ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
128

Lars Melander

Integrating Visual Data Flow Programming with Data Stream Management

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in 2446, ITC, Lägerhyddsvägen 2, Uppsala, Thursday, 6 October 2016 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Sharma Chakravarthy (University of Texas).

Abstract

Melander, L. 2016. Integrating Visual Data Flow Programming with Data Stream Management. *Uppsala Dissertations from the Faculty of Science and Technology* 128. 122 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-506-2583-7.

Data stream management and data flow programming have many things in common. In both cases one wants to transfer possibly infinite sequences of data items from one place to another, while performing transformations to the data. This Thesis focuses on the integration of a visual programming language with a data stream management system (DSMS) to support the construction, configuration, and visualization of data stream applications. In the approach, analyses of data streams are expressed as continuous queries (CQs) that emit data in real-time. The LabVIEW visual programming platform has been adapted to support easy specification of continuous visualization of CQ results. LabVIEW has been integrated with the DSMS SVALI through a stream-oriented client-server API. Query programming is declarative, and it is desirable to make the stream visualization declarative as well, in order to raise the abstraction level and make programming more intuitive. This has been achieved by adding a set of visual data flow components (VDFCs) to LabVIEW, based on the LabVIEW actor framework. With actor-based data flows, visualization of data stream output becomes more manageable, avoiding the procedural control structures used in conventional LabVIEW programming while still utilizing the comprehensive, built-in LabVIEW visualization tools.

The VDFCs are part of the Visual Data stream Monitor (VisDM), which is a client-server based platform for handling real-time data stream applications and visualizing stream output. VDFCs are based on a data flow framework that is constructed from the actor framework, and are divided into producers, operators, consumers, and controls. They allow a user to set up the interface environment, customize the visualization, and convert the streaming data to a format suitable for visualization.

Furthermore, it is shown how LabVIEW can be used to graphically define interfaces to data streams and dynamically load them in SVALI through a general wrapper handler. As an illustration, an interface has been defined in LabVIEW for accessing data streams from a digital 3D antenna.

VisDM has successfully been tested in two real-world applications, one at Sandvik Coromant and one at the Ångström Laboratory, Uppsala University. For the first case, VisDM was deployed as a portable system to provide direct visualization of machining data streams. The data streams can differ in many ways as do the various visualization tasks. For the second case, data streams are homogenous, high-rate, and query operations are much more computation-demanding. For both applications, data is visualized in real-time, and VisDM is capable of sufficiently high update frequencies for processing and visualizing the streaming data without obstructions.

The uniqueness of VisDM is the combination of a powerful and versatile DSMS with visually programmed and completely customizable visualization, while maintaining the complete extensibility of both.

Keywords: data stream management; data stream visualization; visual data flow programming; LabVIEW

Lars Melander, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden. Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1104-2516
ISBN 978-91-506-2583-7
urn:nbn:se:uu:diva-286536 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-286536)

Contents

Acknowledgements

Thank you,
Tore, for giving me the chance, and for the things I have learned,
Kjell, for the guidance and the support,
to my colleagues, for the good times and the bad.

This project is supported by eSSENCE; the Swedish Foundation for Strategic Research, grant RIT08-0041; and EU FP7 project Smart Vortex.

To Liz

As I walk and leave a trail
upon the sands of time,
your prints match mine without fail
and with a scent of lime.

When I faltered you were close
and helped me find my way,
always guiding me at those
times I go astray.

Singular, the luck one has
considering how your
patience is as boundless as
the seas that I explore.

For our matrimonial bliss
there's one thing left to do.
Kneeling, I am asking this:
Please, let me marry you.

Summary in Swedish

I denna avhandling presenteras en plattform för visuell dataflödesprogrammering och visualisering av dataströmmar, kallad *VisDM* (Visual Data stream Monitor). Dess syfte är att låta en användare enkelt och effektivt kunna hantera och visualisera dataströmmar.

Möjligheten att effektivt kunna hantera dataströmmar i industriella miljöer är numera kritiskt för att kunna utveckla tillverkningsindustrin. Åtskilliga internationella forsknings- och utvecklingsprojekt, såsom *Industrial Internet* [26], *Industry 4.0* [14][43] och *Made in China 2025* [40], har som mål att höja produktiviteten och kvaliteten för industriella tillverkningsprocesser och produkter. Ett mycket viktigt område som belysts i EU:s Smart Vortex-projekt [72] är förmågan att skalbart kunna samla in, behandla, analysera och visualisera dataströmmar.

Industriella system skapar väldiga mängder sensordata i form av kontinuerliga realtids-dataströmmar från industriella processer och produkter utrustade med sensorer. En dataström kan bestå av mätningar från en enda sensor, med värden uppmätta för en enstaka komponent, eller bestå av en sammanställning av flera dataströmmar. En resultatström kan vara en enkel filtrering eller aggregering, eller en tillämpning av komplexa statistiska analyser, komplexa modeller, vibrationsanalyser, etc. Datakällor kan ligga både på komponentnivå och systemnivå. Till exempel kan industriell utrustning ha en uppsättning av sensorer installerade, vilka fortlöpande mäter utrustningens tillstånd. Ett kluster av dessa sensorer kan sedan användas för att mäta nötning, belastning, åverkan, mm. för enskilda komponenter. Aggregering över en uppsättning av dessa strömmar kan användas för att få en enhetlig översiktsbild av en hel produktionsenhet.

Allt eftersom dataströmshantering blir mer och mer omfattande och komplex så krävs metoder och lösningar som kan underlätta denna hantering och motverka den ökande komplexiteten. Lösningen som presenteras i denna avhandling tillhandahåller enkel analys och visualisering av dataströmmar genom visuell dataflödesprogrammering av industriella tillämpningar. Ett generellt dataströmshanteringssystem exekverar kontinuerliga frågor som har definierats av användaren. Dessa frågor kopplas upp mot dataströmmarna och kör analyser, filtreringar, transformationer, mm. Resultatet kan sedan enkelt visualiseras av användaren.

Generellt sett så är det önskvärt att flytta design- och programmeringsuppgifter så nära slutanvändaren som möjligt, genom att höja abstraktionsnivån och gömma komplexa moment genom automatisering. Detta kan åstadkommas genom att fokusera på dessa områden:

- Endast deklarativ programmering. Användare bör så långt det är möjligt endast behöva fokusera på *vad* de vill göra, inte *hur*. Programmeringsspråk på lägre nivå (C++, Java, etc.) är procedurella och fokuserar nästan uteslutande på hur ett program skall implementeras, och kräver oftast omfattande erfarenhet för att användas korrekt. De flesta användare kommer därför att finna dem alltför svåra att tillämpa. Databasfrågor, som SQL och liknande, är å andra sidan deklarativa och kräver inte att användaren har insikt i algoritmer eller andra detaljer för att kunna utföra sin uppgift.
- Undvika behovet av specialiserad programmering, genom att beskriva och hantera begrepp på en högre abstraktionsnivå och undvika implementationsspecifika lösningar.
- Applikationsorienterad visuell programmering. Att låta användare bygga sina program med symboliska byggstenar är mycket mer intuitivt än textbaserad programmering, och kan tilltala de som finner programmering främmande.

VisDM är ett klient-serversystem där klienten har konstruerats i det visuella programmeringsspråket LabVIEW och servern är baserad på dataströmshanteringssystemet SVALI. Det bygger på en uppsättning av *VDFC*-definitioner (Visual Data Flow Component), vilka är uppdelade i *producers*, *operators*, *consumers* och *controls*. De bygger upp de olika delarna av dataflöden som används för att hantera dataströmmar.

LabVIEW har ett *actor framework* som utgör grunden till VisDM-klienten. Ovanpå detta har ett *data flow framework* byggts som innehåller dataflödesabstraktioner, dynamisk typhantering, felhantering, visualiseringsstöd, m.m. Detta ramverk ligger sedan till grund för VDFC-definitionerna. VDFC:er används för att definiera och hantera strömkällorna medelst kontinuerliga frågor, samt koppla dem till korrekt visualisering. De används också för att hantera uppdateringsfrågor, vilka kan köras för att ändra serverns tillstånd närhelst användaren önskar. Vidare har SVALI utökats med ett ramverk för att dynamiskt kunna ladda och köra LabVIEW-instrument för att kunna inhämta externa dataströmmar genom visuell programmering.

VisDM har testats i två verkliga tillämpningar:

- Visualisering och validering av dataströmmar från industriella maskiner hos Sandvik Coromant.
- Signalbehandling och visualisering av radiodata inhämtat från en digital 3D-antenn som sköts av institutet för rymdfysik i Uppsala (IRFU).

I båda fallen visualiseras data i realtid. Avsikten är att VisDM skall erbjuda fullt stöd genom hela strömhanteringsprocessen, utan att tumma på vare sig prestanda eller användarvänlighet.

Centralt för industriella processer är översyn av dataströmmar och problemlösning, vilket är uppgifter som är starkt beroende av användarorienterad visualisering och lättillgänglig inmatning av parametrar.

Sandvik Coromant[1] utvecklar och tillverkar verktyg för metallbearbetning, och tillhandahåller en utförlig kunskapsbas om skärning av metall. De har ett världsomspännande nätverk av maskinparker för bland annat fräsning och borrning, och dessa maskiner är utrustade med sensorkluster vars utdata behöver behandlas och övervakas. Med moderna produktionsflöden blir traditionella övervakningsmetoder otillräckliga. Slitage och nedbrytningar behöver upptäckas så tidigt som möjligt i produktion, vilket är omständligt och kostsamt utan automatisering.

VisDM har använts för att definiera ett gränssnitt till en LOFAR (LOw Frequency ARray) antennprototyp som används av institutet för rymdfysik i Uppsala (IRFU) på Ångströmlaboratoriet[2]. Antennen är en sofistikerad, helt digital antenn som har tre ortogonala antennelement, vilket möjliggör mätningar av radiosignalers riktning och polarisering.

Unikt för VisDM är dess utbyggbarhet. Inget annat system är så anpassningsbart för att kunna hantera alla möjliga sorters lösningar för dataströmshantering.

1 http://sandvik.coromant.com

2 http://www.irfu.se

List of papers

The papers are referred to in the text by their Roman numerals:

I Lars Melander, Kjell Orsborn, Tore Risch, Daniel Wedlund
Visualization of Continuous Queries using a Visual Data Flow Programming Language
[Submitted for journal publication]

I am the primary author of this paper.

