is an integral part of the Pipe language specification (see 2.5 “Runlet API”).

3.2.3.2.3. Composite Runlets

Composite runlet contains a pipeline inside. Therefore, composite runlets are containers for other Pipe components (see 2.1.2.2.1 “Runlet Types”). Pipelines may exist only inside of composite runlets (see 2.1.2.2.4 “Runlet Tree”). **Ports** inside of a composite runlet connect internal pipeline elements with external pins of the runlet.

3.2.3.2.4. Inline Runlets

Inline runlets are Python-based components placed at specific location of a connection and processing all passing signals. There are the following types of inline runlets:

- **Mutator** – inline runlet with one input and one output. Mutators can modify input data contents but not input data structures.

- **Tester** – inline runlet with one input, and one or two outputs (one is a **positive output** and another – **negative output**). Testers are not allowed to change input data contents or input data structures. Tester must direct input signal to either positive or negative output.

Inline runlets were created with the goal to simplify embedding Python code fragments into a Pipe workflow: dropping inline runlet on a connection is much faster and easier than wrapping Python code into a scripted runlet and then explicitly assigning domains to all its inputs and outputs.

Both inline runlet types have counterparts in non-visual languages: testers are equivalent to an if-statements in non-visual languages, and mutators – to expressions.

3.2.3.2.5. Fixed Runlets

Fixed runlets are build-in elements of the language. They have a predefined behavior. There are the following types of fixed runlets in Pipe (see 2.1.2.2.2 “Fixed Runlets”):

- **Mergers** – components for merging two or more data objects from multiple connections. They are discussed in 3.2.3.12 “Mergers”.
- **Transformers** – components for transforming structure of a single data object. They are discussed in 3.2.3.13 “Transformers”.
- **Beacons** – components for generating startup/shutdown signals. They are discussed below.

Beacons are components for generating signals during **application** (see 3.2.3.2.7 “Runlet Tree”) startup and shutdown. Beacon has no inputs and one output. The order of beacon firing inside of a pipeline and between pipelines is not defined. However, there are **priority beacons** that are fired in specific order inside of a pipeline according to assigned priority number (lower number means higher priority). Startup beacon is shown as black upside-down rectangle and shutdown beacon – as black upside rectangle (see 2.2.2.1.1.3 “Beacon”). Startup beacons are equivalent to class constructors in non-visual programming languages, and shutdown beacon – to destructors:

```
class MyClass {
    MyClass() { }   // Constructor (non-visual equivalent of startup beacon).
    ~MyClass() { }  // Destructor (non-visual equivalent of shutdown beacon).
};
```

3.2.3.2.6. Reusability

When new runlet is created, it can be reused multiple times on the same or different pipelines by drag-and-dropping the element into a workflow. The originally created runlet is called a **template runlet** – it does not participate in pipelines directly, but its copy (called **instance runlet**) gets created when inserting it into a pipeline. All types of runlets can be reused except inline and fixed runlets that do not have corresponding template runlets, existing as a single instance at locations where they are placed. Runlet types that can be reused in multiple locations are called **reusable runlets** (see 2.1.2.2.3

"Template/Instance Runlets"). Only prime, scripted and composite runlets are reusable in Pipe, and only those runlet types may have domains (see 2.1.4.1 "Domains") explicitly assigned to its inputs and outputs (see 2.1.4.1.21.1 "Domain Assignment Scope" and 2.1.4.1.21.2 "Domain Assignment Methods"). Explicit domain assignment to runlet pins requires extra efforts when adding a runlet to a pipeline, but it makes the runlet independent of its context. Therefore, the runlet can be reused in different places, because explicitly assigned domains will ensure that data will enter and leave the runlet using the same data structures of defined domains no matter where the component is placed. All non-reusable runlet types (inline and fixed runlets) automatically inherit domains from connections attached to runlet inputs/outputs and therefore, structure of incoming and departing data depends on a context where the runlet is placed. Such runlets cannot be reused, and the only way to reuse such components is by wrapping them up into reusable runlets. On the other hand, absence of explicit domain definitions for all inputs and outputs makes it much easier and faster to insert such runlets into diagram, and that was the primary reason why domain assignment for inline and fixed runlets were made implicit in Pipe despite the fact it makes them non-reusable components.

3.2.3.2.7. Runlet Tree

As a composite runlet contains other components inside, it comprises a tree structure. A set of pipelines nested within one another, starting from the lowest tree leaves and extending up to the top-most (root) level defines a tree called a **runlet tree** (see 2.1.2.2.4 "Runlet Tree"). A root of the runlet tree is called a **root runlet** – composite runlet that holds a top-level pipeline inside, unless runlet tree represents a single coded runlet – the root runlet contains programming code in such case. Runlet tree available for modifications is called a **project**. A set of projects packaged together is called a **solution**. The following project types are defined in Pipe:

- **Library** – root runlet of such project type can be reused in the same solution, or **exported** to be **imported** into another solutions for reuse.
- **Application** – one project of such type can be assigned as an entry point for execution in a standalone application. Root runlet of such projects can also be reused in the same and other solutions like library project.

The same solution can play the role of either an application to be loaded into Pipe virtual machine for execution, or library to be imported to another solution for reuse.

3.2.3.2.8. External Connectivity

A root runlet of an application cannot use regular ports, but it must connect to Pipe virtual machine via **system input ports** and **system output ports** (see 2.1.2.2.5 "External Connectivity"). **Application type** determines what specific system input and output ports are available for connectivity, and data structure of each port. For example, Web application type may provide HTTP request as a system input port and HTTP response as a system output port. Each Pipe implementation is free to determine its own application types and corresponding system input/output ports.

3.2.3.3. Memlets

3.2.3.3.1. Memlets and Membanks

Memlets are components for storing data. However, the data is not stored directly in a memlet – it is stored in a connected membank which does not have its own inputs and outputs. Therefore, all membank data operations are performed via connected memlets (see 2.1.2.3.1 “Memlets and Membanks”). Memlets sharing the same membank are called **sibling memlets**. Data modifications in a memlet must be instantly reflected not only in connected membank, but also in all its sibling memlets.

Membanks are analogous to variable declarations in non-visual languages. Variable declarations do not participate in expressions and therefore, they cannot be a part of any data operation. Like membanks, variable declarations only allocate a named memory area for storing data. Memlets are equivalent to reference variables which can be part of expressions performing data operation on behalf of referred variable declarations, similar to how a memlet connected to (referred to) a membank participates in data operations on behalf of its membank:

```
int a = 0;  // Like membank, variable declarations can’t be a part of an
            // expression, they only allocate named memory area where data
            // will be stored.
a = b + 1;  // Expression of data writing into the reference variable “a”
            // (similar to writing to memlet).
c = a – 2;  // Expression of data reading from the reference variable “a”
            // (similar to reading from memlet).
```

3.2.3.3.2. Memlet Bond Attributes

Memlet has one input, but different connections may attach to that input via different **bonds** (see 2.1.2.3.2 “Memlet Bond Attributes”). This feature is called **bond separation**.

Memlet response to input signal depends on attributes of a bond the signal is passing through on its way from connection to memlet input. The following bond attributes indicating **memlet bond type** are defined for memlet input bonds (exactly one of these attributes must be specified for each memlet input bond):

- **Direct attribute** – memlet content gets overwritten with a value from arrived signal and output are generated with a new content. It is equivalent to the sequence “first write then read” in a non-visual language:

```
int a = 0;          // Declaring variable.
int ret = (a = 2);  // Writing new value to the reference
// variable and then reading new value in the same expression.
```

- **Push attribute** – value from arrived signal overwrites memlet content and output is generated with a previous content. It is equivalent to the sequence “first read then write” semantics in a non-visual language:

```
int a = 0;   // Declaring variable.
int ret = a; // First reading old value from the reference
             // variable ...
a = 2; // ... then writing new value into the reference variable.
```

- **Write attribute** – value from arrived signal overwrites memlets content and no output is produced. It is equivalent to "write-only" semantics in a non-visual language:

```
int a = 0;   // Declaring variable.
a = 2;       // Only writing into the reference variable.
```

- **Read attribute** – value from arrived signal is ignored, and output is generated with the current content. It is equivalent to "read-only" semantics in a non-visual language:

```
int a = 0;   // Declaring variable.
int ret = a; // Only reading from the reference variable.
```

An example provided in the Figure 47 swaps values between two non-sibling memlets, showing how different memlet bonds work together. Tracing of the diagram is provided in Table 16. Please note that the read bond type for ***Memlet_A*** used in **Step 1** can be replaced with a push bond type and the swap algorithm would still produce the same result, but with different tracing (verification of this fact is left to a reader).

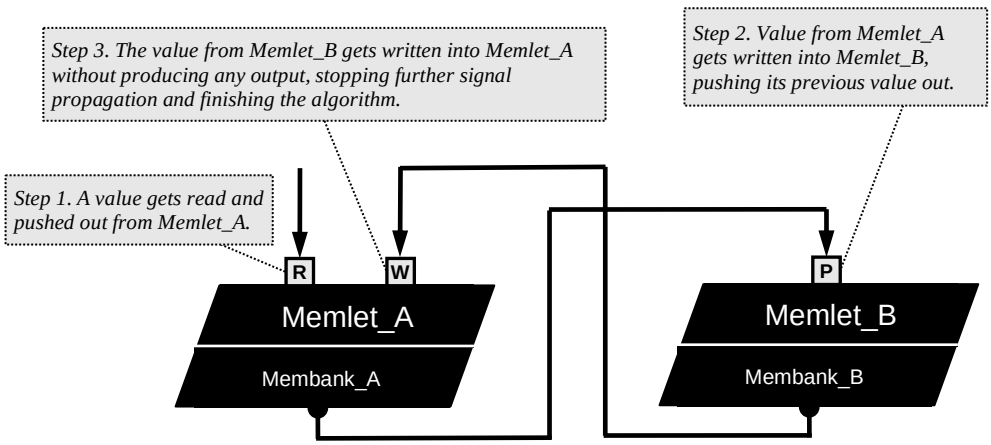


Figure 47. Value swap between two non-sibling memlets.

Table 16. Tracing of the value swap diagram.

Step	Memlet values at the start of the step	Memlet value at the end of the step	Step description
1	Memlet_A = V_1 Memlet_B = V_2	Same (no change)	Input signal enters ***Memlet_A*** via a read bond type. As a result, ***Memlet_A*** does not change and its value $\mathbf{V_1}$ is pushed out to the output.
2	Memlet_A = V_1 Memlet_B = V_2	Memlet_A = V_1 **Memlet_B = V_1**	Value $\mathbf{V_1}$ pushed out from ***Memlet_A*** in previous step, enters ***Memlet_B*** via a push bond type. As a result, new value $\mathbf{V_1}$ gets written into ***Memlet_B*** and its old value $\mathbf{V_2}$ is pushed out to the output.

Step	Memlet values at the start of the step	Memlet value at the end of the step	Step description
3	Memlet_A = V_1 Memlet_B = V_1	**Memlet_A = V_2** Memlet_B = V_1	Value **V_2** pushed out from ***Memlet_B*** in previous step, enters ***Memlet_A*** via a write bond type. As a result, value **V_2** gets written into ***Memlet_A*** and nothing is produced at the output, stopping further signal propagation, and finishing the algorithm.

3.2.3.3.3. Broadcast Bond Attribute

A signal entering memlet generates output only from the memlet that accepted the signal, and not from any of its sibling memlets. To generate output for all sibling memlets, a memlet (called **triggering memlet**) must accept the signal via a bond with a **broadcast attribute** (see 2.1.2.3.3 "Broadcast Bond Attribute"). The type of response from all sibling memlets depends on input bond type of a triggering memlet. For example, the previous content of each sibling memlet is produced in case of a signal entering the triggering memlet via a push bond type.

3.2.3.4. Connectivity

3.2.3.4.1. Signal

Signal is produced by a component output or by an internal event inside of a component, passing data between components and triggering component execution after entering input of a component, or after starting to handle an internal event produced inside of a component (see 2.1.3.1 "Signal"). A signal may optionally carry an attached **data object** containing data passed between components, but a signal triggers component execution no matter if it has an attached data object or not. A signal with attached data object is called **carrier signal**, and signal without an attached data object – **blank signal**.

There are two types of signals: **external** and **internal**. External signals enter a component via its inputs. Internal signals are produced inside of a component.

Signal is a powerful concept in Pipe because it merges dataflow and control flow together into a unified concept of a generalized dataflow. Let's look at the example in the Figure 48, with a source code fragment (representing both dataflow and control flow) on the left and corresponding dataflow graph on the right. Please notice that dataflow in the code can be represented by a single connected graph.

```
a = b + c;
f = a;
```

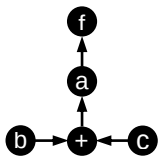


Figure 48. Dataflow of the code fragment.

Let's swap two source code lines and re-build dataflow graph again (see Figure 49). Now dataflow graph falls apart into two disconnected fragments because dataflow always

assumes passage of data, but there is no dataflow dependency between the first and the second code lines anymore. There is still a dependency between two lines of code, but this is a control flow dependency now, not dataflow dependency. That is one the problems of working with non-visual programming languages – they operate in two separate dataflow and control flow dimensions, and that makes much harder to read and understand text-based source code.

```
f = a;
a = b + c;
```

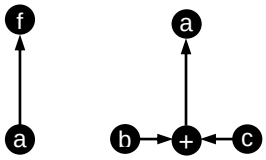


Figure 49. Dataflow after swapping two lines of code.

Let's generalize dataflow concept and assume that dataflow transitions without data are allowed, i.e., control flow between two executable elements is represented by a dataflow transition without passing data in the generalized dataflow model (see Figure 50).

```
f = a;
a = b + c;
```

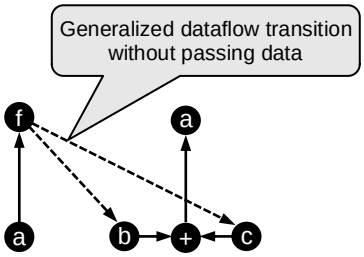


Figure 50. Generalized dataflow model.

Rebuilding dataflow graph using generalized dataflow model produces a single connected graph fully reflecting the source code logic. As a result, the single graph completely describes text-based code. Generalized dataflow model allows to show logic on a single graph instead of a split two-dimensional dataflow and control flow models used by non-visual languages. Pipe uses this concept by combining dataflow and control flows together and allowing a signal without an attached data object to trigger component execution. Specifically, carrier signals in Pipe are equivalent to dataflow in non-visual languages, and blank signals – to control flow.

3.2.3.4.2. Connection

Connection is a directional relation for a signal propagation from ***M*** inputs to ***N*** outputs, where ***M*** > 0 and ***N*** > 0 (see 2.1.3.2 "Connection"). Starting connection points are called **source points** and ending – **destination points**. Source and destination points are called collectively **endpoints**.

Please note that connection in Pipe is defined in its most complex and general form: it is allowed to have arbitrary non-zero number of both source and destination endpoints.

3.2.3.4.3. Signal Distribution and Processing Rules

Signal distribution and processing rules (see 2.1.3.3 “Signal Distribution and Processing”) represent the core part of the Pipe language specification, outlining key signal propagation rules in a consistent and clear manner. Each rule is discussed below. Exact wording of each rule is provided in **bold**.

Rule SDP-1. If a signal is produced from a source point of a connection with multiple destinations, then it is delivered to all destination inputs (see Figure 9). Signals entering inputs of different components are processed in parallel, independently from each other. Signals entering input(s) of the same component are processed sequentially one-by-one according to the rule SDP-4.

Rule SDP-1 defines conditions for a signal to branch out into multiple execution streams running in parallel. The rule states that parallel processing is achieved only for signals entering inputs of different components, while signals entering inputs of the same component are processed by a single execution stream sequentially.

Signal branching is equivalent to UNIX function **fork()** and other similar methods that create a new parallel execution stream (process, thread, etc.) in non-visual languages.

Rule SDP-2. If several signals are produced from multiple source points at the same time, they are enqueued at the connection and fetched from the queue one-by-one for delivery to destinations: the next signal is fetched from the queue only after the current signal is delivered to all destinations for processing according to the rule SDP-1 (see Figure 10).

Rule SDP-2 expands rule SDP-1 to the case of a connection with multiple sources and multiple signals arriving from those sources. The most important aspect of this rule is that the signals do not merge at the connection merging point, getting sent to destinations one-by-one as separate signals.

Rules SDP-1 and SDP-2 seem obvious and excessive. However, the reason why Pipe language specification is so detailed is because the goal was to remove any ambiguity on all aspects of the language, leaving no stone unturned in every case where alternative interpretation has a reasonable probability to occur. For example, even if the rule SDP-1 seems obvious and most users would correctly assume mechanics of how a single signal is delivered to multiple destination, there is still a non-zero probability of alternative interpretations of that case. For example, some users may assume that the signal is delivered only to destinations not busy processing other signals. Such kind of implicit assumptions would create ambiguity, so these cases must be mentioned explicitly. Use-case of a connection configuration branching out and then merging back (see 2.1.3.4 “Signal Multiplication”) proves that point. At first sight, such connection configuration changes nothing in signal propagation as it is equivalent to a single-line connection and therefore, the stream of signals must pass such connections unchanged. However, applying rules SDP-1 and SDP-6 produces a different result: branching and merging connection back multiplies a signal, producing many output copies of a single input signal.

Enqueuing multiple signals at the connection in Pipe are analogous to explicit queues used in non-visual languages to ensure sequential processing of incoming requests.

Rule SDP-3. Signal processing inside of a component is triggered by an external signal entering the component input, or by an internal signal produced inside of the component. Processing of a signal may produce side effects (such as printing output, changing database, etc.). Zero or more output signals can be generated from any outputs as a result of signal processing.

Rule SDP-3 states that component execution is triggered by an external or internal signal. This rule allows side effects of signal processing. It also states that output production is optional, and arbitrary number of output signals can be produced from any combination of outputs as a result of processing of a single input signal.

Rule SDP-4. All signals are processed inside of a component one-by-one in the order of arrival. A component may process no more than one input signal at any given moment. Processing of the next input signal starts only after the current signal is fully processed and all output signals are produced. Output signals must become available on external outputs immediately after their production without waiting for completion of input signal processing. Signals waiting at inputs of a merger inside of a composite runlet are considered fully processed, thus not blocking fetching the next input signal for processing in that composite runlet.

An exception from the rule SDP-4 is a merger (see 2.1.4.2.1 "Mergers"), which waits for arrival of signals to all its inputs and then starts processing all signals together.

Rule SDP-4 is very important. It states that signals arrived at different inputs of a multi-input component are processed sequentially one-by-one. Sequential order of signal processing is extremely important to avoid situation when multiple signals enter different inputs simultaneously, producing unpredictable result due to multiple race conditions, thus making their tracing and debugging impossible. This rule establishes sequential order of cause-effect chain for signal processing: the next input signal (next cause) is not starting unless the current cause (currently processing signal) plays out all its effects (i.e., until it generates all output signals). Sequential nature of signal processing does not guarantee problems with signal processing (such as race condition – see example in the Figure 51), but such cases will very likely be detected by a static diagram analysis, without a need in a dynamic tracing.

An important note about merger is that signals waiting at merger inputs are considered fully processed from the point of view of a parent composite runlet, thus not blocking the parent runlet from fetching the next signal for processing. This rule was introduced because otherwise a signal waiting at the merger input would create a deadlock situation for the parent runlet, which would not be able to start processing the next signal until the current signal is fully processed, while the currently processing signal must wait at merger input for the next signal to trigger merger execution and

removal of the waiting signal from the merger input. This creates a circular dependency between signals waiting at merger inputs inside the composite runlet and signals waiting at inputs outside of the runlet.

Please also notice this sentence in the rule: *"Output signals must become available on external outputs immediately after their production without waiting for completion of input signal processing"*. It means signals can appear on outputs in the middle of processing. This part of rule is crucial for cases when a component is processing one input signal in a loop, continuously producing output signals.

An exception from rule SDP-4 is a merger that processes input signals differently, waiting for all signals to arrive to all its inputs before starting their processing together.

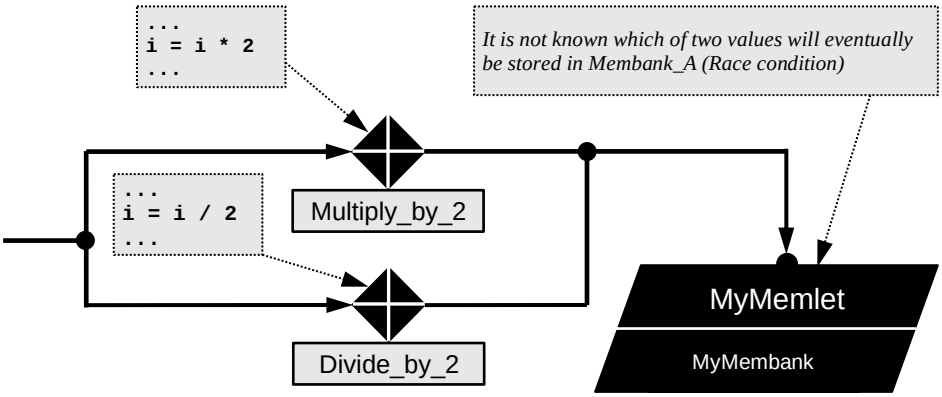


Figure 51. Race condition.

Rule SDP-5. All internal signals produced inside of a component are waiting for processing in the same common queue (called input queue) together with the external input signals arrived at all inputs of the component. However, propagation of internal signal during its processing inside of a component starts from an internal component's output which produced the internal signal before it was enqueued at a parent component, while propagation of external input signal starts from a port connected to a corresponding input the signal has arrived at. An exception from this rule is a merger that has separate input queues per each input.

Rule SDP-5 says that internal signals produced inside of a component are getting enqueued together with external input signals of the component in a common input queue for all input signals. The rule also states that internal signal processing starts not from a port connected to an input of the component, but from the output of an internal component that produced the signal. Internal signals in Pipe are analogous to asynchronous callbacks in non-visual languages.

Rule SDP-6. Several signals cannot be processed together until their data objects are explicitly merged. No implicit signal merge is allowed. Merger (see 2.1.4.2.1 "Mergers") should be used to merge two or more signals together by combining their data objects.

Rule **SDP-6** specifies how to merge multiple signals together, stating that there is only an explicit way to merge signals by using a merger.

The closest counterpart of the merger component in non-visual language is **waitAll()** call that waits till all running threads or tasks from a specified list are completed.

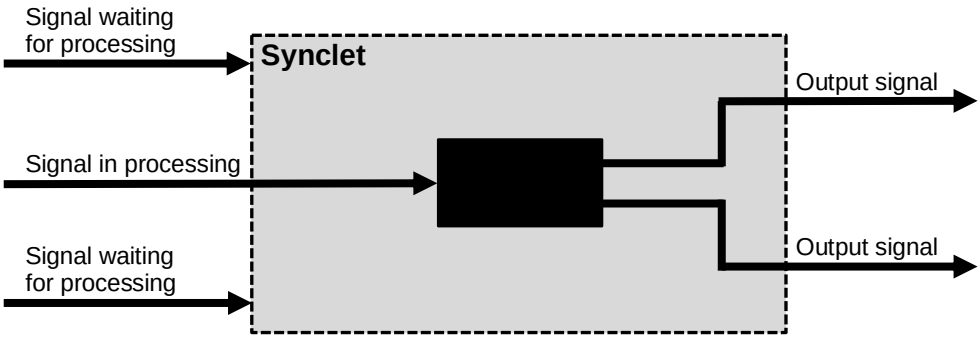


Figure 52. Signal processing by synclet.

3.2.3.4.4. Synclets

Synchronization is a capability to process incoming signals one-by-one in the order of arrival, when processing of the next input signal starts only after the current signal is fully processed and all outgoing signals are produced. **Synclet** is an arbitrary convex shape providing synchronization for all components located inside of its shape (see Figure 52). Synclets are equivalent to **lock()/unlock()** call pairs used in non-visual languages to ensure no more than one thread runs inside of a block at a time.

To enforce rule SDP-4, every component except merger has an **implicit synclet** attached to the component and covering its area edge-to-edge.

It is not necessary to wrap a part of workflow into a component just to leverage its implicit synclet capabilities to ensure synchronization. The synclet rectangular shape can be placed directly on a pipeline surface, providing synchronization to all components inside of the shape. It is a key reason why explicit synclet shape was introduced in Pipe.

3.2.3.5. Domains

Domain is a structure arranging data as a tree called a **domain tree** (see 2.1.4.1 “Domains”). Each node of a domain tree contains the following **domain node slices**: **value slice** and **children slice**. Each of these slices is optional and domain nodes are classified according to presence or absence of each slice at the node (see Table 3). For example, a node containing children slice is called **group node**, and node containing value slice – **storage node**.

Pipe domains use their own format of **domain specification** (see 2.1.4.1.2 “Domain Specification”) and **object specification** (see 2.1.4.1.3 “Object Specification”). The reason why none of existing data formats (such as XML, JSON, YAML, etc.) were used for domain/object description was because existing formats bring many features that are

not needed, while many required features are often either missing or hard to implement in existing formats, so it was decided to create a new “clean” data format with all necessary features designed properly instead of dealing with complexities of existing data formats.

3.2.3.6. Domains vs Structures

Data holding entities in Pipe are called domains, not structures. This is because of one big flaw of non-visual language structures that resulted in a more generic data storage design pattern chosen for Pipe.

Table 17. Pipe domain and equivalent C++ structure.

Pipe domain	C++ structure

`Person`
`@`
`  FirstName -> string`
`  MiddleName -> string`
`  LastName -> string`
`  IsCustomer -> bool`
	`struct Person {`
	`  string firstName;`
	`  string middleName;`
	`  string lastName;`
	`  bool isCustomer;`
	`};`

Structures in non-visual languages are implemented as collections of data-holding members called fields. Each field has name and data type. A field is equivalent to a storage domain node in Pipe, so single-level pipe domain can be easily implemented as a structure in non-visual language (see Table 17).

However, multi-level domains with group nodes do not have equivalence in non-visual languages, because structures cannot directly group fields into higher levels of hierarchy. The only way to implement multi-level structure is via referencing another structure as a data type (see Table 18). Such capabilities already exist in Pipe language too as a separate feature, where storage nodes may reference another domain as data type. However, there is no direct equivalent of Pipe group nodes in structures. The key difference between structures and domains is an ability for domains to group its members “inline”, without resorting to referencing another data type.

Table 18. Emulation of multi-level pipe domains in C++.

Pipe domain	C++ structure

`User`
`@`
`  Person`
`  FirstName -> string`
`  MiddleName -> string`
`  LastName -> string`
	`struct User {`
	`  Person person;`
	`};`
	
	`struct Person {`
	`  string firstName;`
	`  string middleName;`
	`  string lastName;`
	`};`

Pipe domain nodes can hold both value and child nodes – there is no equivalent for that feature in non-visual languages too. The main reason why every Pipe domain node must be allowed to have both values and children simultaneously is because otherwise it is impossible to implement domain merge without either loss of data or restrictions on

domain structures to be merged. However, foundational and multi-purpose nature of the merge operation in Pipe assumes there should not be restriction on structures of merged domains and all nodes must be combined in a resulted merged domain without exceptions.