II S. Badiozamany, L. Melander, T. Truong, C. Xu, T. Risch
Grand challenge: implementation by frequently emitting parallel windows and user-defined aggregate functions
Proceedings of the 7th ACM international conference on Distributed event-based systems, 2013, pp 325–330

Authors are listed in alphabetic order. My contributions:

- Implemented the "Shot on Goal" query, and its inclusion in the main solution.
- Wrote 11% of the text in the paper.
- Responsible for testing the solution.
- Demo visualization.

III M. Leva, M. Mecella, A. Russo, T. Catarci, S. Bergamaschi, A. Malagoli, L. Melander, T. Risch, C. Xu
Visually Querying and Accessing Data Streams in Industrial Engineering Applications
21st Italian Symposium on Advanced Database Systems, SEBD 2013
Roccella Jonica, Italy, June 30th–July 3rd, 2013

I provided text input, and the DSMS server functionality and API. ■

1 Introduction

Sir Lancelot: "Look, my liege!"
King Arthur: "Camelot!"
Sir Galahad: "Camelot!"
Sir Lancelot: "Camelot!"
Patsy: "It's only a model."
King Arthur: "Shh!"

—Terry Gilliam et al., *Monty Python and the Holy Grail*

The capability to efficiently handling data streams in industrial processes is becoming critical for transforming the current manufacturing industry. Several major international research and development initiatives such as *Industrial Internet* [26], *Industry 4.0* [14][43], and *Made in China 2025* [40], are focussing on this transformation of the current manufacturing industry with the overall goal of improving productivity and quality of industrial processes and products. A critical area within this context, addressed in the EU project *Smart Vortex* [72], is scalable capability to collect, process, analyse, and visualize data streams to support cyber-physical systems [43] as found in industrial processes and products, in the project exemplified by machining processes, hydraulic power systems, and heavy vehicles in production.

In an industrial system, large volumes of sensor data are produced in the form of continuous data streams from industrial processes and products equipped with sensor installations. A data stream can be generated by a single sensor, measuring some quantity at the component level, or it can be a derived stream that constitutes aggregated values over one or several other streams. A derived data stream can be based on some simple filtering or aggregation operation but can also involve the application of much more complex analytical and empirical models, such as statistical analysis, on-line clustering algorithms, vibration analyses, etc. The data streams can further originate from all levels of an industrial system, from the component level to the system level. For example, industrial equipment will be equipped with collections of sensors that will generate data streams providing data about the current condition of a machining process. A set of sensors can then be used for measuring wear, stress, strain, etc. for the individual components of the equipment in use. Aggregations over collections of streams can also be applied to derive more general states and conditions by selecting various sets and compositions of streams or from equipment used in production lines. To make the output data streams intelligible by an analyst, they should be visualized in real-time.

As data stream management is becoming increasingly complex, we need methods that counter-balance the complexity and make it more accessible. The approach in this Thesis enables easy analysis and visualization of streaming data. The proposal presented is a flexible visual specification and deployment of visualizations of data stream analyses produced by a data stream management system (DSMS) [30].

A DSMS is similar to a database management system (DBMS) with the difference that while a DBMS allows querying only stored data using a declarative query language like SQL, a DSMS in addition provides *continuous queries* (CQs) to query data streams in real-time. The CQs can filter, transform, combine, and distribute the accessed data streams. A CQ differs from its DBMS counterpart in that it may not have a determinate endpoint; it runs until the data streams feeding it are terminated or its operation is interrupted by the user or the system. CQs are very responsive, immediately returning results as soon as they are available, unlike a batch query that returns results only when it has finished running.

In general terms, it is desirable to move design and programming tasks closer to the end user, by raising the level of abstraction and hiding more complex tasks through automation. There are some direct ways of accomplishing this:

- Making application programming declarative. Users should as far as possible be able to state *what* they want done, without having to state *how* to do it. Low-level programming languages (e.g. C++ or Java) are procedural, meaning that programming is almost exclusively about how things are done, and they require extensive programming training and experience. Non-expert programmers may consequently find them too difficult to use. By contrast, database query languages such as SQL are declarative, where users do not specify the algorithms to be used and other details when performing database searches.
- Avoiding the need for programming specialists. A common approach to data stream processing is to build systems from the ground up, using libraries in a conventional programming language [4][5][6][31]. The major drawback is that such implementations rely heavily on the expertise of the development team involved, which is becoming increasingly rare as programmer demand and application complexity increases. An alternative is to use declarative CQs to enable very high level specification of data stream processing, without having to explicitly specify details.
- Introducing application oriented visual programming. Letting users build programs by manipulating graphical building blocks is much more intuitive than textual programming, and may appeal to those who find programming awkward and difficult.

1.1 Research questions and proposed solution

Looking at particularly industrial machining operations, it becomes clear just how much data streaming applications can vary even within the same operational context. Collecting all issues, certain research questions stand out:

- Can application usability be increased without hurting efficiency? Tasks should become easier to implement, in a shorter time span, and requiring fewer resources, while at the same time getting the same results as or better than existing systems.
- How does high-rate stream throughput from multiple sources affect design decisions? Scalability is a keyword in the database world, and visualization should not impose constraints.
- How can sophisticated visualization accommodate both ease of use and extensible customization?
- To what extent can programming become more user-centric? A data stream management system may be used by a dedicated program developer, an engineer, or an operator, roles which may or may not belong to the same person. Regardless of the role, a person should be comfortable using the software.

Visual data flow programming [20][22] offers rapid and robust prototyping of applications. Data stream management is conceptually similar to data flow programming, and with data flows the step between specification and implementation is eliminated; the program specification becomes the program. Development time decreases, and programming tasks can be moved closer towards the end user.

LabVIEW [67] from National Instruments[1] is a widely used visual programming platform for building solutions to all sorts of industrial and scientific signal processing applications [84][71][62]. It is often cited as a "de facto standard" for developing testing and simulation solutions for signal processing, e.g. to generate and visualize data streams [46][13].

The approach presented in this Thesis, *Visual Data stream Monitor* (VisDM), addresses the above issues by utilizing the existing state-of-the-art visual programming environment in LabVIEW to enable high-level visualization for engineering and scientific DSMS applications. LabVIEW offers a visual programming environment that is comprehensive, yet has a flat learning curve, and a user interface that many find attractive [25][91][9]. It supports object-oriented programming and has an interface for calling external functions. Like most programming tools of its kind, LabVIEW supports the building of stand-alone programs that can be deployed without depending on the development environment.

1 http://ni.com/labview

It is shown how *visual data flows* enable declarative specification of application programs visualizing data streams defined as CQs to a DSMS, specifically how producer-consumer pairs are created to link a CQ to its appropriate visualization. A visual data flow is a program specified using graphic building blocks called *function nodes* [22][81] where each node consumes one or several input data flows and produces output data flows or visualizations. The function nodes are implicitly driven by the flow of data, rather than by explicit control structures as in regular programming.

The prototype system provides an integrated visualization and scalable data stream analysis platform, by interfacing LabVIEW with the *SVALI* (Stream VALIdator) data stream management system [93]. SVALI is fully extensible and includes several ascending technologies, such as distributed stream processing, stream windowing, and customized indexing. SVALI has been tested and scrutinized in several real-world industrial applications [Paper II][10][93], and has proven itself to be a robust and flexible DSMS. It is the fundamental building block for the solutions presented in this Thesis, and has been thoroughly tested in the *Smart Vortex*[1] project [72]. SVALI scales very well with the work load, as it can dynamically start parallel stream query processes when needed.

1.2 Contributions

LabVIEW has been extended with a toolbox, *Visual Data Flow Components* (VDFCs), which enable declarative visual specification of data stream applications as visual data flows. The declarative, data flow centric programming with VDFCs does not rely on control structures the way regular programs do. The set of VDFCs is extensible, so that adding new components when needed is easy.

The integration of LabVIEW and SVALI has made it possible to develop a mechanism for users to visually define *data stream wrappers* on a high level in LabVIEW. A data stream wrapper is a program module to handle communication between SVALI and external stream sources. Visual data stream wrappers enable entire applications to be defined in VisDM, only using CQs combined with visual data flow specifications.

In VisDM the visualization is specified by connecting CQs to function nodes in LabVIEW that continuously visualize consumed stream elements. The sources of the visualized data flows are function nodes connected to CQs through a stream-oriented client-server API. The function nodes are based on LabVIEW's *actor framework* [56]. Actors are stand-alone, thread-based processes that communicate between each other using messages [1][33]. It is fairly straightforward to

1 http://smartvortex.eu

design a data flow environment using actors; each actor becomes a function node, and each entity in a data flow becomes a message that is sent from one actor to another. By using the actor framework to define function nodes in VisDM, the procedural control structures used in conventional LabVIEW programming are eliminated.

VDFCs are constructed using a *data flow framework* that has been developed for VisDM, based on the actor framework. It contains visualization components, dynamic tuple [24] handling, error handling, etc.

There are typically many CQs running concurrently, and there may be *update* queries running occasionally. Each query needs exclusive access to the SVALI system when running, and to accommodate this a *multiplexing server structure* is introduced. Implementing the new server structure requires *cooperative multi-threading primitives*, which are required for making query operations responsive, both for the server structure and general query processing. However, queries cannot be interrupted in the classic non-preemptive manner of most operating systems. Instead, queries relinquish control at certain points of their execution, allowing other processes to execute. Typically, a query will wait for some time for new data to arrive on a stream. While it waits, the query is moved to a processing queue, letting another query process operate in the meantime.

1.3 Terminology

Application programming interface, API Provides functionality for accessing a software component. It defines a set of routines for input, output, types, etc., creating logical independence between the base system and its calling conventions, and the component being accessed.

Asynchronous VI A subVI that runs independently of all other VIs. It is not managed by the run-time environment and error handling is generally very limited.

Background execution When a coroutine has yielded operation, but continues to execute. It cannot make changes to the system environment.

Block diagram Contains the code of a LabVIEW program. Programs are displayed graphically on a two-dimensional canvas and execution is done from left to right.

Class A program template, from which objects can be instantiated.

Continuous query, CQ A query that does not have a determinate end point. It outputs derived stream elements in real-time, immediately after initiation. It typically has a window function, looking at a part of the stream at a time and performing operations over that part.

Daemon A process that runs hidden from users, usually performing an automated service.

Database management system, DBMS Software that provides efficient storage and management of data. There are many types, the most well-known being relational DBMSs.

Data-driven execution Program execution is dictated by the flow of data. As soon as a program component has sufficient data for execution, it will do so.

Data stream management system, DSMS Software that provides efficient handling of data streams.

Demand-driven execution Program execution is dictated by data requests. Program components will only execute when subsequent components want data.

Derived stream A filtered data stream, or output from a stream operator. In the context of a data stream management system, it is typically the output from continuous query.