3.2.3.7. Data Types

Value slices of storage domain nodes hold actual data and therefore, they must have **data type** (see 2.1.4.1.4 "Data Types") specified. The following **data type categories** are provided in Pipe:

- **Primitive data types** – simple data types such as integers, strings, etc. (see Table 4).
- **Domain data types** – domain (called **reference domain**) is used as data type.
- **Collection data types** – arrays holding objects of the specified data type (called **base type**).

Special **null value** is used to indicate storage node that does not contain a value. **Null object** is a special case of null value indicating null in place of a data object in memlet, or null in place of a data object attached to a carrier signal. Null value can be changed to non-null value via assignment. There is also a **none value** used for non-storage nodes to indicate they cannot store value. Value none cannot be changed to any other value without adding value slice to the node in its domain specification.

3.2.3.8. Node Attributes

Node attribute is a special token attached to a domain node and defining its behavior in certain cases (see 2.1.4.1.5 "Node Attributes"). A node can have more than one attribute assigned to it. By default, an attribute specifies behavior only for a single node it is assigned to. However, a special flag can be set for an attribute to indicate it is applied not only to a node it is directly assigned to, but also to all direct and indirect children (this feature is called **attribute grouping**). The following note attributes are defined in Pipe:

- **Default attribute** – a value assigned to the node for each newly created data object instead of a **default type value** specified for each data type.
- **Constant attribute** – same as default attribute, but the value of the node cannot be changed.
- **Nullable attribute** – the node is allowed to have null value. A node is not allowed to hold null value unless the nullable attribute is specified.
- **Optional attribute** – the node can be missing in data objects.
- **Mandatory attribute** – the node must participate in every overlap (see 3.2.3.9 "Overlap").
- **Forbidden attribute** – the node must not participate in any overlap (see 3.2.3.9 "Overlap").

- **Injection attribute** – the node will be included to any overlap (see 3.2.3.9 “Overlap”).
- **Rejection attribute** – the node will be excluded from any overlap (see 3.2.3.9 “Overlap”).

3.2.3.9. Overlap

Component inputs and outputs can have domains assigned to them (see 2.1.4.1.21 “Domain Assignment”). A signal transfers data from an output to input only for storage nodes that have the same domain path in output domain (called **source domain**) and input domain (called **destination domain**). Two domain nodes with the same domain path in source and destination domains are called **twin nodes**. A complete set of twin nodes for a specific pair of source and destination domains is called an **overlap** (see 2.1.4.1.11 “Overlap”) and it is also a domain. Overlap contains all nodes whose values will be transferred from a source to destination point in any data transfer operation.

There are certain conditions that must be met for the overlap to be valid (see 2.1.4.1.13 “Overlap Validation”). Overlap validity conditions are related to compatibility of data types and node attributes between twin nodes. Connection between the source and destination points is possible only if their overlap is valid. Transformer (see 2.1.4.2.2 “Transformers”) must be used to correct source domain if overlap is invalid. Overlap validation happens during workflow construction and it is equivalent to a compile-time data type checks in non-visual languages.

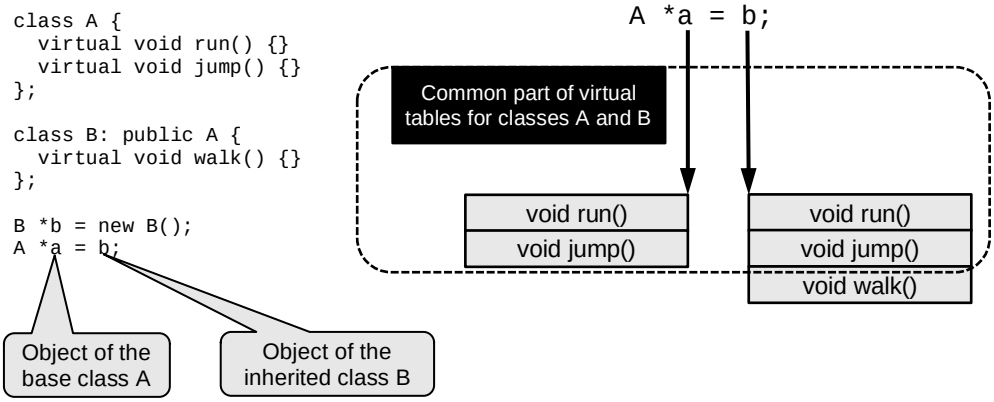


Figure 53. Virtual tables in C++.

Overlap concept in Pipe is a modification and generalization of polymorphism in non-visual object-oriented languages. Unlike polymorphism, overlap matches data fields instead of method signatures. The basic name-based member matching principle is the same in both Pipe and object-oriented languages, but overlap is much more flexible model because even an empty overlap might still be valid. On the other hand, there are strict polymorphism rules in object-oriented non-visual languages. For example, let’s look at the left-hand side (LHS) and right-hand side (RHS) objects of assignment expression in C++, where virtual table of RHS object must a continuation of virtual table

of LHS object (see Figure 53). To achieve such result, RHS object class must be inherited from the LHS object class. There are no such limitations in Pipe, where overlaps are valid as long as data types and node attributes of all twin nodes in overlap are compatible. Such flexibility allows connectivity between a wide variety of diverse components, even if they are produced by different manufacturers.

3.2.3.10. Domain Merge

Domain merge is a very important operation in Pipe used for mergers, multipoint connections, and several other cases (see 2.1.4.1.14 “Domain Merge”). Domain merge produces new domain based on both source domains, while overlap does not change source or destination domains, adapting data transfer to structures of both domains.

3.2.3.11. Domain Inheritance

Like C++ and other non-visual object-oriented languages, Pipe supports extension of existing domains via inheritance (see 2.1.4.1.22 “Domain Inheritance”). However, Pipe provides only data field (domain node) inheritance and not method inheritance, because Pipe domains do not have methods. Multiple inheritance is also supported.

Polymorphism in non-visual object-oriented languages is tightly linked to inheritance (see code example in the Figure 53). It means the polymorphism is not applicable for two different classes not related by inheritance, but with the same set of methods (see Figure 54). Pipe uses a different approach and domain inheritance does not affect how overlap works: two domains with no inheritance relation between them can still produce a valid overlap.

```
class A {
  virtual void run() {}  // Common method signature between A and B.
  virtual void jump() {} // Common method signature between A and B.
};

class B {
  virtual void run() {}  // Common method signature between A and B.
  virtual void jump() {} // Common method signature between A and B.
  virtual void walk() {}
};

B *b = new B();
A *a = b;       // Error: cannot convert *B to *A.
```

Figure 54. Unrelated classes in C++.

3.2.3.12. Mergers

Merger is a fixed runlet type designed for merging data objects from two or more signals (see 2.1.4.2.1 “Mergers”). Tree structures of source domains are merging at the output of a merger. Domains of different merging data objects might have nodes with the same domain path (twin nodes), creating **domain collision** situation during the merge: data objects from twin nodes of two or more source domains compete to get their values into one common destination node of a merged domain. There are several methods of domain

collision resolution for **collided nodes**. The following merger types are defined depending on method of collision resolution (see 2.1.4.2.1.2 “Merger Types”):

- **Naming merger** – collided nodes are converted to group nodes and special **merger resolution names** are assigned to each merger input to be used as names of storage nodes created under the group nodes (see Figure 12).
- **Priority merger** – numeric priorities are assigned to each merger input and version of a collided node with a higher priority (lower priority number) is used for resolution.
- **Exclusion merger** – twin storage nodes are excluded from a merged domain.
- **Strict merger** – no overlaps are allowed between domains assigned to merger inputs.

Merger waits for at least one signal to appear on each input before starting merging data objects of input signals together (see 2.1.4.2.1.3 “Merger Processing Algorithm”). Full set of signals at the front of each input queue is called **merge vector**. The process of waiting for signal accumulation on all inputs is called **vector formation**. Merger removes all vector signals from inputs (this process is called **vector collapse**) after vector formation is completed, and then it merges data objects of removed signals together. The following bond attributes are defined for mergers:

- **Immunity attribute** – it is not required for a signal to be present on an input with immunity bond attribute for a vector collapse to start.
- **Purge attribute** – signal passing via bond with purge attribute triggers vector collapse even in absence of a full merge vector.

Let’s consider an example of converting the following C++ function into equivalent Pipe diagram:

```
void doFirst() { /* Do something first */ }
void doSecond() { /* Do something second */ }

void doIt() {   // Simple two-step function without parameters.
  doFirst();
  doSecond();
}
```

Figure 55. Two-step C++ function without parameters.

The C++ code above can be easily converted to the following Pipe diagram:

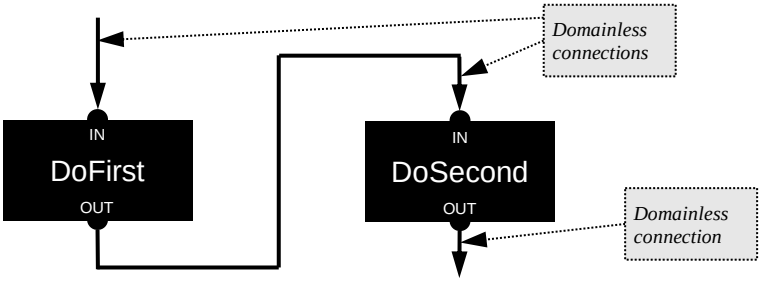


Figure 56. Pipe diagram equivalent to a two-step C++ function without parameters.

As C++ functions ***doFirst()*** and ***doSecond()*** are both void functions without parameters, inputs and outputs of equivalent runlets ***DoFirst*** and ***DoSecond*** on Pipe diagrams must be domainless (see 2.1.4.1.16 “Domainless Component Pins”). Such composition does not create problems and Pipe diagram fully reflects sequential execution of both steps in the right order, precisely mimicking behavior of the C++ code. Now let’s introduce one parameter for both functions:

```
void doFirst(int v) { /* Do something first */ }
void doSecond(int v) { /* Do something second */ }

void doIt(int p) { // Simple two-step function with a single parameter.
  doFirst(p);
  doSecond(p);
}
```

Figure 57. Two-step C++ function with one parameter.

C++ functions ***doFirst()*** and ***doSecond()*** are still void functions, meaning that runlet ***DoFirst*** cannot pass data to runlet ***DoSecond*** on Pipe diagram and output of runlet ***DoFirst*** must be domainless. However, passing input data object to inputs of both runlets ***DoFirst*** and ***DoSecond*** at the same time would mean they are going to be executed in parallel, while the requirement is a sequential execution of the runlets following the same order ***DoFirst*** -> ***DoSecond*** as in the case above (see Figure 57 and Figure 56). The solution of this problem is placing a merger between runlets ***DoFirst*** to ***DoSecond*** to ensure both dataflow part (arrival of an input data object corresponding to an argument of function ***doIt()*** in C++ code) and control flow part (completion of execution for runlet ***DoFirst***) are merged before starting execution of the runlet ***DoSecond*** (see Figure 58).

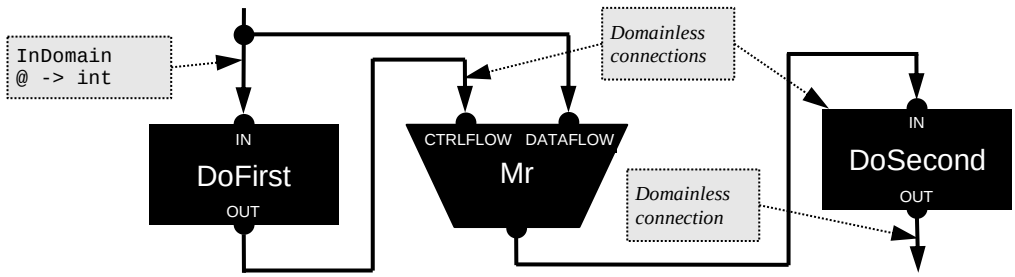


Figure 58. Pipe diagram equivalent to a two-step C++ function with one parameter.

Merger ***Mr*** on the diagram above plays the role of a gate, blocking propagation of the input data object with a structure of a scalar domain ***InDomain*** to the runlet ***DoSecond*** until execution of runlet ***DoFirst*** is completed. Runlet ***DoFirst*** has domainless output and therefore, the output of the merger ***Mr*** has the same domain as the one passed to the input of runlet ***DoFirst***: domainless merger input ***CTRLFLOW*** does not contribute any new nodes to the resulting merged domain produced at the output of the merger ***Mr***, keeping the output domain the same as the domain of a merger input ***DATAFLOW***. As a result, both runlets are running sequentially in proper order with an input data object passed to both in their original structure, making the diagram equivalent to C++ code.

There is another simpler solution avoiding merger altogether by assigning domain ***InDomain*** to output of runlet ***DoFirst***, making that output a passthrough for data objects of signals arriving to ***DoFirst***. This allows us to have same diagram as in the Figure 56, except runlet ***DoFirst*** is not domainless anymore, as it is passing data object to ***DoSecond***. This solution is simple, but it is not fully equivalent to source code in the Figure 57 as runlet ***DoFirst*** is supposed to have domainless output equivalent to ***void*** return data type of C++ method.

3.2.3.13. Transformers

Transformer is a fixed runlet type designed for changing domain structure of incoming data objects (see 2.1.4.2.2 “Transformers”). Transformer contains a chain of transformation steps – **translets**. Each translet applies a certain transformation to an input domain received from a previous translet in a chain, or from a transformer input for the first translet in the chain. The list of available transformation operations is provided in Table 12.

3.2.3.14. Exceptions

There is a mechanism in Pipe to catch and handle exceptions before they escape and terminate an application. This mechanism is equivalent to try/catch construct used for intercepting and processing exceptions in non-visual languages. Exceptions are implemented in Pipe by placing a rectangular area called **traplet** on a pipeline surface. The traplet will intercept exceptions coming from any level of depth for any runlet located inside of the traplet rectangle. The traplet may intercept all exceptions, or only selected exceptions with certain **exception codes**.