Dispatcher A process that polls the states of a set of processes and executes them in order according to a set of rules.

Dynamic link library, DLL See shared object.

Dynamic typing A variable's type is set at run-time when assigned a value, and can change several times during execution.

First class object An entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights that other variables in the programming language have.

Foreground execution The state of a coroutine that executes while maintaining ownership of the system environment.

Front panel The interface for a LabVIEW program, displaying input boxes, diagrams, etc.

Function node An entity that first waits for data to arrive on all of its inputs. Once data has arrived, the node is said to fire; it executes its function, typically ending with data being transmitted on one or more output wires. Execution is compartmentalized; function nodes do not interact with or change the state of the system in which they operate. Their own state may change internally during firings.

Impedance mismatch Appears when an entity or concept from one system cannot readily be translated to another system. The most common example is object-relational mismatch, where objects in a programming language do not have a corresponding entity in a database, and the relations in the database likewise do not have a corresponding concept in the programming language.

Internet protocol, IP The base protocol for transmitting data packets over Internet.

Method A function that belongs to a class.

Multitasking Running more than one process at the same time in the same operating system. This can be done by utilizing parallel processing pipes, or by switching between processes. Of the latter, the most common type is preemptive multitasking, where the operating system interrupts executing processes, as opposed to non-preemptive (cooperative) multitasking, where processes relinquish execution on their own accord.

Overloading Defining several functions with the same name. They are discerned by the type and number of parameters.

Polymorphism Using a single interface or calling convention for entities of different types. It is useful for preserving unique behaviour of objects in a collection.

Port An endpoint of communication in an operating system. Identifies a specific process or a service.

Preallocated clone reentrant execution By default there will only be a single instance of a VI residing in a LabVIEW process. All calls to the VI will go to that instance. The calls cannot be concurrent, and the local state of the VI will be shared among the calls. This mode instead causes a separate instance (clone) to be allocated for each separate call to the VI. This preserves the local state of the VI for that particular call, and is independent of other calls to the VI.

Race condition When events must occur in a certain order, but the supporting system fails to uphold that order. May appear when separate processes share resources, and is usually caused by program bugs or a lack of proper synchronization.

Run-time engine Provides an environment for running programs that would otherwise not be executable on a certain system.

Secure sockets layer, SSL A cryptographic protocol for providing secure communication over a computer network.

Shared object A program module that can be loaded by another program at run-time. It is useful for inserting new functionality into an existing system.

Single assignment The idea of increasing program stability by allowing variables to be assigned values only once during their lifetime. All data flow programming languages uphold this rule.

Static typing Variable types are resolved before running a program, and cannot change.

Structured query language, SQL A programming language designed for managing data in a relational DBMS or DSMS.

SubVI A user-defined function in LabVIEW, i.e. a VI that is used inside another VI.

Tuple A tuple is a collection of ordered data. It can be handled as a single entity, but the contained elements can also be accessed individually.

User datagram protocol, UDP A simple protocol for transmitting data packets. It is useful for data streaming, but unreliable.

Virtual instrument, VI A program written in LabVIEW.

Wrapper In general terms, a function or set of functions that is/are used for accessing another function or set thereof. In data streaming terms, a function used for accessing an external data stream source.

XControl A front panel object that encapsulates other front panel objects. Provides functionality for handling different kinds of events, and allows programmers to include various automation for the encapsulated objects.

Diagram arrows

There are several diagrams in this Thesis, with arrows of different shapes and colours. Each arrow type has a certain meaning:

→ A blue arrow indicates a data transfer of a single entity at a single instance.

⇢ A dashed blue arrow indicates a change of state, initiated by an operation that is not part of the affected process.

⇒ A double lined blue arrow indicates a data flow or data stream. Entities are transferred continuously until operation is halted or the source runs out of entities to transfer.

→ A grey arrow indicates an execution flow, where the process maintains ownership of a system.

⇢ A dashed grey arrow indicates an execution flow, where the process does not have system ownership, and thus must not access the components of the system, or must do so with caution.

⋯ A dotted grey line indicates a halted process. The process will sleep until it is signalled.

—▷ A black arrow with white-filled tip indicates class inheritance, pointing to the parent class in a hierarchy. ■

2 Monitoring industrial machines

'Cheshire Puss, would you tell me, please,
which way I ought to go from here?'
'That depends a good deal on where you
want to get to,' said the Cat.
'I don't much care where—' said Alice.
'Then it doesn't matter which way you go,' said the Cat.
'—so long as I get *somewhere*,' Alice added as an explanation.
'Oh, you're sure to do that,' said the Cat,
'if you only walk long enough.'

—Lewis Carroll, *Alice's adventures in wonderland*

The flowchart in Figure 1 shows a generic overview of a data stream management system with visualization that can be applied to various data streaming applications. As the backend and frontend have very different computational properties, it makes sense to divide them into a server and a client part. The client can be kept on a portable device, while the server handles the resource-intensive computations on a stationary machine or cluster.

Figure 2 shows a very simple data flow schematic. As data arrives to a function node [81], it is processed locally. Adding visualization of output to a data flow is much easier than is usually the case with other programming platforms. It is just the matter of adding the visualization where it is desired, often at the end of a data flow, but also in the middle if one wishes, as with Figure 3, where the data flow has two display nodes added, one in the middle of a program and one at the end.

Figure 4 illustrates how equipment is monitored with VisDM. *Instrumented industrial machines* produce machine data streams from sensors [26], which are continuously processed in real-time by VisDM. VisDM includes a *stream visualizer* where engineers can observe derived data streams produced by a *continuous data stream analyser* that analyses data in the machine data streams. When anomalies are detected the operator will perform *feedback actions* that alter the behaviour of the machines. With a conventional batch data mining approach the turnover rate can be counted in hours, days, or even longer, which is far too slow for many industrial processes, especially manufacturing, where process degrada-

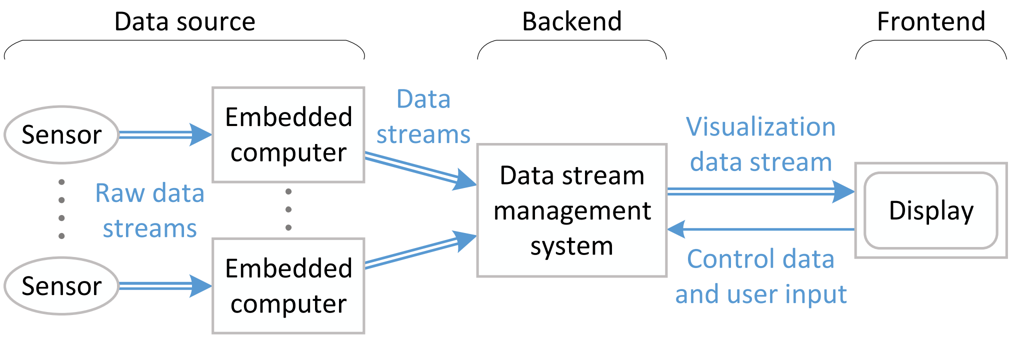


Figure 1: A DSMS-based data collection and visualization system.

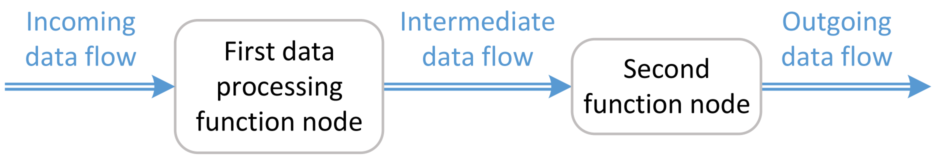


Figure 2: A simple data flow example.

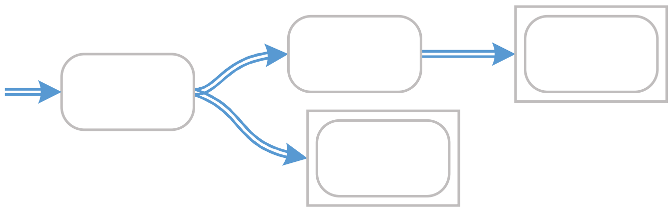


Figure 3: A data flow program with two display function nodes added.

tion can come very quickly. If data can be processed immediately in real-time, without intermediate storage, the feedback time is only measured by the reaction time of the operator. The information delay from a machine to a supervisor is only determined by the latency of the system, allowing an engineer (or a system) to react to changing circumstances in very short order. The continuous data stream analyser also supports immediate feedback without human involvement. If automated feedback is used, the response time can be counted in milliseconds.

In VisDM the continuous data stream analyser processes CQs over a general model of the monitored equipment in terms of a local main-memory database inside the system [93][72]. The model consists of a set of functions and types that define the database schema as well as derived quantities. Functions defined in the

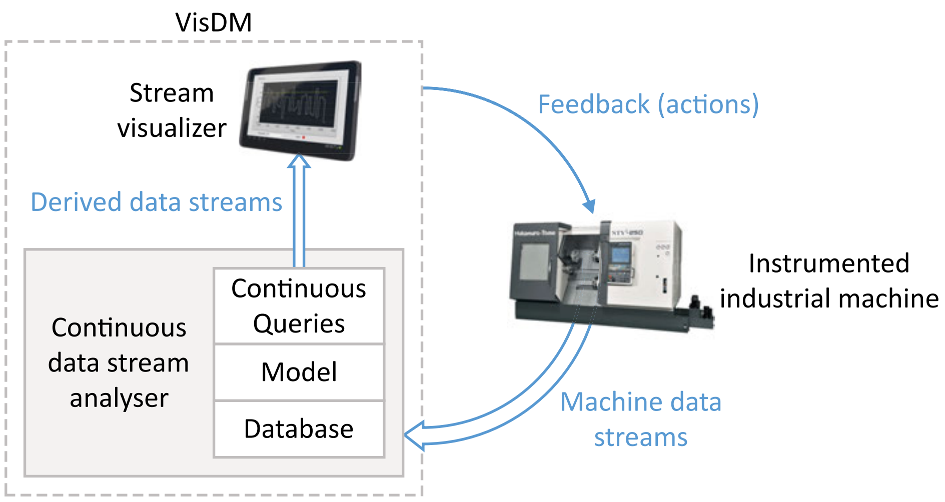


Figure 4: Streaming data feedback. Data is processed as it is being visualized.

model may return streams that combine data stored in the database with on-line data from the machine data streams, e.g. continuously identifying or predicting deviations from normal machine behaviour based on the model.

Figure 5 shows a simple example of how a data stream visualization may look to an end user, with the corresponding specification in Figure 6. The specification is minimal in that it contains only the parts needed to specify the visualization, and nothing else. Every component that is used to build the infrastructure for the specification is available for customization, but hidden from the user.