3.2.3.15. Component State

The default behavior for the following Pipe component types is to be **stateful**, keeping the state between processing of two successive signals (for definition of the state for each component type see 2.1.4.4.1 “Stateful Components”):

- Coded runlets (except inline runlets).
- Composite runlets.
- Membanks.
- Memlets.
- Synclets.
- Traplets.

Component state can be reset explicitly by sending a signal to a **reset input**, provided by default for every component type supporting explicit reset, or implicitly by overriding default behavior of a component (see 2.1.4.4.4 “Component State Overrides”).

3.2.3.16. Pipe Examples

3.2.3.16.1. Hello, World!

The simplest example of a Pipe diagram printing “Hello, World!” to the output is provided in the Figure 59. Signal propagation on the diagram starts from input port ***IN***. The signal from the port enters input of a mutator ***PrintHelloWorld*** that prints incoming string using Python. The rule SDP-3 allows for runlets to produce side effects (such as output printing) and it also allows not to produce output signals, so printing output without producing output signal is a legitimate runlet behavior in Pipe. Inline runlets are non-reusable Pipe components (see 2.1.2.2.3 “Template/Instance Runlets”), so the runlet ***PrintHelloWorld*** cannot be reused in other places. However, inline runlet may be wrapped into a reusable runlet type, and then the function printing “Hello, World!” can be reused anywhere in the solution, or in other solutions after exporting the runlet as part of library (see 2.1.2.2.4 “Runlet Tree").

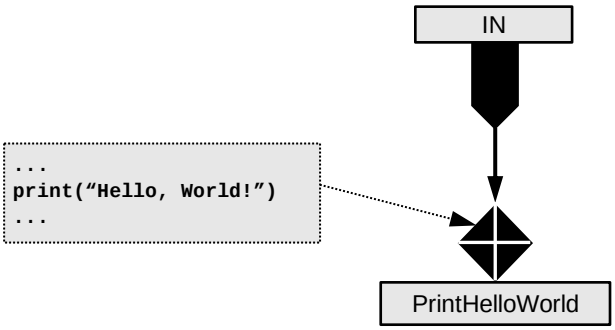


Figure 59. Diagram for printing “Hello, world!”.

3.2.3.16.2. Iterations

Iteration is a fundamental programming language concept. However, there is no dedicated loop component in Pipe because it can be easily built from other components. Reusable loop component should probably be a mandatory part of a Pipe standard library (see 3.2.3.18 “Pipe Standard Library”). Implementation of a loop must generate a sequence of integer numbers in range from specified starting and ending numbers, and with a specified step. A diagram of a loop implementation is provided in the Figure 60.

The algorithm starts after receiving signal from a port ***INIT***. The signal has an attached data object of domain ***LoopParams*** containing all loop parameters. Arrived data object gets initialized in mutator ***LoopInit*** by assigning start value to the current index, and then initialized data object is stored in memlet ***LoopParams*** via direct bond, meaning a new value is immediately pushed out from the memlet, where it is checked on end of the loop condition (current value > end value) in tester ***CheckEndOfLoop***. The current index is incremented by step value in mutator ***IterateToNextStep*** if end of the loop check in tester ***CheckEndOfLoop*** fails. An output of the mutator ***IterateToNextStep*** arrives to

the input of memlet ***LoopParams*** via direct bond, storing new value and pushing out that value from the memlet, repeating the loop in next iteration. Output of the mutator ***IterateToNextStep*** also goes into output port ***CUR***, intersecting with domain ***LoopCurrent*** and producing an index of the current loop iteration at the output. As a result, the loop produces a sequence of integers at the output ***CUR*** in a range from starting to ending numbers, and with a specified step. Memlet ***LoopParams*** seems to be redundant (we can pass incremented values directly without storing them), but the role of ***LoopParams*** would become clear after we modify the algorithm below.

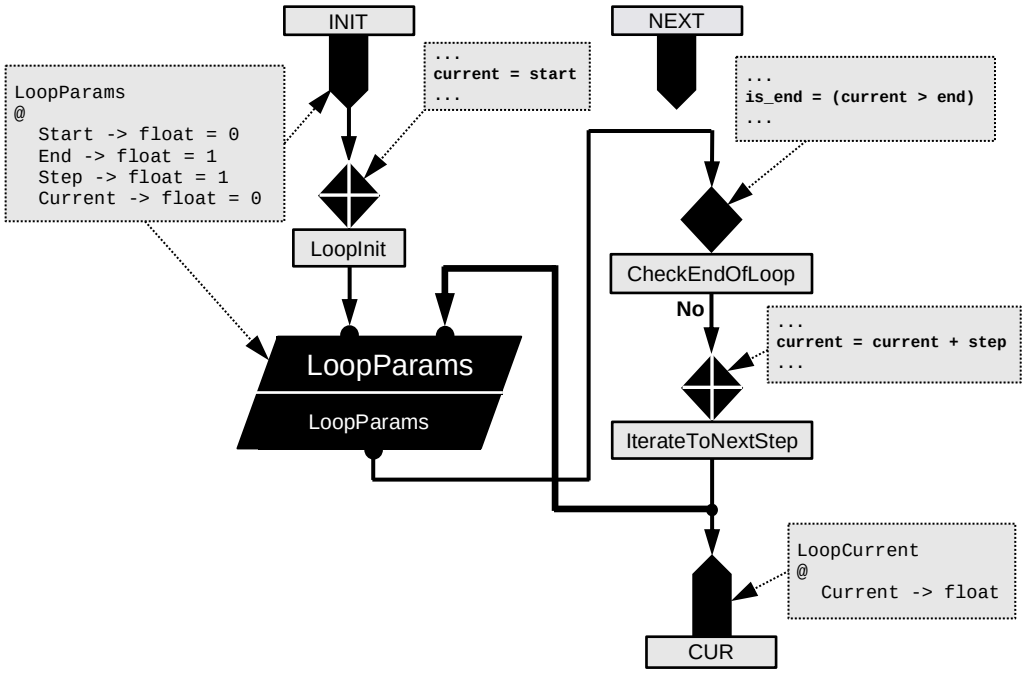


Figure 60. Loop component diagram.

Non-indexed direct iteration over a collection can be implemented in Pipe by using a single mutator with Python code inside iterating a collection and producing all collection elements at the output one-by-one – an example of Python code fragment is provided in the Figure 61. The code uses Runlet API (see 2.5 “Runlet API”).

```
for elem in coll:                                   // Iterating collection.
    data_obj = domain.create_data_object()          // Creating new data object.
    node = data_obj.get_node(“@/Item”)              // Searching for node to add value.
    node.set_value(elem)                            // Adding element to the node.
    input_signal.send_output(output_pin, data_obj)  // Sending data object to output.
```

Figure 61. Python code directly iterating over elements using Runlet API.

Direct and index-based methods of collection iteration provided above have the same problem: they generate full sequence at once, potentially consuming large amount of memory to hold all generated elements in queue until they are processed. Non-visual

programming languages have a concept of iterator object that holds current position of the collection, so that elements may be iterated in a collection one-by-one instead of extracting all elements at once and consuming memory space to hold them all.

Similar concept can be implemented in Pipe by using one more extra input ***NEXT*** in a loop runlet which will be generating output signals one-by-one in a diagram above (see Figure 60, all needed modifications are described at the end of the paragraph). The next output signal will be produced only after a signal enters input ***NEXT***. External iteration workflow must send signal to the input ***NEXT*** to get output signal for the next iteration after processing the current one. However, the loop runlet with external iteration input must be enclosed in a synclet if the loop is part of a larger workflow of the same pipeline, to avoid situations when the next incoming signal re-initializes the loop while the current iteration is still in progress (see Figure 62). The following changes in diagram in the Figure 60 will turn the loop into an optimized iterator-like version with one-by-one output production (tracing of the changed diagram is left to the reader as an exercise):

- Changing input memlet bond type from direct to write for a destination point of a bold connection line connecting output of runlet ***IterateToNextStep*** and input of the memlet ***LoopParams***.
- Connecting domainless output of a port ***NEXT*** to an input of the memlet ***LoopParams*** via a read bond type.

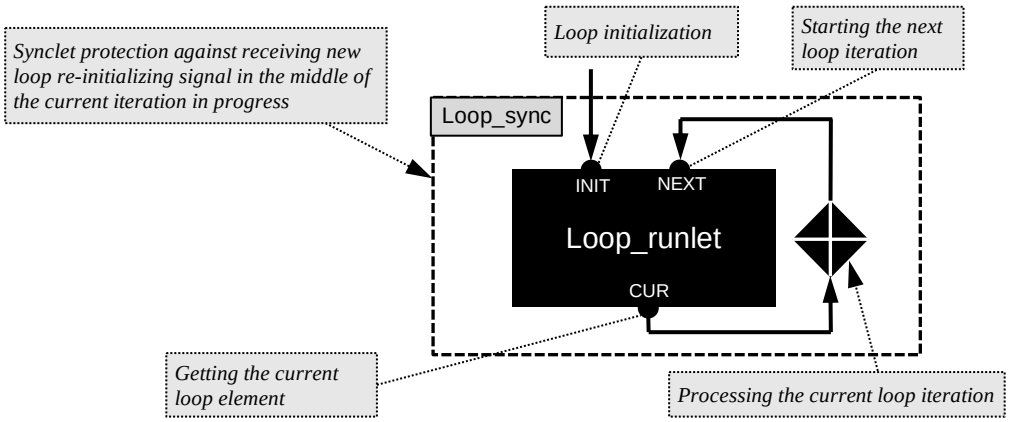


Figure 62. Optimized loop.

It might be noticed that a new optimized version of a loop diagram has a problem of race condition in case output ***CUR*** is connected directly to input ***NEXT***: the input may arrive earlier than the data object with current loop parameters gets stored in memlet ***LoopParams***. This is because of this part of rule SDP-4: *"Output signals must become available on external outputs immediately after their production without waiting for completion of input signal processing"*. However, it does not happen because of another part of rule SDP-4: *"Processing of the next input signal starts only after the current signal is fully processed and all output signals are produced"*. Therefore, new input signal will be waiting at the input ***NEXT*** until the current loop parameters are stored in

memlet ***LoopParams***, thus finalizing processing of the current input signal before starting to process a new one. This shows importance of sequential processing of input signals.

3.2.3.16.3. Recursion

There is a saying: “To iterate is human, to recurse, divine”. So, the next topic after iteration is going to be recursion. It is not possible to implement recursion in Pipe the same or even a similar way to how it is done in non-visual programming languages, because executable components in Pipe (runlets) are static, while recursion requires creating new stack frames via method calls as recursion progresses. However, a deeper look on how methods are used in recursion reveals that recursive calls are made mostly to hold separate stack frames and that’s what makes the recursion possible. There is nothing preventing from externalizing stack frames in Pipe by creating a stack-emulating collection, where each element represents a single stack frame (all local variables plus input parameters). Adding new element to a collection will emulate new recursive method call and removing element for the collection – returning from the recursive call. However, Pipe recursion diagram will be much bigger than equivalent non-visual code. On the other hand, recursion is not used very often – it is mostly part of fundamental algorithms. Recursion is almost non-existent in business algorithms. Therefore, it is recommended to create a prime or scripted runlet using a non-visual programming language in a rare case when recursion is required. Planned dynamic runlet feature (see 3.2.3.19.4 “Dynamic Runlets”) will make it possible to implement recursion in Pipe in a way similar how it is done in non-visual programming languages.

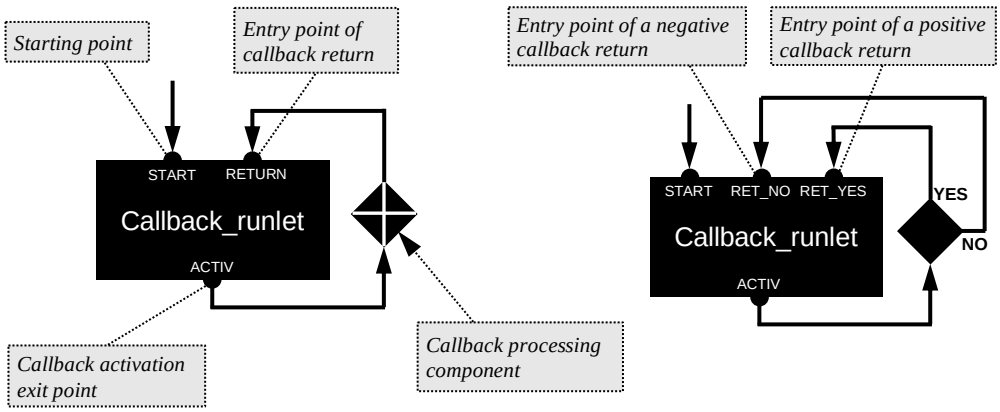


Figure 63. Callback implementation in Pipe.

3.2.3.16.4. Callback

Callback is a reference to a code block passed as an argument to a function or method to be invoked later. Callback pattern is used by lambdas, closures, or function objects in different non-visual programming languages. A simple example of a callback in C++ is provided in the Figure 64, where callback is implemented as a pointer to function.

```
using f_int_t = int(*)(int);

int foobar (int x, f_int_t moo) {
   return x + moo(x); // function pointer moo called using argument x.
}
```

Figure 64. Callback implementation in C++.

It is easy to implement callbacks in Pipe by providing an output for activating callback workflow, and input to get back from the callback (see left example in the Figure 63). There can be two inputs instead of one for accepting **true** and **false** callback result in case of a boolean-returning callback (see right example in the Figure 63).