2.1 Showcases

VisDM has been used in one industrial case and one academic case: Validating machine operation for Sandvik Coromant, and measuring radio signals in the LOFAR project. It is intended to offer comprehensive functionality throughout the entire stream handling process, while maintaining both powerful, scalable stream processing and ease of use.

Central to industrial cases are data stream monitoring and problem solving, issues that rely on user-oriented data presentation and responsive user input. For industrial cases, projects can be divided into three distinct parts:

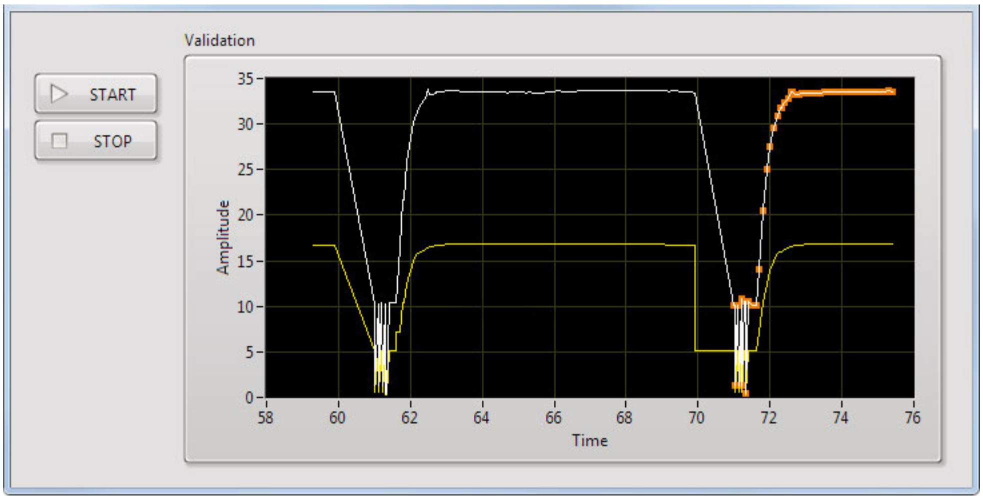
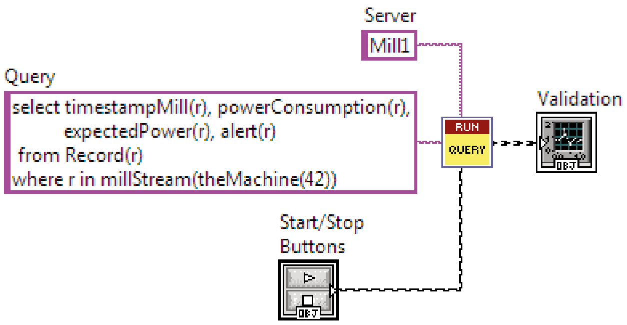


Figure 5: A simple data stream visualization application.



```
select timestampMill(r), powerConsumption(r),
        expectedPower(r), alert(r)
 from Record(r)
where r in millStream(theMachine(42))
```

Figure 6: Visual data flow specification for the application.

1) **Model design**. This consists of queries that process incoming data, functions that define the operational model of equipment, and schemas for data and local storage. Designing and programming a model is non-trivial and requires a certain amount of domain knowledge. On the other hand, this needs to be done only once for each type of machine.

This Thesis touches only very briefly on this part, as it is not within the focus of topics presented in the Thesis.

2) **Operational design**. This is the part which benefits the most from visual data flow programming. Remote machines, on-board and off-board computers, their interaction, and the operation of each is programmed using drag-and-drop

symbolic function nodes wired together with virtual cables. Stream data is managed and processed by calling stream functions in the model from the function nodes.

Data flow programming substantially reduces the amount of possible errors, simply by eliminating the procedural programming mode. Function node operation is localized, and when changes that are global for the process are made to the data stream management system, they become regulated simply by the nature of the underlying database system and its application programming interface.

A user may not need to design or maintain the application program, but doing so becomes easy and intuitive. The risk of introducing errors because of inexperience is minimized.

3) **Visualization**. This is where a user will spend most of their time, actually running a system. As shown, visualization nodes become part of the application design, meaning that both the customization of visualization and the operational behaviour thereof become transparent to the user, to the same extent as for any other function in the solution.

Since each application may demand its own type of visualization, and the platform is supposed to accommodate for future implementations, it means that not only does existing visualization elements need to support full customization, but the design of new elements must be as easy and forthcoming as possible. This is not possible with any function library, however user-friendly it may appear, if the underlying solution does not resolve the issues mentioned.

The Sandvik case will be used throughout the Thesis for examples.

Sandvik Coromant – remote machine process monitoring

Sandvik Coromant[1] develop and manufacture tools for the metalworking industry, and also build extensive knowledge in the field of metal cutting. This combination is provided as a package to Sandvik Coromant customers. In production and testing facilities around the world, Sandvik Coromant has a collection of various machine tools where some of them are performing milling and drilling tasks for which various sensors and derived data via formulas and models need to be monitored [76]. While performing these tasks, monitoring is crucial, yet traditional point-wise comparison does not always solve the monitoring task. Faults in the process need to be caught at the earliest possible juncture. This is tedious and costly without automation.

In this scenario, each machine tool is equipped with a set of sensors, measuring various properties, which include rotation speed, power consumption, movement, and torque, counting from 15 parameters in total and upwards. The

1 http://sandvik.coromant.com

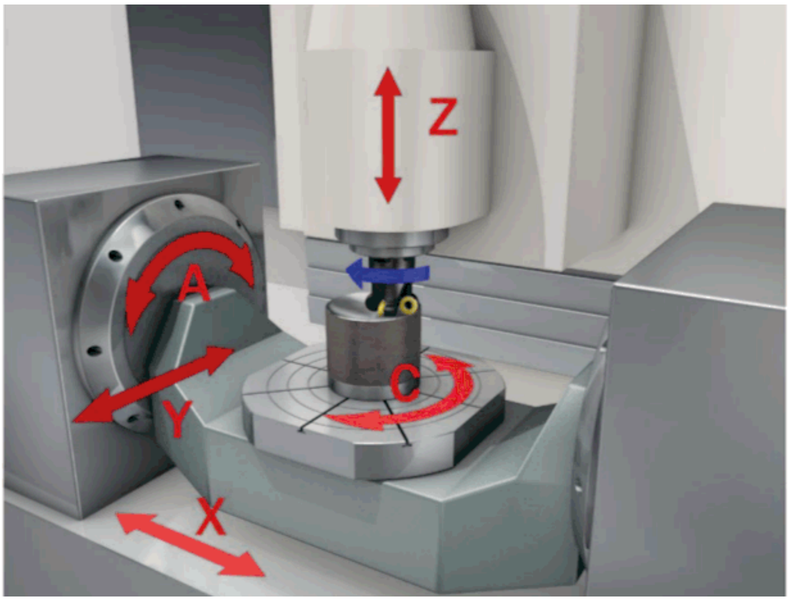


Figure 7: A milling machine.

behaviour of each property can be learned using a statistical model trained by measurements of a healthy machine during a learning phase. Milling machines are very manoeuvrable, and Figure 7 shows how milling can be manipulated in the X, Y, and Z axes, and rotated around the X and Z axes. The actual machine tools used in the use case are multi-operational machine tools, but for exploring the ability to inherit stream parameters one machine tool has been denoted as a milling machine and another as a drilling machine. The benefit of having inheritance of the stream structure is the ability to configure any sensor configuration at any machine from the simplest drill press which have one axis and one spindle to complex grinding machines and as in our case multi-operation machine tools.

A CQ is a query returning a stream of objects and is defined in terms of stream valued functions. A simple example is the following CQ which returns a stream of tuples that represent the power consumption over time of the milling machine in Figure 7:

```
select timestampMill(r), powerConsumption(r)
  from Record r
 where r in millStream(theMachine(42));
```

The function *theMachine()* accesses the database to return the object representing the machine labelled "42". The derived stream valued function *millStream()* encapsulates the interface for a given milling machine and produces a stream of JSON records [51] *r* containing time stamped sensor values. The query returns

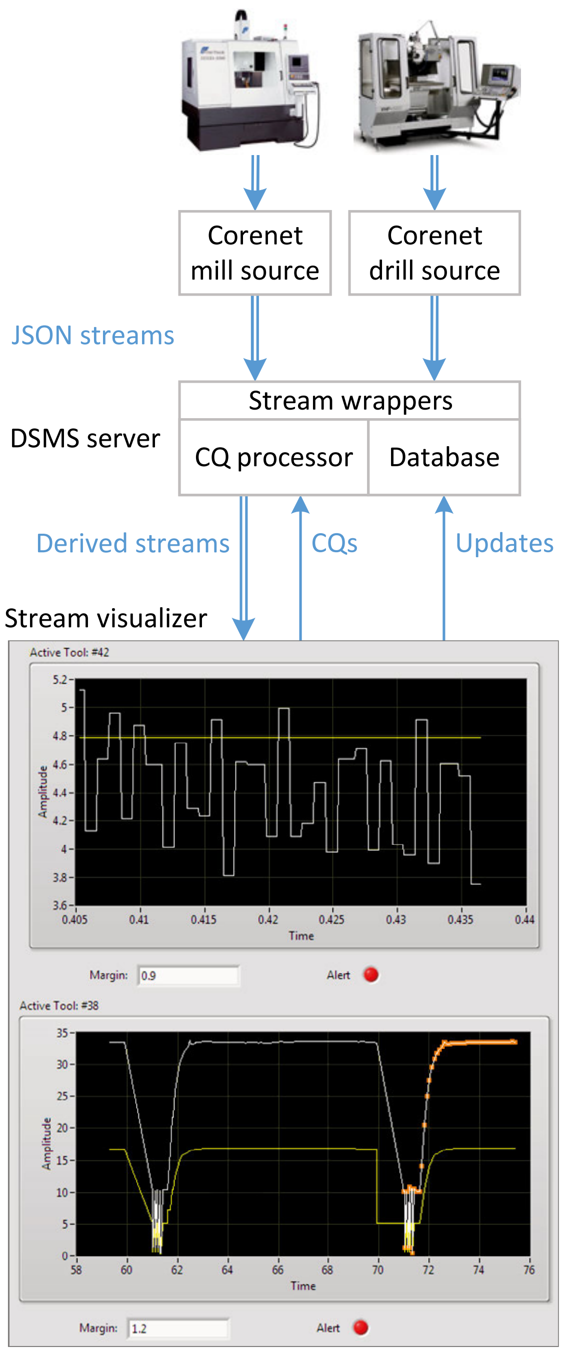


Figure 8: Machining equipment monitoring. The yellow line marks a threshold that dictates the correct operation.

a stream of time stamps and power consumptions extracted from these records. The functions *timestampMill()* and *powerConsumption()* extract attributes from each JSON record.

Figure 8 illustrates how VisDM is used for monitoring two machine data streams, one from a milling machine and one from a drilling machine. The raw data output is collected and structured by a software package called *Corenet* (Coromant Extended Network) that contains a device gateway and a factory gateway. The device gateway is the interface to a generic machine tool and its sensors, and exposes a well formulated data stream. The factory gateway exposes all connected device gateways over a single channel upstream to enable connectivity without exposing each individual machine tool which would result in a much larger attack surface. The Corenet server broadcasts JSON streams over an SSL-encrypted connection.

The servers can connect to the Internet and thus the monitored machines can be located anywhere where there is an Internet connection. The JSON streams are interfaced with VisDM through a *stream wrapper*, which is a plug-in to SVALI that iteratively converts the received measurements into the format used by SVALI. Metadata and models about the monitored machines are stored in a main-memory local *database* inside SVALI. The tuples in the derived streams produced by the *CQs* are continuously emitted to the visualizer. Furthermore, model data referenced in monitored CQs can be dynamically updated at run time to alter the visualization. For example, a CQ definition may depend on a user-provided threshold stored in the local database and when the threshold is updated the CQ visualization changes.

The output diagrams in Figure 8 both continuously plot a stream of power consumption data measurements while comparing them to desired power consumption, indicated by yellow lines. The desired power consumption is defined by the model. Alerts are signalled to notify the operator if the measured power consumption deviates more than a user-specified margin from the desired power consumption. In the top diagram, the model is a mathematical formula based on machine specifications, while in the bottom diagram a statistical model is trained by measuring the behaviour of a healthy machine. In both cases the margin can be changed by the user, which will update the local database and influence the alert sensitivity.

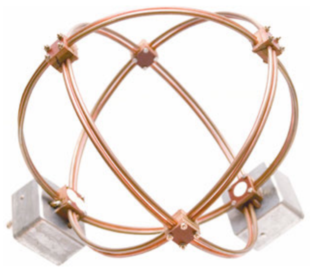


Figure 9: The 3D antenna prototype.

LOFAR digital antenna

VisDM has been used to define an interface to a LOFAR [32][89] (LOw Frequency ARray) antenna prototype [49] that is operated by the Swedish Institute of Space Physics in Uppsala (IRFU)[1] at the Ångström Laboratory. The antenna (Figure 9) is a sophisticated, completely digital antenna, and has three orthogonal antenna elements, allowing an operator to measure not just the radio signal strength, but also things like direction and polarization, and thus allowing for advanced radio data handling and visualization. LOFAR is a *synthesis array* [55] and is used for astronomical observations.

LOFAR consists of about 20 000 antenna units operating in tandem. Combining all signals through very processor-intensive calculations, the antennas operate as one very large radio telescope. This setup produces vast amounts of data, which has to be processed immediately as it is collected.

High band antennas (Figure 10) are collected in arrays (Figure 11), which are then clustered (Figure 12) throughout Europe (Figure 13), mostly in the Netherlands. Unlike the prototype, they only have two antenna elements, leaving out the Z axis. These antennas have a bandwidth of 50 MHz whereas the prototype has a more moderate bandwidth, running at less than 100 kHz.

Shown in Figure 14, the DSMS server invokes a *wrapper handler* for running a *visual stream wrapper*. The stream wrapper was defined in LabVIEW and then dynamically loaded in SVALI using VisDM's wrapper handler framework. The *antenna controller* sends a stream of UDP packages to the stream wrapper. The package data is forwarded to the wrapper handler which converts them to SVALI types. The *CQ* then applies *signal transformations* to the data. ■

1 http://www.irfu.se

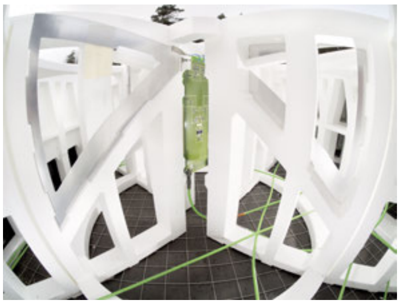


Figure 10: A high band antenna. © Nout Steenkamp.

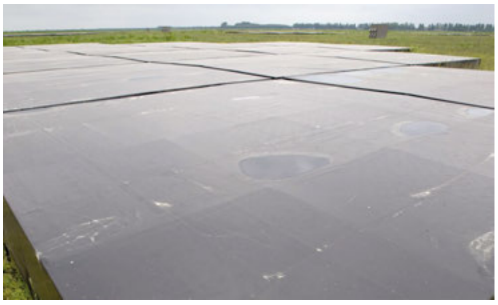


Figure 11: Black casing covering antennae. © Hans Hordijk.

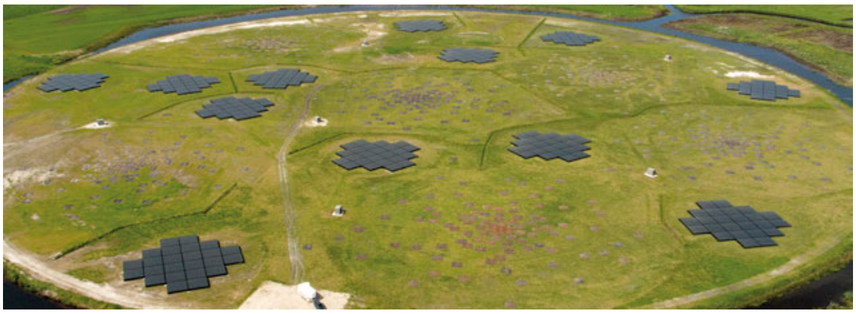


Figure 12: The "superterp" on which six LOFAR stations are housed. © Top-Foto, Assen.

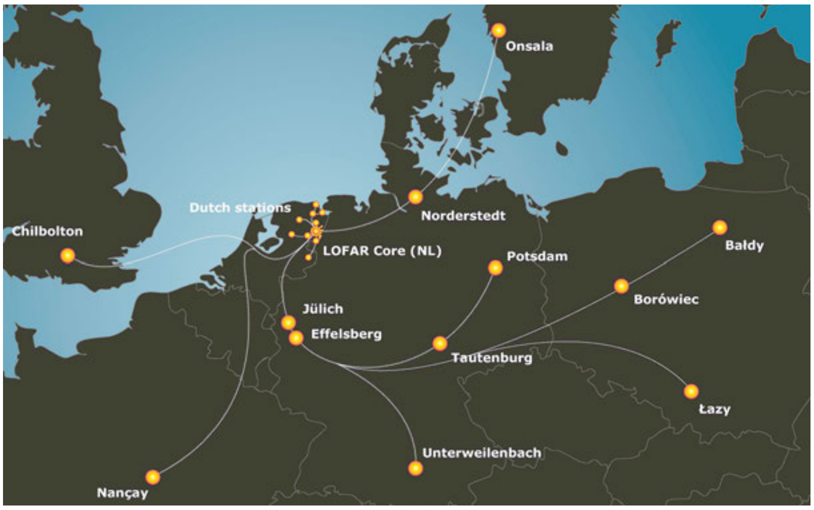


Figure 13: The international LOFAR telescope. © ASTRON.

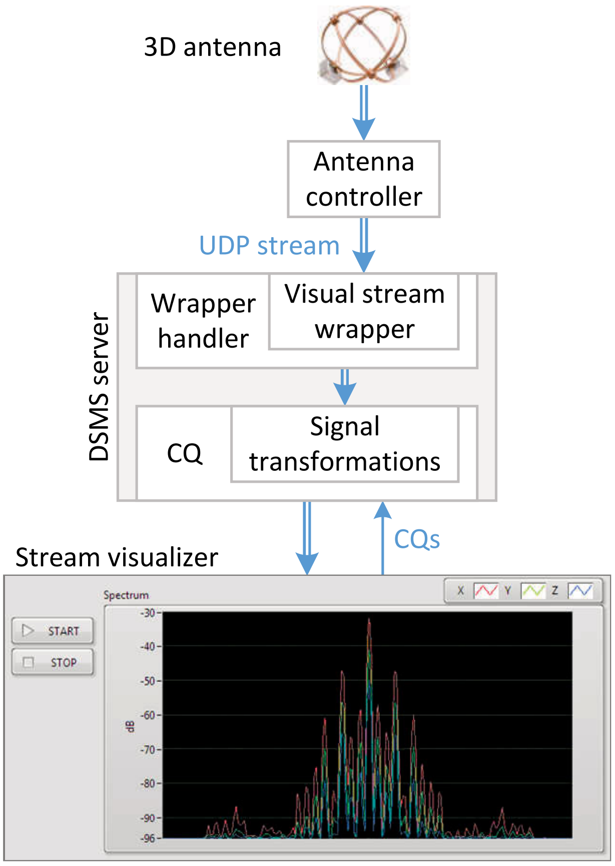


Figure 14: Visualization of radio data.

3 Background

Cat [to Rimmer]: "What is it?"
Rimmer: "It's a rent in the space-time continuum."
Cat [to Lister]: "What is it?"
Lister: "The stasis room freezes time, you know, makes time stand still. So whenever you have a leak, it must preserve whatever it's leaked into, and it's leaked into this room."
Cat [to Rimmer]: "What is it?"
Rimmer: "It's a singularity, a point in the universe where the normal laws of space and time don't apply."
Cat [to Lister]: "What is it?"
Lister: "It's a hole into the past."
Cat: "Oh, a magic door! Well, why didn't you say?"

—Rob Grant & Doug Naylor, *Red Dwarf: Stasis Leak*

Visualization functionality comes with trade-offs. We want it to be applicable for whatever task we may think of without being bloated, easy to use without being limited, and customizable without requiring extensive user training. At one end of the spectrum, there are function libraries such as the *Visualization Toolkit*[1] (VTK) [79], which allows programmers to make just about anything they want, but requires extensive programming experience in a text-based programming language. Conversely, programs such as *Visual Molecular Dynamics*[2] (VMD) [35] provide a user with a ready-made, application specific visualization environment which is powerful to use, yet easy to learn. Ideally, we would like to break the boundaries of application specific programs, without having to increase the complexity of the platform.

It is a common solution when adding visualization to data streaming systems that application specific visualization tends to be added on an ad hoc basis, using custom functions that are highly specialized and platform dependent, e.g. [28] [37][92]. A related approach is to use an integrated development and visualization environment for event or data stream processing [21][80][87][97].