The problem of callback implementation in Pipe is handling case when callback output and input are not connected, meaning the control flow of a callback is disrupted. The closest analogy for such case in non-visual languages is a callback method that runs indefinitely inside, but the likelihood of a mistake not connecting callback workflow in Pipe is much higher than introducing a callback method code running indefinitely in a non-visual language. Planned dynamic runlet feature (see 3.2.3.19.4 “Dynamic Runlets”) will make callback implementation more reliable by not requiring for an external workflow to be used as a callback, but rather making callback invocations inside of a runlet, similar to how callbacks are implemented in non-visual programming languages.

3.2.3.16.5. Business Case

Let’s explore Pipe implementation of a simple business case – calculating interest amount by multiplying interest rate, balance, and time duration. For simplicity, the history of interest rates and history of balance amounts are out of the scope – an assumption is that those features are externalized out of a composite runlet implementing interest calculations. The diagram should cover the following operations (a corresponding input name is provided in parentheses after operation name):

- **Initial deposit (DEPOSIT)** – storing initial interest rate, initial balance, and today’s date as a starting time for future interest calculations. No interest calculation is performed.
- **Interest calculation (CALCULATE)** – calculating interest amount with the current interest rate and current balance for the period since the last operation.
- **Interest rate change (RATE)** – storing new interest rate. Outstanding interest amount with the previous interest rate and current balance for the period since the last operation must also be calculated.
- **Balance amount change (BALANCE)** – storing new balance. Outstanding interest amount with the previous balance amount and current interest rate for the period since the last operation must also be calculated.

A composite runlet containing the pipeline has four inputs according to the number of operations described above. The runlet has one output for the calculated interest amount. The diagram of the pipeline is provided in the Figure 65.

The pipeline has four inputs ***DEPOSIT***, ***CALCULATE***, ***RATE***, ***BALANCE***, and one output ***INTEREST*** for the calculated interest. All pipeline inputs are connected to inputs of memlet ***InterestRate*** holding current interest rate, and to memlet ***Balance*** storing current balance amount. However, different bond types are used for connecting to those memlets. Pipeline input ***DEPOSIT*** connects to both ***InterestRate*** and ***Balance*** memlets via a write bond type, storing new interest rate and balance amount without producing any output from ***InterestRate*** and ***Balance*** memlets, thus not triggering interest rate calculations.

Other pipeline inputs (***CALCULATE***, ***RATE***, ***BALANCE***) do trigger interest calculations. Input ***CALCULATE*** enters memlets ***InterestRate*** and ***Balance*** via a read bond type, pushing out current rate and balance amount to calculate interest for the period till today's date. Pipeline input ***RATE*** enters memlet ***InterestRate*** via a push bond type (writing new value and pushing previous one) also entering memlet ***Balance*** via a read bond, thus not changing balance amount and pushing current value out. Pipeline input ***BALANCE*** does the opposite, entering memlet ***Balance*** via a push bond type (writing new value and pushing previous balance value out), and entering memlet ***InterestRate*** via a read bond type, not changing interest rate value while pushing the current rate value out. Push (not direct) bond type is used for inputs of ***InterestRate*** and ***Balance*** memlets when receiving new interest rate or balance from inputs ***RATE*** and ***BALANCE*** respectively, because interest rate needs to be calculated from the period **before** interest rate or balance is changed and therefore, previous values must be used. As a result, push bond type must be used in this case.

All pipeline inputs are also connected to input of the mutator ***GetCurrentDate*** to calculate and store date of interest calculation in memlet ***LastOpDate*** (input separation for ***GetCurrentDate*** bonds prevents flow of signals from a connection attached to one bond into connections attached to other bonds). Domains of all pipeline inputs contain a constant boolean node ***WillCalc*** defining if interest rate calculation will be triggered in the pipeline. The node ***WillCalc*** has value **false** for input ***DEPOSIT***, preventing pushing out value from the memlet ***LastOpDate*** by going via output ***NO*** of a tester ***ShouldPush***, entering memlet ***LastOpDate*** via write bond and therefore, not triggering interest calculations. The node ***WillCalc*** has value **true** in all other pipeline inputs, triggering interest calculations by going via output ***YES*** of a tester ***ShouldPush***, entering memlet ***LastOpDate*** via a push bond type and pushing out its previous value out. Domains of all pipeline inputs also contain placeholder node ***OperDate*** to directly provide a node to store current date without the need to use transformers for adding new domain node ***OperDate***. Domain of a memlet ***LastOpDate*** contains placeholder node ***Duration*** used by a mutator ***GetDuration*** to calculate duration between the current date and the date of previous interest calculation.

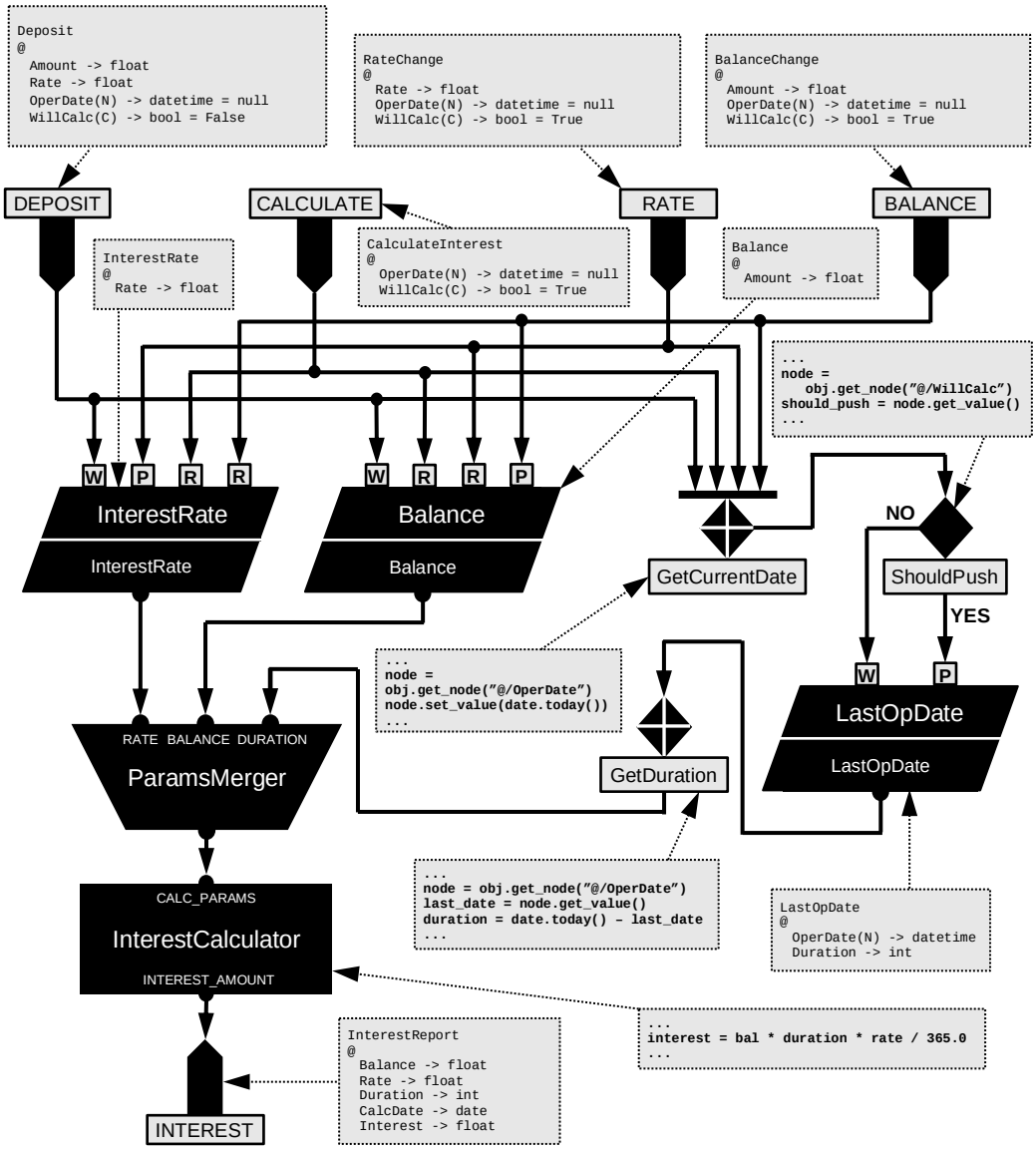


Figure 65. Interest calculation diagram.

The workflow above is designed in such a way that if interest calculation should be triggered by design, then signals always arrive to all inputs of the merger ***ParamsMerger*** together, or no signal arrives to any of the merger input in case when no interest calculations should be performed. If interest calculation is triggered, then three signals with data objects containing balance amount, interest rate and time duration respectively enter distinct inputs of a merger ***ParamsMerger***, and then the merged object is sent to runlet ***InterestCalculator*** for calculating the interest amount. Mutator cannot be used in places of runlet ***InterestCalculator*** because an interest calculating component must have explicitly specified domains and they should be different for input and output, while

mutator automatically uses merged domain produced from all domains of connections attached to inputs of the mutator (see 2.1.4.1.21.1 “Domain Assignment Scope” and 2.1.4.1.21.2 “Domain Assignment Methods”), and the same domain is used for mutator output too. Domain ***InterestReport*** of the output port ***INTEREST*** contains both calculated result and all parameters (rate, balance, duration, calculation date) corresponding to the calculated interest.

3.2.3.17. Most Frequent Issues

There are many issues that may cause crash or freezing of a running Pipe application and developers must be prepared to handle such cases properly. The list below describes the most frequent issues in a decreasing order of probabilities.

Input queue overflow. Every Pipe component has input queues where incoming signals are waiting before entering a component for processing. Queue overflow might happen for input queue configurations allowing potentially unlimited number of signals to be accumulated: common input queue of non-merger components, and separate input queues of a merger.

Accumulation of signals in a common input queue of a non-merger component might happen if a signal processing time is significantly larger than the rate of signal arrivals to the component, or a signal is stuck inside of the component due to a deadlock or an infinite loop, thus preventing the next signal from entering the component for processing while new signals keep arriving to the input queue. Pipe virtual machine crash caused by not enough memory to hold all accumulated signals can be avoided if a maximum queue limit is supported and actively monitored by the Pipe virtual machine during application execution. It is recommended not to stop, but to suspend execution when the maximum queue length is reached, keeping application state intact during suspension and giving developers an opportunity for “live” issue investigation by probing different points of a suspended application.

Signals might also accumulate at merger inputs. The most likely reason for such issue is the rate of signal arrival to one merger input significantly exceeding the rate of signal arrivals to other inputs, resulting in a rapid accumulation of signals on a problematic input due to them waiting longer for arrival of matching signals to other inputs to trigger vector collapse with removal of signals from merger inputs for processing (see 2.1.4.2.1.3 “Merger Processing Algorithm”). Another reason might be a performance issue, when components supply signals to some merger inputs too slowly, while other inputs already have signals in the queue, and they are waiting longer for matching signals to arrive. The best way to avoid queue overflow for merger inputs is by minimizing differential of a signal arrival rates between merger inputs to mitigate imbalance of signal accumulation on merger inputs.

Usage of general bond attributes other than a regular attribute (front, back, or reset attribute – see 2.1.4.5 “General Bond Attributes”) for runlet inputs guarantees that no more than one signal will be accumulated for signals coming from bonds with such

attributes. However, semantics of corresponding component inputs should allow usage of such attributes.

Stale signals. Pipe allows assigning priorities to input signals. It means that the signal can stay in an input queue for a very long time and potentially even infinitely if signals with higher priorities constantly keep arriving to the queue. The best solution for this problem is a better signal priority planning and usage of signal priority feature only when it is really needed.

Unconstrained processing time. Runlets process input signals one-by-one, meaning other signals must wait for processing while the runlet is in processing state. The longer processing lasts, the more signals can potentially accumulate in input queue, eventually causing queue overflow. One of the biggest mistakes made during runlet construction is waiting for asynchronous events during signal processing syncronously without finalizing signal processing, effectively making signal processing time unconstrained – for example, when signal processing relies on a keyboard input from user for completion of processing. It is a mistake to wait for keyboard input while keeping processing in progress. Instead, runlet must complete processing without generating output signals related to receiving keyboard input, while a separate internal signal will be triggered after arrival of an asyncronous event of key pressing, producing output signals exactly at that time. As a result, waiting for asynchronous event of keyboard input will not block processing of the next signal.

Race condition. Race condition happens when the order of arrival for two or more signals to a specific point of the workflow is not deterministic while the overall result of the workflow depends on a specific order of signal arrivals. An example of race condition is provided in the Figure 51**.** The most likely root cause of race condition in Pipe is a workflow design error.

Unhandled exceptions. There are two types of exceptions in Pipe: native exceptions and Pipe exceptions. Pipe exceptions may be intercepted and handled by traplets (see 2.1.4.3 “Handling Exceptions”). Native exceptions are non-visual language specific exceptions thrown inside of a coded runlet and Pipe cannot intercept them outside of the runlet, so they must be caught inside of the runlet, converted to a Pipe exception and re-thrown using runlet API (see 2.5 “Runlet API”).

Infinite loop. Infinite loop might happen both in workflow diagrams (because Pipe allows connecting output and inputs of the same runlet), and inside of coded runlets. An infinite loop is a result of a diagram design error or a coding mistake. It may cause an input queue overflow for a component running an infinite loop inside.

3.2.3.18. Pipe Standard Library

3.2.3.18.1. Introduction

Standard library is not a part of this version of the Pipe language specification (see 2 “Language Specification”), but it will very likely be included in the next Pipe language

versions. Some ideas about what should be included in Pipe standard library are provided below. The reason they are not part of Pipe is because these ideas require further analysis and detailed design before adding them to the language.