1 http://vtk.org
2 http://www.ks.uiuc.edu/Research/vmd

In ViSDM the data stream processing itself is provided through a general data stream management system, while LabVIEW provides a very powerful visual programming language in which the user easily can define custom visualization of data sets. A library of common controls provides the basic primitives for building the visualizations. The visualization primitives are highly customizable using a point-and-click interface and forms, and its visual programming capabilities offer a comfortable and intuitive way to create specialized solutions. However, LabVIEW does not have built-in support for continuous visualization of external data streams. This is provided by VisDM, through its library of VDFCs.

3.1 Data stream management systems

A data stream management system (DSMS, Figure 15) is similar to a database management system (DBMS) with the difference that while a DBMS allows searching only stored data, a DSMS in addition provides continuous query facilities to search directly in real-time data streams from one or multiple sources. The continuous queries can filter, transform, combine, and distribute the interfaced data streams. The result from a continuous query is also a data stream called a *derived* data stream.

A continuous query differs from its DBMS counterpart in that it may not have a determinate endpoint; it runs until the data streams feeding it are terminated or its operation is interrupted be the user.

A continuous query may have real-time properties which can pose concerns for the system in which it is running. The system must be able to process data at least as quickly as it arrives, preferably quicker than the arrival rate, since there must be room for processing user input and overhead (memory and resource management, concurrent processing, etc.).

Regular database queries, once started, usually cannot be modified. They are created, run, and return a result. Modifications to a query are made in between query executions. However, queries that run on a data stream management system may run indefinitely, and should preferably be altered without stopping and restarting them, when needed.

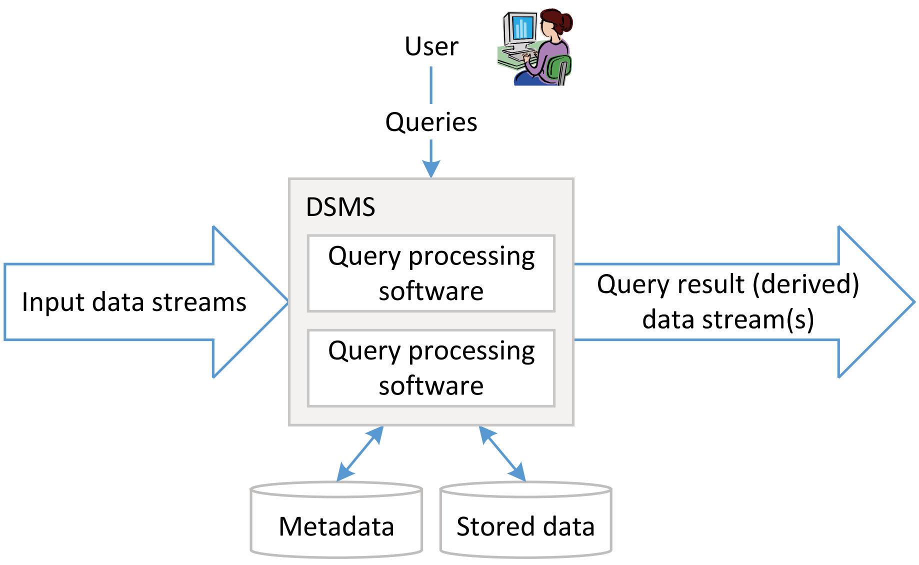

User
Queries
DSMS
Query processing software
Query processing software
Input data streams
Query result (derived) data stream(s)
Metadata
Stored data

Figure 15: The main building blocks of a data stream management system.

Amos II

The *Active Mediator Object System* (AMOS) [73][74] is an object-relational DBMS developed at Uppsala University. It is a main memory functional and extensible DBMS, with several appealing properties:

- Platform independence. As long as a computer meets some minimum system requirements, it can run a copy of the software. This includes embedded systems.
- Lightweight operation. The main memory and disk footprint is very small, counting in kilobytes.
- Sophisticated query optimization.
- A functional query language, called AmosQL [27], which is fully relational and compiles to predicate algebra.
- Tuple-by-tuple materialization of query execution, making it very responsive and ideal for handling continuous (non-ending) queries.

These advantages with Amos II – which is its current moniker – make it extremely adaptable, not just for data stream processing, but also data mining, distributed computing, and much more.

SCSQ

The *Super Computer Stream Query processor* (SCSQ) [96] is based on Amos II, and adds many stream processing capabilities through its query language SCSQL. Its most notable features are:

- The ability to start massively parallel stream query processes dynamically, adapting to the system load.
- Query language parallelization.
- Primitives for networked stream connections.

The main strength of SCSQ is how well it scales with the work load. This sets it apart from other stream programming languages such as Curracurrong [39], where work load distribution is static.

SVALI

The *Stream VALIdator* (SVALI, Figure 16) [93] is in turn built on top of SCSQ, and adds new functionality to streams:

- Predicate windows; an extension to the more static timing and counting windows found in other data stream management systems.
- Model learning; training a system to respond correctly to deviations in machine operation.
- Scalability; parallel streaming functions allowing systems with arbitrary complexity.

SVALI is the fundamental building block for all solutions presented in this Thesis, and has been thoroughly tested in the *Smart Vortex*[1] project [72].

3.2 Visual programming languages

With visual programming, programs are built using symbols and visual abstractions, rather than entering text. This way programming becomes more intuitive and can appeal to people who are uncomfortable with text-based programming [54]. Visual programming languages (VPLs) are usually limited in scope, and bound to a particular context or concept. For example, the *NXT* visual programming language (Figure 17) is used solely for controlling LEGO electronics kits[2].

1 http://smartvortex.eu
2 http://mindstorms.lego.com

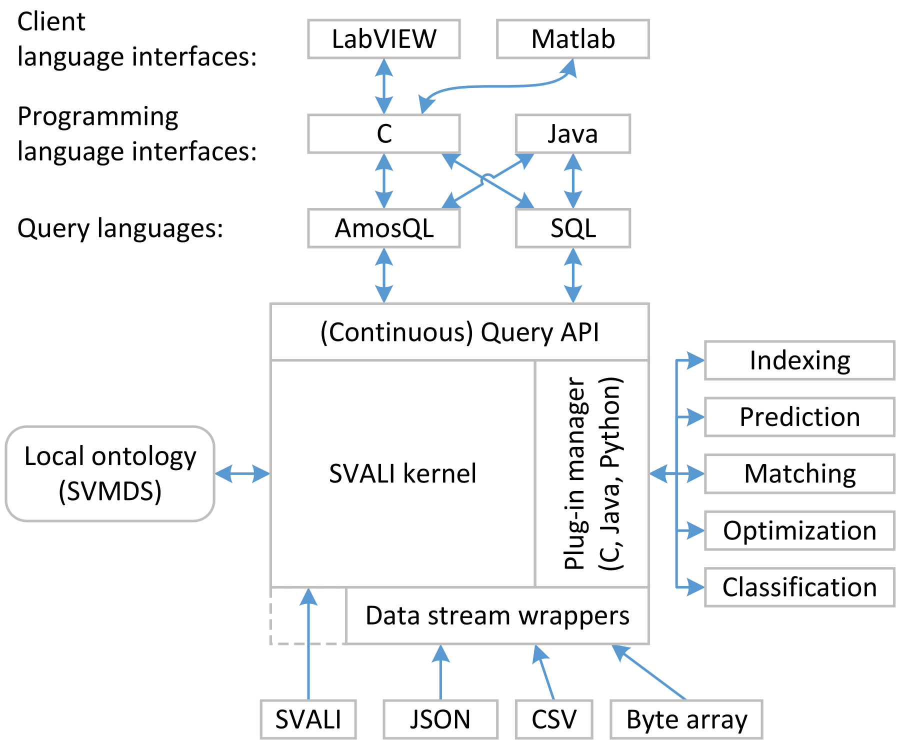


Figure 16: SVALI architecture.

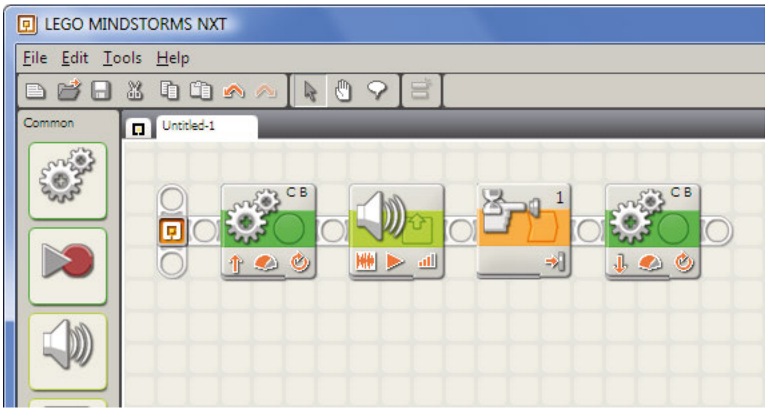


Figure 17: NXT programming environment
for LEGO MindStorms.

While not required, VPLs usually offer automation of several tasks, the main of which is resource management [38]; memory allocation, handling errors, etc. VPLs require an integrated development environment (IDE), where a user can create their programs, and there is usually only one proprietary IDE for each language.

Another common feature of VPLs is more or less sophisticated visualization of data output and user input. The user often has a library of text boxes, diagrams, plots, grid tables, push buttons, and more at their disposal, making user interface development trivial.

LabVIEW (National Instruments)

LabVIEW[1] [67] from National Instruments is a visual programming language (Figure 18), and has many properties that make it attractive to use for visualization: It maintains the user-friendliness of visual programming while still being very versatile and supporting many types of applications. It was first intended for controlling external measurement instruments and collecting data from those, but has since grown in scope and become the programming environment of choice for many engineers. The learning curve is flat, many complex tasks can be handled with ease, and it is easy to deploy applications during any part of development. LabVIEW comes equipped with many tool sets, and presentation of data is easy with preconfigured visual tools that do not need customization, for text as well as 3D graphics. It is easy to extend: functions compiled in a dynamic link library or shared object can be loaded at run-time and called dynamically. Like most VPLs, it offers automated resource handling and process management.

The programming language in LabVIEW is called *G* [57][59]. It defines all the components of the LabVIEW programming environment.

LabVIEW comes equipped with many components that are used for creating the VisDM client:

- An actor framework that forms the foundation for data flows in VisDM.
- Class polymorphism which enables dynamic type resolution.
- Extensive connectivity to external functions.