3.2.3.18.2. Loop Component

The first very important component that should be a part of a standard library is a generalized loop runlet, because iteration is a fundamental part of almost any algorithm. This book discusses Pipe loop implementation in 3.2.3.16.2 “Iterations”, and those ideas can be used for creating loop component for a standard library.

3.2.3.18.3. Collections

The second important part of standard library is collection support. Pipe domains contain only array-like collection type without dedicated sets/dictionaries data types, because sets and dictionaries might easily be implemented using array data type (sets can be implemented as arrays of keys, and dictionaries – as arrays of key/value pairs using a domain data type for holding them). This will be a structural collection implementation and not behavioral (for example, search by keys will not be available for dictionaries), because Pipe does not support domain methods. It means sets and dictionaries will have to be loaded into appropriate collection object inside of a coded runlet every time collection data arrives to the runlet, but this is not the only problem of collection support in Pipe.

The main problem of passing collections via data objects is related to question about the most optimal way of implementing data transfer operations in Pipe. Any reasonable Pipe implementation will inevitably follow these data transfer rules (perhaps with some variations):

1. All storage node values will be passed internally by reference during data transfer without any data copying to avoid overhead of data copy operation, especially for storage nodes with variable data size such as strings, arrays, and binary blocks.
2. The references to storage node values will be immutable to avoid situations when changing value in one data object impacts other data objects received the same value after transfer.
3. Eventually, a storage node value may need to be changed inside of a coded runlet in some cases. To avoid impacting other data objects, the value of a storage node must be fully copied to a separate memory area before making the first change (copy-on-write approach).

Rule 3 in the list above shows that attaching large collections to data objects is not an optimal method of data transfer, because it entails a high risk of full data copying in case of node value modifications. Standard library would provide an alternative method – named collection and referring to such collections by name strings passed between

components instead of passing collection data directly. As a result, data will be passed only by reference when using named collections. For collection modifications within a local scope (without impacting other collection name holders), the named collection must still be copied explicitly, but only selected elements might need to be copied into new collection in many cases, thus decreasing the scope of copy operation.

Standard library would have to provide full range of operations for named collections: creating new named collection of specific type (array, set, dictionary, etc.), adding and deleting an element, searching element by key or index. The support must be implemented both in visual (runlets) and non-visual (extension of a runlet API) levels to be able to use named collections in both runlet types.

3.2.3.18.4. Math

There are two possible ways to implement math in Pipe and standard library should ideally include both methods.

The first method is a set of runlets for atomic mathematical operations. For example, trigonometric runlet would have multiple inputs where each one corresponds to a specific trigonometric function – sine, cosine, tangent, etc.

The second method is making a generic expression evaluation runlet calculating the result from a supplied math expression tree. Different outputs might be required for different types of expressions. For example, two logical outputs (**false** and **true**) will be required for a boolean expression evaluator, which can be used as a non-coded replacement of a Pipe tester component for basic and intermediate levels of Pipe usage (see 2.3.6 “Non-Coded Testers”). The following domain can be used for describing a math expression tree in a generic evaluation runlet:

```
MathExpression                          # Domain for describing an expression tree.
@
   Op -> int                            # Code of a math operation (+, -, …) or code of
                                        # a math function (sin,log, …).
   LeftOperand(N) -> MathExpression     # Left operand must always be present unless
                                        # it is leaf.
   RightOperand(N) -> MathExpression    # Right operand can be null in case of unary
                                        # operation.
```

3.2.3.19. Future Development

3.2.3.19.1. Introduction

The current version of Pipe language includes a reasonably sufficient set of features. However, there are many more features that were left out of the current version for possible implementation in future versions.

3.2.3.19.2. Enumerations

One apparent feature missing in Pipe language is enumerations. The basic structure of an enumeration specification will be close to the structure of a domain specification. Top part of enumeration specification will contain domain structure of the enumeration values – similar to a structure definition in a domain specification. The line starting with the

keyword “@CASES” is a separation line between top part of enumeration specification defining structure of enumeration, and bottom part containing all enumeration values (cases). The name of enumeration and names of all enumeration values must start with a dollar character (“$”). Value of each enumeration case is specified as a domain-expanded value of the enumeration after equal character (“=”) and with enumeration values enclosed in curly brackets (“{}”), except cases when enumeration is a scalar domain – curly brackets for enumeration values are not needed in such case. Comments in enumeration specifications are shown using the same syntax as in domain and object specifications. For example:

```
$AccountType            # Enumeration name.
@                       # All lines below before the line with keyword “@CASES” define
                        # enumeration structure.
   Code -> int          # Code of account type.
   Name -> string       # Name of account type.
@CASES          # The second part of specification (below) contains enumeration values.
   $Basic = {
      Code = 100
      Name = “Basic”
   }
   $Standard = {
      Code = 200
      Name = “Standard”
   }
   $HighInterest = {
      Code = 400
      Name = “HighInterest”
   }
```

The default value of enumeration is the first case in an enumeration specification. Enumerations can be exported for use in other solutions. Enumerations cannot be assigned to membanks and component pins, but they can be used as domain data types in node definitions. Like domain names in domain data types, enumeration names must be enclosed in curly brackets (“{}”) when referenced in domain specifications. For example:

```
Account
@
   Type -> {$AccountType} = $Basic      # Node type is enum “AccountType”.
   Status -> {$AccountStatus} = $Open   # Another enum “AccountStatus”.
```

Enumerations allowing multiple value combinations can be defined with two-dollar character combination (“$$”) as a name prefix. For example:

```
$$NumberType            # Multiple values enumeration.
@ -> string             # This is a scalar enumeration.
@CASES                  # Enumeration values that can be combined.
   $Positive = “Positive”
   $Integer = “Integer”
   $PowerTwo = “Power of Two”
```

Values of such enumerations can be combined in domain and object specifications using plus (“+”) character. For example: ***$Positive+$Integer+$PowerTwo***.

3.2.3.19.3. Generics

Generics might be implemented for domains the same way they are implemented for classes in non-visual languages. Type parameters will be provided on the first line of the domain specification divided by comma character (“,”) and enclosed in angle brackets

(“<>”). Declared type parameters can be used as data types for storage nodes in domain specification. For example:

```
<T, U>          # Type parameters T and U.
<- Device       # The domain is inherited from a domain “Device”.
<- Component    # The domain is inherited from a domain “Component”.
Monitor         # Domain “Monitor”.
@
   Ports -> T                          # Ports defined by type parameter T.
   Resolution -> ScreenResolution      # Screen resolution.
   MaxRefreshRate -> int               # Maximum refresh rate.
   CustomSpecs -> U     # Set of custom specifications defined by type parameter U.
```

Generics are instantiated into real domains when they are getting assigned to component pins and membanks. Both domain and primitive data types may be used for generic instantiation whenever appropriate. Generic instantiation data types should be enclosed in angle brackets (“<>”) after domain name (underscore character “_” must be used if the domain does not have a name). Actual types used during generic instantiation must replace type parameters for generic domain specifications shown inside of comments. Substituted type parameters in such comments must be shown as **underlined bold**. An example is provided in the Figure 66.

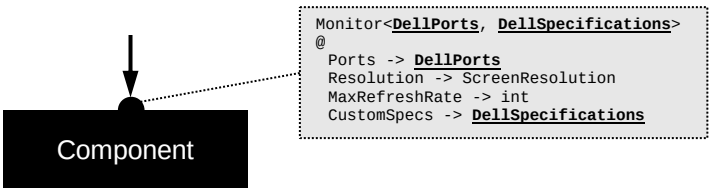


Figure 66. Generic instantiation during its assignment to a component pin.

Pipe generics can use constraints. Each constraint is shown enclosed in angle brackets (“<>”) in lines below the line with type parameters and it contains a type parameter name followed by the list of allowed types (called **constraint list**) for that type parameter, with an arrow character combination (“->”) shown between type parameters and the constraint list. Type names in a constraint list must be divided by a comma character (“,”). Presence of a domain in constraint list means that the domain and any of its inherited domains are allowed for generic instantiation. Both primitive and domain types can be used in a constraint list. For example:

```
<T, U>                          # Type parameters T and U.
<T -> Conductor, Isolator>      # Domain “Conductor”, “Isolator” or their inherited
                                # domains are allowed.
<U -> int, string>              # Primitive types “int” and “string” are allowed.
Material                        # Domain “Material”.
@
   MaterialType -> T            # Material type is defined by a type parameter T.
   ProductCode -> U             # Product code may be either integer or string.
```

3.2.3.19.4. Dynamic Runlets

The difference between functions in non-visual language and runlets in Pipe is that functions have a new local memory context (stack frame) created dynamically during function invocation, while Pipe runlets have static memory context created once during

application startup. This is what makes callback and recursion so easy to implement in non-visual languages, while it is a huge problem to construct callbacks and recursion in Pipe (see 3.2.3.16.3 “Recursion” and 3.2.3.16.4 “Callback”). To solve this problem, the concept of dynamic runlets is proposed.

An idea of dynamic runlets in Pipe is based on a notion of **interface** – abstract specification containing a list of all component pins with a name and assigned domain specified for each pin. Interface name must start with an ampersand character (“&”). Interface can be assigned to a runlet, meaning the component must have all its pins conform to the interface specification. Any number of interfaces may be assigned to a runlet. It is allowed to mix interface specification pins with manually added ones, i.e., a runlet may have pins from assigned interfaces as well as manually added ones. Special rules must be designed to handle name collisions for pins coming from different interfaces assigned to a runlet, and collisions between manually assigned and interface pin names.

Interface specification is a list of pairs where each pair contains pin name and assigned domain. The first line should contain interface name. Each line of the interface specification describes a single pin with a pin name followed by the name of an assigned domain divided by the arrow character combination (“->”). If pin domain is defined in-place (see 2.1.4.11.3.4 “Anonymous Domains”) then underscore character (“_”) is used instead of a domain name, and domain specification is provided on the next line starting from the root (“@”) node, with one indentation relative to line above. Interface specification may have comments with the same syntax as for comments in domain and object specifications. Interface specification is divided into two parts: input section providing specifications for input pins, followed by an output section with specifications of output pins. Input section starts from the line below the line with keyword “@INPUTS”, which must be the second line of the specification just below the line with interface name. Output part starts after the line with keyword “@OUTPUTS”. Each input and output pair should be shown with one indentation. For example:

```
&FinancialOperation             # Interface name.
@INPUTS                         # Starting input pins.
   DEPOSIT -> Balance           # New deposit input pin.
   OPEN -> _                    # Opening input pin with in-place domain shown below.
   @                            # Definition of in-place domain for an input pin OPEN.
      Person
         LastName -> string
@OUTPUTS                        # Starting output pins.
   AMOUNT -> Amount             # Amount output pin associated with a domain “Amount”.
```

Two interfaces are called **convertible** if both interface specifications contain the same set of pin names with mutually (both ways) convertible domains between each name-matched pair of pins from two interfaces. Domains can inherit from interfaces.

Dynamic runlet feature is implemented by a special type of a runlet called **placeholder runlet**, which is shown on diagrams as a rectangle with black border, light-grey fill color and black text inside (see Figure 67). Placeholder runlet must be associated with one or more interfaces, automatically showing the full set of pins from specifications of all interfaces it is associated with.

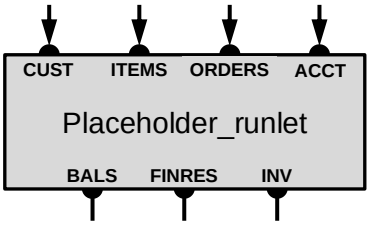


Figure 67. Placeholder runlet.

Placeholder runlet is an empty socket until it starts processing a signal with a data object containing exactly one reference to an existing template runlet or instance runlet, putting referenced runlet into the socket before starting to process the signal. The reference can exist anywhere in a domain structure. Only reusable runlet types are allowed to be referenced. This reference is a physical runlet used temporarily for the duration of processing of a single signal by putting a referenced runlet into placeholder location before sending a signal to it.

There are two signal types a placeholder can process: **template signals** and **instance signals**. Template signal contains a domain node referencing template runlet, while instance signal references an instance runlet. To reference template and instance runlets, two new primitive data types are introduced: **template reference data type** and **instance reference data type**. The value of a template reference data type is a full name of a template runlet. The value of an instance reference data type is a member name (see 2.1.4.11.3.2 “Member Names”) of a runlet located in the same pipeline with a placeholder runlet. Both template and instance reference data types must be associated with one interface. The name of associated interface is shown in domain specification enclosed in angle brackets (“<>”) for template reference, and double angle brackets (“<<” and “>>”) for instance reference. For example:

```
ResultCallback  # Domain “ResultCallback”.
@
   CallbackRef -> <<&CallbackInfo>> # Instance reference via interface “CallbackInfo”.
```

Domains containing a template or interface reference may be used only in interface definitions and domains inherited from an interface. All interfaces assigned to template and instance reference data types in domains assigned to pins of an interface must be compatible with that interface, or with an interface inherited by domain. For example:

```
&Http                           # Interface “Http”.
@INPUTS
   REQUEST -> _                 # Input pin “REQUEST” with in-place domain definition.
   @
      HttpRef -> <<&Network>>   # Interface “Network” must be compatible with interface
                                # “Http”.
```

When placeholder runlet receives a template signal, it creates a new instance runlet according to referenced template runlet before starting to process a signal and destroys the instance runlet after finishing processing. Startup beacons inside of the referenced template runlet will be triggered after new instance runlet is created, and shutdown beacons will be triggered before instance runlet deallocation. Created new instance runlet

for a template reference, or directly used instance runlet in case of an instance reference are called a **target runlet**.

Placeholder runlet routes received input signal to corresponding input of a target runlet with the same input name as an input name of the placeholder runlet that received the signal. Placeholder runlet also routes each signal emitted by a target runlet output to a corresponding external output of the placeholder runlet with the same name.