Data flows in LabVIEW are driven by control structures [2]. These structures unavoidably make much of LabVIEW code procedural, and because of this, *declarative-procedural* impedance mismatch is introduced should LabVIEW be used in conjunction with a DSMS.

1 http://ni.com/labview

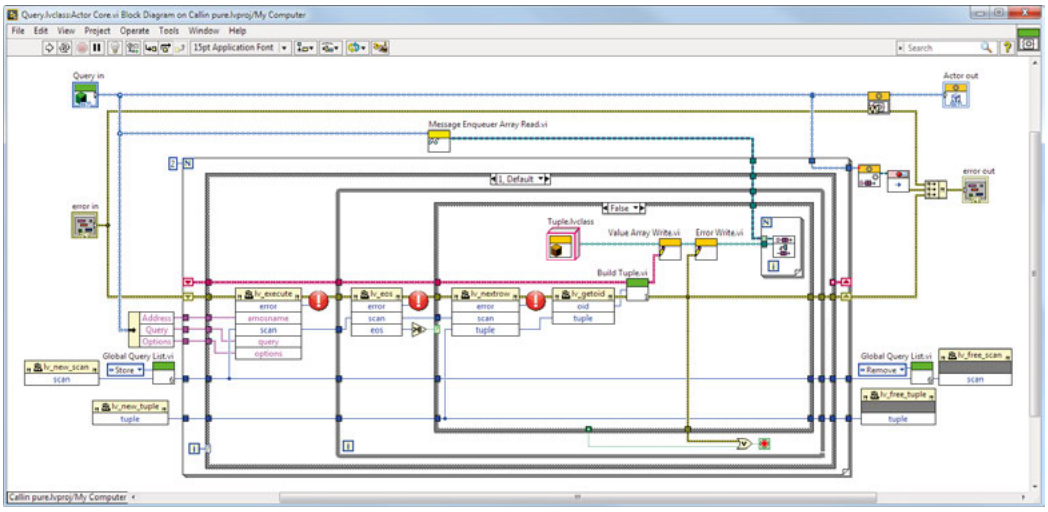


Figure 18: LabVIEW program example. This is the action loop of the actor for the RUN QUERY VDFC (see Chapter 4, "The VisDM system" on page 45).

Impedance mismatch

The term "impedance mismatch" originates from electrical engineering [88]. It was adopted by computer science to define the problems that may arise when two models, schemas, or technologies of different types are combined. The term is often used when describing the differences between object models used in programming and relational models used in database storage [36]. This is called *object-relational* impedance mismatch.

Query languages are declarative, meaning that the programmer states *what* operations they want performed, not *how*, as opposed to what is usually the case of procedural programming languages, such as C/C++, Java, Python, etc. However, since these are the languages we use to access databases, by the means of an application programming interface (API), we get a *declarative-procedural* impedance mismatch (D-P mismatch). D-P mismatch can increase the complexity of even fairly simple tasks significantly.

The common way of handling D-P mismatch is to introduce a *scan* primitive. A scan can be seen as a placeholder; calling a scan will return the next set of values from a query result, allowing a procedural language go through the result in an ordered manner.

```
SELECT timestamp, power FROM output;
```

This SQL statement is a simple example; we select all “timestamp” and “power” pairs from the table “output”. How this retrieval is done is not specified, but left to the DBMS to decide. By whatever means we execute this statement, it is preferable if this level of abstraction can be maintained.

```
rs = conn.execute(“SELECT timestamp, power FROM output”);
while (rs.next()) { // loop until we have exhausted the query
    ts = rs.getInteger(1);
    pw = rs.getDouble(2);

    // Do something with the values
}
```

In contrast, the above Java code snippet shows what is required of a Java API if we want to access the database output in that language. We have to specify what to do, and then how to do it. From this short example there are at least two issues to address:

- Extraction is bound to a while loop. Anything we want to do with the variables, we need to do inside of it.
- Resource management is prevalent. We need to make sure the right type of variable is retrieved from the right position in the scan, lest an exception is triggered.

The object *rs* (abbreviation of “result set”) is in this case the scan object.

In the same manner, visualization can also become a rather tedious endeavour. While there are very sophisticated tool sets available nowadays for visualizing data, they still force a user to focus on how to visualize something right after deciding what to visualize.

Any mismatch issue can be alleviated by a sufficiently advanced programming framework. The challenge is to introduce a framework that becomes less complex than the issue it is trying to resolve.

3.3 Data flow programming languages

In a visual data flow programming language (VDFPL) [38], it is often the case that a program specification becomes the program: a user specifies *what* should be done, and the programming environment takes care of the rest; *how* things should be done.

Figure 19 shows a simple diagram of data from a single stream source flowing through an operator that manipulates the data, and then to a display node presenting the data to the user. The diagram is completely declarative and easy to follow, and it works equally well for data stream manipulation and data flow programming.

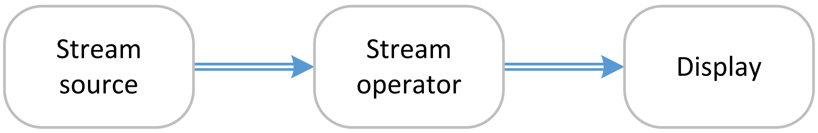


Figure 19: Data flow relationship between a stream source, an operator, and a display.

A DFPL offers several advantages compared to a procedural language:

- Order of execution is implicitly determined by how functions are wired, making DFPLs declarative, just as query languages are, which helps avoid D-P mismatch issues.
- Multi-threading and parallelization is completely automated; nodes may fire at the same time, as long as data is available.
- Functions do not have side effects and generally cannot become deadlocked, at least for a demand-driven DFPL [20].

Data streams v. data flows

There is one difference between data streams and data flows that plays an important part of program development: data flows must be semi-synchronous, in that the total amount of data in all wires or all variables must be equal if a program is to finish properly, whereas data streams can be completely asynchronous, running independently of each other.

A data flow function node will only execute once all inputs have a value. This means that one input must not fill up with values faster than any other. On the other hand, a data stream has its own source, producing values at its own rate, and therefore function nodes in a data stream may not be able to wait for values to arrive on all inputs.

It may not be obvious when either type of execution manifests. For example, a *sorted merge join* [50] function node may fire as soon as a tuple arrives on any input. A *union* [8] node on the other hand may only fire when all inputs have data. In the latter case, disparate stream rates require some form of *load shedding* [83][53] strategy to handle the data overflow.

Retaining values for incremental visualization

There are three plots displayed in Figure 20 that are updated incrementally from a streaming query. Different strategies exist for realizing the incremental plots, depending on the functionality of the platform.

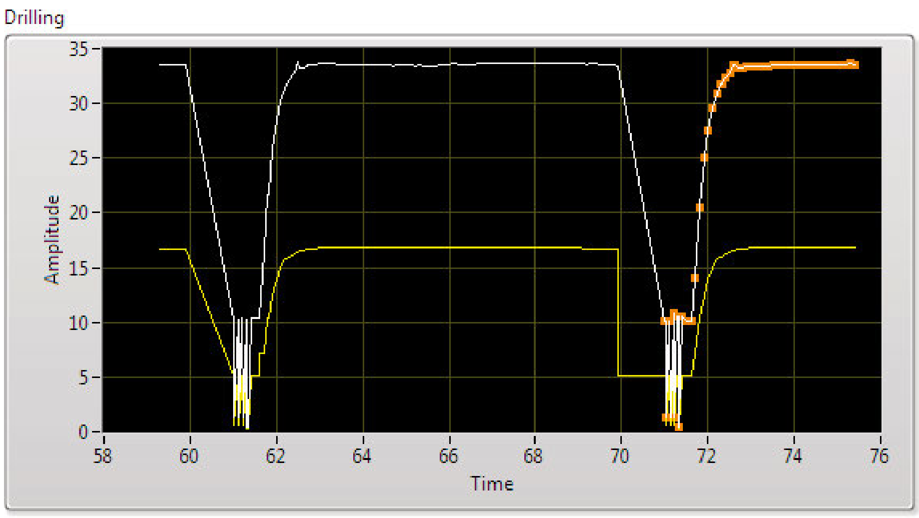


Figure 20: A LabVIEW *XY Graph* with three plots, running a machine monitoring and validation system.

1) A plot is a **sliding window** [29]. The visualization output is treated like the result of any data stream windowing function, and is created and maintained within the DSMS. The plot will be defined entirely in the CQ. For each display refresh, the entire plot is sent as a single tuple to the display diagram. There are two advantages with this approach:

- All logic is confined to the data stream management system. The visualization object will only display the data, without any need for further data management.
- LabVIEW diagram objects always expect arrays of points. The contents of the tuple become syntactically equivalent to the desired input for the object.

However, this approach comes with two rather big and obvious disadvantages:

- Plotting of streaming data tends to occur with small increments, meaning that data will be sent over and over again, resulting in very inefficient data transfer.
- Each tuple can become very big for large plots, which can strain the capabilities of the underlying system.

This method is better suited for small plots, and plots that are updated infrequently.

2) All plotting functionality is contained within the display object, which only accepts **incremental updates**. The display canvas is refreshed with each update, and the size of the plot is set in the object. This is generally an efficient approach,

with the drawback that it adds extra programming baggage to the block diagram. The arrays expected by the diagram objects must be handled in the implementation.

There is an approach to automate the incremental updates of a display canvas, by maintaining a history log of tuples in the data flow programming language. Whenever a tuple is retrieved from an input, it is possible to retrieve previous tuples as well. This is a feature of *temporal* languages [70][68], which all text-based data flow programming languages are. This is however not a feature of any existing visual data flow language.

3.4 Actors

Actors [1][33] are stand-alone, thread-based processes that communicate between each other using message queues. They are designed specifically with concurrent and distributed systems in mind. It is fairly straightforward to design a data flow environment using actors; each actor becomes a function node, and each entity in a data flow becomes a message that is sent from one actor to another. Practical implementations of data flow programming languages have existed for several years [38], and there are many who are looking into actor-based data flows [11] [48][94]. Actors are very well suited for parallelizing tasks, and work well with many different multi-core processor architectures [78].