Dynamic runlets make implementation of callbacks more reliable in Pipe by using a template reference (for stateless callbacks) or instance reference (for stateful callbacks), similar to interface-type arguments in non-visual languages utilized to pass callbacks as objects conforming to a specific interface.

Dynamic runlets also make possible recursion implementation in Pipe by using a self-reference of the current runlet as a template reference, thus creating a new memory frame for each template reference instantiation by invoking internal placeholder runlet in a context of instantiated current (parent) runlet's template passed via a template reference. Dynamic runlets with template references can be used for handling expensive resources that must be allocated for a short period of time (i.e., for the duration of processing), guaranteeing their deallocation after the end of the processing via shutdown beacons. Self-reference is not allowed for instance references to prevent deadlock.

Dynamic runlets complicate application debugging and make it more difficult to understand the logic of the flow, so they should be used with caution.

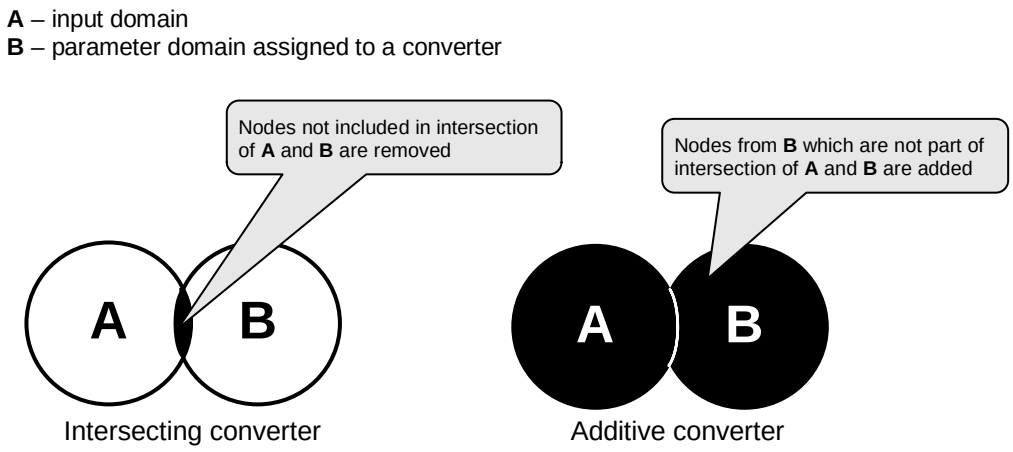


Figure 68. Intersecting and additive converters.

3.2.3.19.5. Converters

Transformers are used for modifying domain structure. Domain transformation can also be achieved by intersecting two domains. However, the resulting overlap domain is not available for use outside of the domain intersection context (i.e. outside of a connection producing the overlap). Therefore, a simplified version of a transformer called **converter** is proposed that modifies incoming domain by intersecting it with another domain assigned to a converter. Converters fall into a category of fixed runlets.

Converter has one input and one output. There are two types of converters: **intersecting** and **additive**. Intersecting converter changes input domain by intersecting it with assigned **parameter domain** and producing overlap as an output domain. Additive converter adds parameter domain nodes to an input domain that are not included in the overlap of the input and parameter domains (see Figure 68).

A converter may not have any parameter domain specified – intersecting converter will be producing blank signals for any input domain in this case, and original domain will be produced in case of an additive converter.

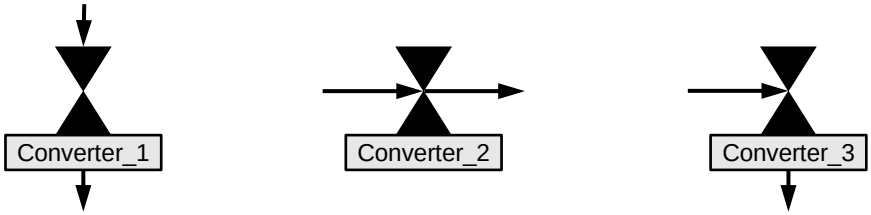


Figure 69. Converters.

Converters are shown on diagrams as vertical hour-glass shapes with two triangles on top of each other, where top triangle is shown upside-down horizontally centered. Member name is shown as a black text in a rectangle with a black border and light-grey fill color. The member's name rectangle is located at the bottom of the converter horizontally centered with the component shape. Component input and output may attach to top, to bottom under name rectangle and to center point both to the left and right (see Figure 69).

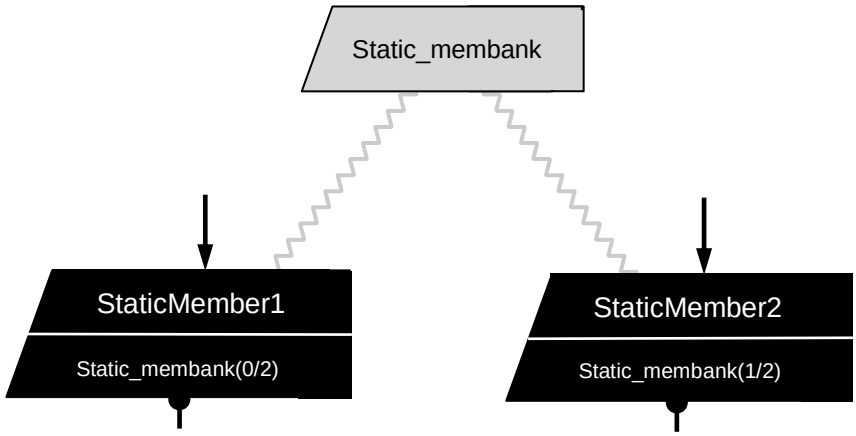


Figure 70. Static membanks and memlets.

3.2.3.19.6. Static Membanks

A membank defined in a template runlet produces separate data storage objects in each instance runlets. However, proposed **static membank** share the same storage memory

among all instance runlets. Static membanks and all their linked memlets have the same shape as regular ones, but with a straight right side (see Figure 70).

3.2.3.19.7. Membank Sharing

Membanks are confined within a scope of a specific pipeline (they are called **local membanks**). Their sharing between pipeline is not a trivial task, because any composite runlet sharing its non-static membanks is most likely a part of a library project. It means that shared non-static membank represents not a single but many instances depending on a number of instance runlets in a solution created from the root runlet of this library project. Therefore, references to non-static membanks from such composite runlets are meaningless as each such reference would represent not a single but many instances. However, there are some solutions allowing membank sharing with some limitations.
There are two proposed ways of membank sharing: **horizontal** and **vertical**. Both methods use some common terminology. Composite runlet owning and sharing its membanks with other composite runlets is called **membank provider**. Other composite runlets using shared membanks in their pipelines are called **membank consumers**.

Horizontal method of membank sharing can be used only for static membanks. Membank provider defines a subset of its static membanks available for horizontal sharing anywhere in the current solution and in other solutions after importing membank provider to these solutions. Membank consumers create horizontal dependencies with their membank providers and therefore, exporting membank consumers should automatically include their membank providers too. As horizontal method shares static membanks always existing as a single instance, it does not create ambiguity on a side of a membank provider.

Vertical method of membank sharing is available only for the following membank consumers:

- Direct or indirect children of a membank provider.
- Defined in-place.

Membank provider defines subset of membanks from its direct or indirect child runlets as available for vertical sharing. Membank provider and all its consumers are always going to instantiated together during instantiation of their parent runlet because all membank consumers are children of their membank provider, always creating 1:1 relationship between a pair containing provider and any of its consumer in every parent runlet instantiation. Also, membank consumers are defined in-places meaning they can never be instantiated independently, outside of being a child of their membank provider.

Membank consumers can reference membank providers (such membanks are called **membank references**) by having references to membank providers inserted into consumer's pipelines. Membank references are shown in diagrams of membank consumer's pipelines as top half side back-sloped and bottom half side forward-sloped both for membank and all connected memlets (see Figure 71). Membank references must have local names assigned to them within the scope of a pipeline they are going to be inserted into.

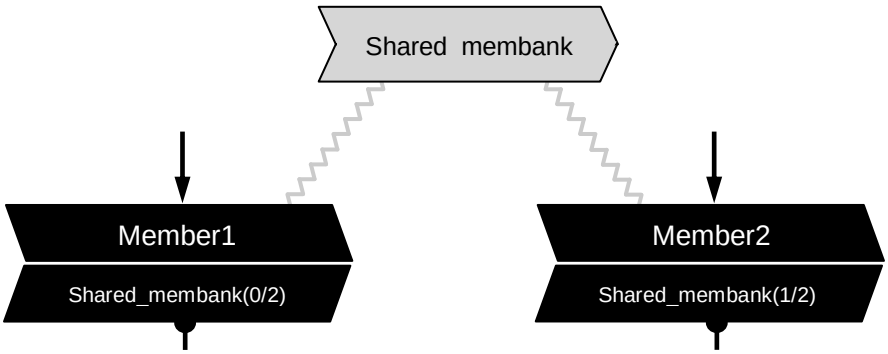


Figure 71. Membank reference and its memlets.

3.2.3.19.8. Selector

Selector is a fixed runlet type for routing input signal to one of outputs based on a certain condition. This component is visual equivalent of switch statement in non-visual languages. Selector is shown as horizontally oriented capsule-like black shape. The component has one nameless input along the top edge, and no less than one named output along the bottom edge (see Figure 72). Selector uses pin junctions for inputs and outputs.

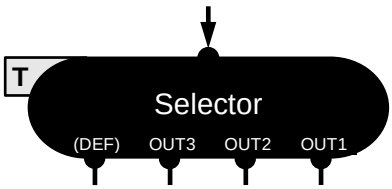


Figure 72. Selector.

Selector must route input signal to no more than one output based on a specified **output condition** (the condition is specified for each output using combination of input data object values and must be true for an output to produce a signal), unless the component has a **default output** producing output if no other output would produce it – exactly one output must be produced by a selector for each input signal in that case. Selector is not allowed to change data object contents or structure, but it can read data object contents to route it to proper output. Default output has the name “DEF” in parentheses (see Figure 72).

Selector has a component-level domain assignment scope with the same domain assigned to all its pins. It uses default assignment method for input (input domain is a merge of domains for all inputs), and auto assignment method for outputs (all outputs inherit their domains from input).

There are two types of selectors: **coded** and **tabular**. Coded selector contains Python code implementing signal routing by using runlet API. Tabular selector uses structured **routing table** to implement signal routing. Coded selector is indicated by a badge with letter “C”, and tabular – with letter “T” (see Figure 72).

Tabular selector routes input signal based on a routing table that has two parts: header and body. Header contains full path to a storage node of an input domain. Value of this node is used to route the signal. Header can optionally contain the name of an alias assigned to the node when referred from routing table body. The alias name should follow the same rules as common names. If alias is not specified, then the name of the last node from the path is used as an alias.

Routing table body contains a list of **routes** corresponding with selector outputs – one route per output, excluding default output if it specified for the selector. Each route has the following attributes:

- **Position** – index of the route in the routing table body. Routes are evaluated in an increasing order of route positions. The first evaluation returning true produces signal from a corresponding output, and further evaluations stop.
- **Condition** – condition that must be evaluated to decide if output must produce a signal in case the condition is true. The condition must use the following components:
    - Alias assigned to an input domain node in header. It is allowed to refer to any child node starting from the alias with dot character (".") as a divider between nodes.
    - Grouping subexpressions using parentheses.
    - Constants.
    - Boolean operations: AND, OR, NOT.
    - Comparison operators:
        - Equal (==) and not equal (!=).
        - Less (<) and more (>).
        - Less or equal (<=).
        - More or equal (>=).
    - Math operators and functions, but only after domain specifications expressions are implemented in future versions of Pipe – see 3.2.3.19.11 "Domain Specification Expressions".
- **Output name** – name of an output generating the signal if the evaluated condition is true.

An example of routing table is provided below:

Header	
Path to domain node	**Alias**
@/Factory/Specification/Product	Prod

Body		
Position	**Condition**	**Output Name**
1	Prod.Weight < 1	LIGHT
2	Prod.Weight >= 1 AND Prod.Weight < 100	MEDIUM
3	Prod.Weight >= 100	HEAVY
4	Prod.Weight >= 100 AND Prod.Color == “Red”	HEAVYRED
5	Prod.Weight >= 100 AND Prod.Color == “Blue”	HEAVYBLUE

As further development of the selector component, a special subtype of a coded or tabular selector can be introduced called **multi-selector** that produces signals from not one, by from all outputs where condition is evaluated as true.

Selector is a generalization of tester. Tabular selector is a much more user-friendly option for lower levels of usage. Selector can potentially replace tester in Pipe language.

3.2.3.19.9. Read-Only Attribute

Storage domain nodes might have a constant attribute assigned to them (see 2.1.4.1.5 “Node Attributes”). However, constant value must be assigned explicitly or implicitly (as a default data type value) to these nodes in domain specification. The constant values are defined during design-time and the current version of Pipe does not allow defining a storage node that becomes non-mutable only after the first value is assigned to it. Therefore, read-only node attribute is proposed for Pipe language. Read-only attribute is the same as constant attribute, except the first value assignment is allowed and that value becomes a constant value of the node after the first assignment. Read-only attribute follows the same multiple attribute assignment rules as for constant attributes (see 2.1.4.1.7 “Multiple Node Attributes”). Read-only attribute is shown in domain specifications as two-letter combination “RO” in parentheses after node name. For example:

```
Account                          # Domain “Account”.
@
   AccountType(RO) -> int        # Account type node has read-only attribute.
   Holder -> {PersonInfo}
```

3.2.3.19.10. Memo Attribute

New memo domain node attribute is proposed for reusing a value from previous overlap if the following overlap does not provide new value for the destination node. The destination node will still be initialized by default value for the first overlap, but the previous most recently assigned value will be reused in case of any following overlap not providing a value for the destination node. Two-letter combination “MM” in parentheses after node name should be used to indicate memo node attribute in domain specifications. For example:

```
Balance                          # Domain “Balance”.
@
   Amount(MM) -> int             # Balance amount with memo attribute.
   Currency -> Currency          # Currency of the balance.
```

Domain assigned to a component and containing any node with memo attributes becomes part the component state (see 2.1.4.4 “Component State”). Memo attribute has effect only when located on destination side of a domain overlap and therefore, attribute modifiers are not applicable for memo attribute (see 2.1.4.1.20 “Attribute Modifiers”).

3.2.3.19.11. Domain Specification Expressions

Current version of the Pipe language does not allow expressions to be used for default and constant values of domain nodes. Therefore, it is proposed to add support of mathematical operations (+, -, /, etc.) and a set of functions for default and constant domain node values. For example:

```
Processor                              # Domain “Processor”.
@
   Model(C) -> string = Concat(“MS”, “12”) # Constant node with model as string concat.
   Frequency(C) -> int = 4 * 1000 * 1000 * 1000 # Constant node with frequency in GHz.
```

All math operations and functions must be fully described in Pipe language specification without any room left for arbitrary non-compatible dialects introduced by specific Pipe implementations.

3.2.3.19.12. Multiple Viewpoints

Pipe allows attaching any number of comments to the same diagram element. Each comment may be intended to one of several stakeholders of the project. For example, text comment for an element might be intended for a citizen developer, while code comment – for professional developer. Putting many comments on diagram may take significant amount of space – for example, take a look at a diagram at Figure 65, where code comments do not leave any spare space for more elements, so it would require significant stretch of diagram size to make some space for new comments. Therefore, it is proposed to implement viewpoints – diagram layers differing by comments only. Each comment must be associated with one of more viewpoints. User can create an arbitrary number of viewpoints, switching between them to see proper comments. For example, there might be one viewpoint for citizen developers containing generic text comments, and another for professional developers containing code comments. Positions of all elements other than comments must stay the same in all viewpoints, and changing a position for such element in one viewpoint must be automatically reflected in all other viewpoints.

3.2.3.19.13. Other Ideas

Additional ideas for future Pipe development are provided below in a general form, without providing more details at this time:

- **Two-way connections.** There are many cases when a communication between two components utilizes “request-response” pattern. Two separate connections are needed to implement such patten in current version of Pipe language. The idea is to introduce two-way connection type that allows sending both request and response over the same wire.

- **Bus connection type.** Bus is a special connection type where all components connected to the bus can communicate with each other. A signal sent to the bus will be received by all other components connected to the bus, or by a selected set of connected components according to the recipient list defined by a sender. A component can have a role of sender, receiver, or both sender and receiver when connected to a bus.
- **Node reference data types.** Currently, values of domain nodes are always isolated from each other. However, there are cases where reference to some part of domain tree might be needed to refer to data present elsewhere in a domain tree. This will allow sharing the same data between two or more nodes. A special node reference domain data type would provide such feature, referencing another node in the same domain tree by its path.
- **Signal priority auto assignment.** According to the current specification, signal priorities can optionally be assigned inside of a coded runlet when posting the output signal. However, signal priority can also be assigned automatically and there are several places where it can be done. First, component outputs can have priorities assigned, meaning that any signal sent to the output takes such priority. Second, signal priority can be auto assigned right on a wire at a certain location of a connection, and all signals passing connection at that location takes priority assigned to that location. Third, component inputs can have priorities assigned to them, meaning that any signal arriving to such input takes a priority assigned to the input.
- **Global beacon priorities.** Priorities of startup and shutdown beacons can be specified within one pipeline only, and their firing order between different pipelines is not defined. Therefore, the feature of defining global beacon priorities across the whole application would provide capabilities to specify certain application-wide order of beacon firing.
- **Runlet inputs for internal signals.** Processing of internal signals inside of a composite runlet starts from an output of a component inside of the runlet. However, a special internal input can be defined for a runlet. All internal signals for such runlet will start propagation during processing from that internal input rather from output of internal component produced the signal.
- **Custom domain transformations.** Transformers provide a fixed set of predefined domain transformations. However, there might be situations when none of provided options would allow to achieve a desired domain transformation. Additional custom domain transformation option will be provided for such cases, where arbitrary algorithm can be implemented using Python code attached to the translet of a corresponding new translet type. The code will be consuming an extension of a runlet API for domain transformations.

- **Domain deanonymization.** Domains of connection overlaps and output domains of transformers/mergers are anonymous and therefore, they cannot be reused anywhere else. Domain deanonymization feature allows assigning special names to connections and outputs of transformers/mergers, thus making these domains reusable in the same solution by referring to assigned name, and even in other solutions after exporting these names.
- **Multiprocessing runlets.** Runlets process input signals one-by-one. However, new option to run more than one signal processing at a time might improve application performance. A close analogy for this feature is autoscaling concept in cloud computing, where multiple instances are allocated for parallel processing of requests. To process several signals simultaneously, runlet must create separate copies of a starting state (state at the beginning of signal processing) for each processing signal. Therefore, this option is applicable only to self-reset runlets (resetting their state after finishing up signal processing), because otherwise runlets must keep many separate runlet states by splitting the state of the same runlet. However, Pipe is designed with assumption only one state for each runlet at a time. Also, a configuration option is needed for specifying maximum number of signals a runlet can start processing at the same time. Output signals will not be generated in the same order as consumed input signals as a result of their parallel processing, so external workflows must account for such behavior.
- **Signal processing retry.** If signal processing results in throwing an exception inside of a runlet, retry feature would implicitly catch such exception at the runlet boundary and the signal is going to be automatically placed back to input queue to retry its processing again later. Retry option can be set for specific signal, for a single input or for the whole component, along with parameters specifying minimum required delay for a signal to wait in input queue before starting its processing again, and maximum number of attempts. The delay can be a constant value or defined as some formula – for example, exponential backoff formula doubling waiting time after each retry. If signal processing fails after all attempts, it is going to be automatically placed in a special failure queue created per component, per pipeline or for the whole application. Signals in failure queue can be pushed back for the retry process again manually.
- **Event handlers.** Certain events happening inside of running application can have coded event handlers – for example, event of signal arrival to component input, start of signal processing, production of output signal, signals merging, signal transformation, etc. All or some of these events may have Python-coded event handlers attached to corresponding elements and performing additional custom processing.

- **Reusable in-place runlets.** Currently, in-place runlet cannot be used anywhere except place where it is defined. The proposal is introduction of a new type of in-place runlets that can be reused anywhere in the same and possibly other solutions. However, reusable in-place runlets will not be able to use vertical method of membank sharing.
- **Static domain node attribute.** New static domain node attribute is proposed that shares the same value among all domain node instances. Example of node attribute usage is the case of paramlet domains where static domain nodes can hold information about runlet templates, not instances.
- **More scripting languages.** Currently, only Python is allowed for scripting runlets. However, the choice of a scripting language can be expanded to several alternatives on top of Python – for example, PHP, JavaScript, etc.
- **Integration with ER and UML.** ER (Entity-Relationship) data modelling is the most successful and very widely used method of software visualization used in computer industry. Unfortunately, it covers only data structures. It would probably be useful to try to integrate Pipe language and ER data modelling. It can be done by having Pipe diagrams as an extension of ER models. Pipe workflows will be triggered by data changes in an ER model. Connectivity between ER model and Pipe workflow will be indicating where and how exactly changes in data structures trigger workflow execution. Pipe can also potentially be integrated with UML.

3.2.3.20. Final Notes

3.2.3.20.1. AI Code Generation

AI code generation creates a perfect synergy with visual programming as AI-generated code can be encapsulated within visual blocks. Such approach would empower citizen developers, making them significantly less dependent on a professional development help. Evolution of a Pipe integration with AI code generation technology will go through three stages. The first stage assumes no usage of AI and creating all non-visual code manually. The second stage is using AI to generate non-visual programming code. The current state of AI code generation technology already makes such option available. The third stage comes after AI grows to the level where it can build Pipe diagrams by itself, thus significantly simplifying analysis and understanding results of AI generation by reading visual diagrams instead of a text-based code.

3.2.3.20.2. Low-Code Platforms

AI-assisted generation of software components followed by their integration via visual programming language can inspire the next generation of low-code platforms. The problem of existing platforms is their low versatility as they offer only a fixed set of predefined components with limited customization. The next generation will not be

constrained by these limitations, allowing generation or modification of any component using AI. Also, a common visual programming language Pipe will be used for integration of generated components, solving the problem of portability between different low-code platforms.

Index

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Font Licenses and Copyrights

Liberation Fonts (Liberation Serif, Liberation Sans, Liberation Mono)

Licensed under the SIL Open Font License, Version 1.1.

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: http://scripts.sil.org/OFL

SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007

PREAMBLE The goals of the Open Font License (OFL) are to stimulate worldwide development of collaborative font projects, to support the font creation efforts of academic and linguistic communities, and to provide a free and open framework in which fonts may be shared and improved in partnership with others.

The OFL allows the licensed fonts to be used, studied, modified and redistributed freely as long as they are not sold by themselves. The fonts, including any derivative works, can be bundled, embedded, redistributed and/or sold with any software provided that any reserved names are not used by derivative works. The fonts and derivatives, however, cannot be released under any other type of license. The requirement for fonts to remain under this license does not apply to any document created using the fonts or their derivatives.

DEFINITIONS "Font Software" refers to the set of files released by the Copyright Holder(s) under this license and clearly marked as such. This may include source files, build scripts and documentation.

"Reserved Font Name" refers to any names specified as such after the copyright statement(s).

"Original Version" refers to the collection of Font Software components as distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting, or substituting -- in part or in whole -- any of the components of the Original Version, by

changing formats or by porting the Font Software to a new environment.

"Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions:

1. Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself.
2. Original or Modified Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.
3. No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.
4. The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.
5. The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION This license becomes null and void if any of the above conditions are not met.

DISCLAIMER THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Nunito

Copyright © 2011, Vernon Adams (vern@newtypography.co.uk), with Reserved Font Name "Nunito".

Copyright © 2014 The Nunito Project Authors (https://github.com/googlefonts/nunito).

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: http://scripts.sil.org/OFL

[SIL OFL 1.1 license text as above in section "Liberation Fonts"]

PT Serif

Copyright © 2009 ParaType Ltd.

FONT LICENSE

PERMISSION & CONDITIONS

Permission is hereby granted, free of charge, to any person obtaining a copy of the font software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the font software, subject to the following conditions:

1) Neither the font software nor any of its individual components, in original or modified versions, may be sold by itself.

2) Original or modified versions of the font software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3) No modified version of the font software may use the Reserved Name(s) or combinations of Reserved Names with other words unless explicit written permission is granted by the ParaType. This restriction only applies to the primary font name as presented to the users.

4) The name of ParaType or the author(s) of the font software shall not be used to promote, endorse or advertise any modified version, except to acknowledge the contribution(s) of ParaType and the author(s) or with explicit written permission of ParaType.

5) The font software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

Carlito

Copyright © 2010–2013 by tyPoland Lukasz Dziedzic with Reserved Font Name "Carlito".

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: http://scripts.sil.org/OFL

[SIL OFL 1.1 license text as above in section “Liberation Fonts”]

Libre Caslon Text

Copyright © 2012, Pablo Impallari (http://www.impallari.com/, impallari@gmail.com**), Rodrigo Fuenzalida (**http://www.rfuenzalida.com/, hello@rfuenzalida.com**), with Reserved Font Name "Libre Caslon".**

Copyright © 2014, Impallari Type (http://www.impallari.com/**), with Reserved Font Name “Libre Caslon Text”.**

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: http://scripts.sil.org/OFL

[SIL OFL 1.1 license text as above in section “Liberation Fonts”]

Roboto

Copyright © 2011 Google Inc. All Rights Reserved.

This Font Software is licensed under the Apache License, Version 2.0. This license is copied above, and is also available at: http://www.apache.org/licenses/LICENSE-2.0

```
                    Apache License
              Version 2.0, January 2004
           http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions. "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document. "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License. "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity,

whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity. "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License. "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files. "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types. "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below). "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof. "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution." "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory

patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions: (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and (b) You must cause any modified files to carry prominent notices stating that You changed the files; and (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or

redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

General-Purpose Visual Programming Language

Pipe

Large language models are increasingly capable to generate high-quality software code. This poses a question about the role and importance of software developers in near future.

Despite all progress, AI is still not capable to fully comprehend complexity of business domain and technical constraints dictated by a set of specific circumstances. It means AI needs detailed and exhaustive specifications to be submitted for successful code generation. Preparation of such specifications is a difficult task, especially for complex systems. It creates an opportunity for software developers to move from the role of code creators to positions of higher-level designers and system architects by creating software design specifications that will be used as instructions for AI code generation. However, even high-quality and very detailed design does not guarantee AI code generation equivalent to the specifications, because no design can provide comprehensive and unambiguous description of required functionality. The gap between requirements and generated code widens as design complexity grows. As a result, the only way to successfully leverage power of AI for software construction is generating code only for base-level components easily explainable to AI, completing the rest of application via manual coding. However, getting back to manual programming undermines the goal of using AI to eliminate the need for human coding. This is where visual programming language capable to integrate non-visual code fragments into visual workflows is going to be especially useful. This book introduces a new visual programming language "Pipe", where visual blocks can encapsulate text-based code. Pipe allows developers to specify high-level software design via visual workflows, leaving low-level aspects of implementation to AI code generation tools.

As a next step of Pipe evolution, AI will be able to generate Pipe diagrams directly. Visual artefacts are much easier for humans to read and comprehend than text-based code and therefore, automatic Pipe diagram generation will allow more efficient control of an output produced by AI.

ISBN 978-1-0696277-0-4