The functionality of LabVIEW actors is illustrated in Figure 21. These actors contain two independently running loops: one *message loop* that handles *incoming messages* delivered in a message buffer queue and calls different *message functions* depending on the types of incoming messages. The *action loop* executes program-

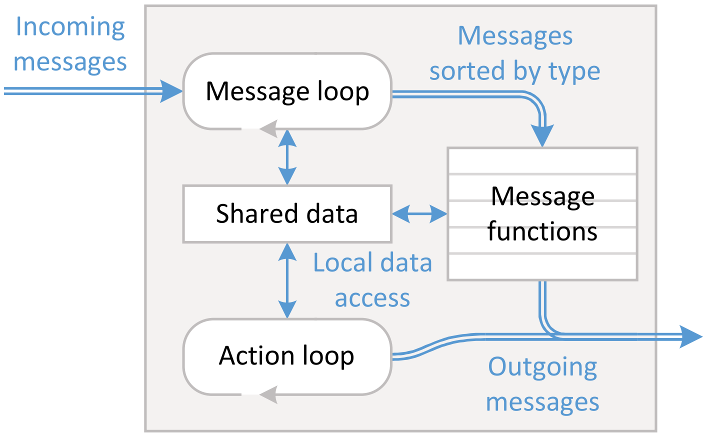


Figure 21: Basic layout of a generic actor.

mer defined tasks. Each LabVIEW actor has local *shared data*, available for all actor components. New *outgoing messages* can be created by the message functions or the action loop and sent to other actors, or back to the actor itself.

The action loop and message loop operate independently. Messages are handled one at a time and their corresponding functions execute serially.

Actors generally come with some infrastructure, which includes a startup phase, shutdown phase, and extensive error handling, all of which is fully programmable. This infrastructure is extended to support the data flow framework on which the VisDM VDFCs are based.

Data flow function nodes based on actors come with some advantages:

- The nodes operate independently of each other, taking advantage of parallelism without introducing race conditions.
- As tuples become messages sent between actors, operations follow the *single assignment rule* [17][86], which is a requirement for data flow programming. ■

4 The VisDM system

Now these points of data make a beautiful line.
And we're out of beta. We're releasing on time.
So I'm GLaD. I got burned.
Think of all the things we learned
for the people who are
still alive.

—Jonathan Coulton, *Still Alive*

Figure 22 shows a simple VisDM application that visualizes a stream in a continuously updated diagram of values representing the current power consumption of a milling process over a time window. Every LabVIEW program has two semantically separated views: a *front panel* containing the visualization and user interface (Figure 22) and a corresponding *block diagram* (Figure 23) that specifies the program.

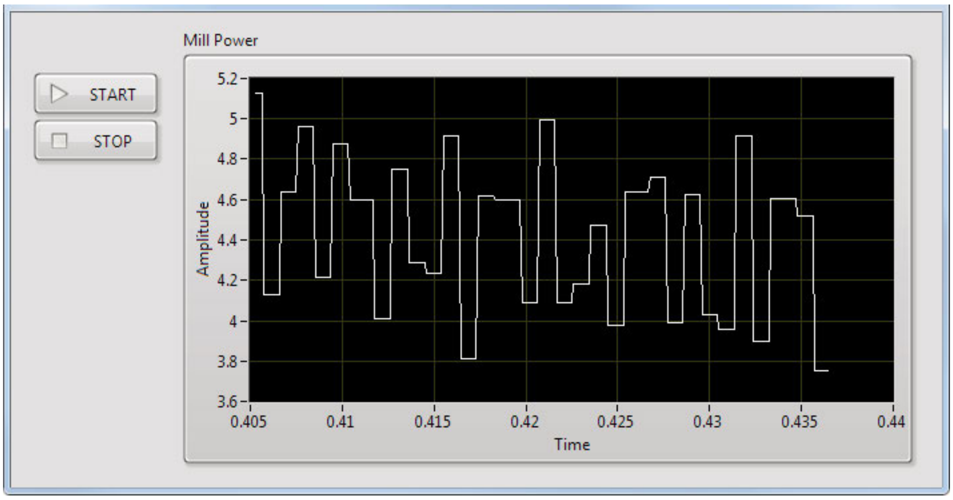
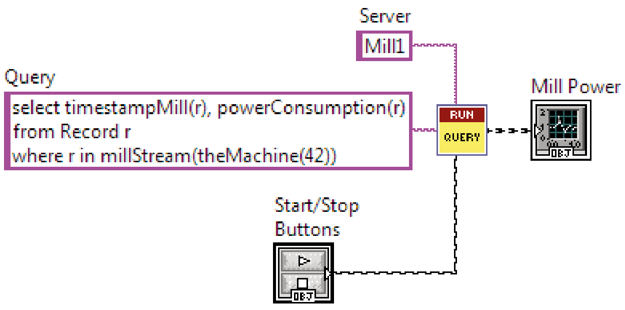


Figure 22: Continuous visualization of power output from a milling machine.



```
select timestampMill(r), powerConsumption(r)
from Record r
where r in millStream(theMachine(42))
```

Figure 23: Visual data flow specification.

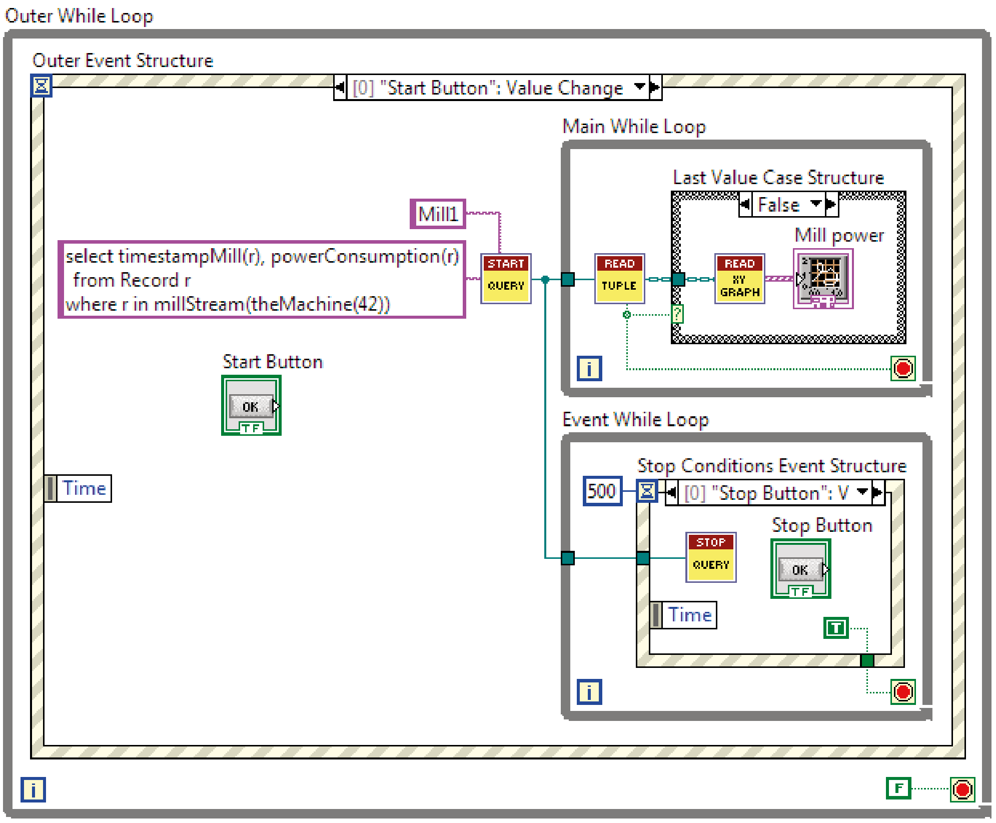
VDFCs are divided into *producers*, *operators*, *consumers*, and *controls*. Producers are the sources of data flows, typically a data stream from a CQ. Consumers are the end points of the data flows, presenting data to the user. Controls accept user input from the user. Operators are function nodes that manipulate data flows. A typical operator is a function node that extracts particular values from a tuple.

Figure 23 shows how the application is specified as a visual data flow in VisDM. In the example, the CQ on page 26 is running on a SVALI server named "Mill1". The red and yellow RUN QUERY VDFC node is a producer, a VisDM function node that is the source of a data flow. In this case the producer sends the CQ to the SVALI server and receives a stream of tuples that constitutes the output data flow represented in VisDM by the black dotted wire [1]. The output of RUN QUERY becomes the input of a VDFC node labelled "Mill Power" that represents the diagram in Figure 22. It is a consumer node that visualizes a stream using a LabVIEW graphical object, in this case an *XY Graph*. Graphical objects have labels that help identify front panel objects and their corresponding block diagram symbols. Pink solid wires denote strings in LabVIEW, e.g. the parameters of the RUN QUERY node.

Visualizing CQs requires some way for the user to start and stop each stream. VisDM provides this functionality through a VDFC representing start-stop buttons that controls the execution of a producer. Such control VDFCs are connected to the producer they control by a black ridged wire .

The program in Figure 24 is functionally equivalent to Figure 23, but uses conventional LabVIEW control structures. As can be seen, the non-procedural data flow code in Figure 23 is much more simple and easy to understand than the procedural code in Figure 24.

1 LabVIEW execution is always from left to right.



```
select timestampMill(r), powerConsumption(r)
 from Record r
where r in millStream(theMachine(42))
```

Figure 24: Conventional LabVIEW code.

A reason for the complexity is that each data stream should be visualized and controlled independently of other streams. The procedural definition is complex since the programmer has to specify in details how to iterate over each data stream, how to handle events, and how to terminate the stream gracefully. By contrast, the data flow specification is simple and straight-forward for visualizing each data stream, since it does not require detailed specification of the execution.

4.1 VDFC implementation summary

Programs in LabVIEW are called *virtual instruments* (VIs) [66]. VIs can run as separate programs, or can be called from other VIs as subroutines, then named *subVIs* [65]. VIs are defined procedurally using different kinds of control structures. As is apparent from Figure 24, subVIs are not self-contained and thus do not qualify as function nodes, unless explicitly implemented as such.

In order to make VDFCs behave like function nodes without any control structures, they are implemented using the LabVIEW actor framework [56]. The actor framework enables creating multiple independently running subVI processes that can communicate with each other asynchronously through message passing. The data-driven execution of actors allow VDFCs to operate independently of each other rather than through the rigid control driven serial execution of regular VIs.

Another issue is that SubVIs and actors alone cannot be used for defining consumers. The reason is that graphical objects that are included in a subVI cannot be made visible on the front panel of the main VI. In order to present graphical objects on the front panel as in Figure 22 of the main VI while encapsulating the actor functionality, VDFC consumer nodes are implemented using LabVIEW *XControls* [58]. XControls are specialized front panel objects that encapsulate other front panel objects and provide methods for handling different kinds of events. For consumer VDFCs, the XControls provide dynamic run-time behaviour defined by actors that are started by the XControls. This behaviour is provided by subVIs that are part of VisDM.

In addition, control VDFCs are also implemented as XControls, since they must encapsulate the code that controls data flow execution while providing the control objects on the front panel.

4.2 VisDM architecture

The architecture of VisDM is illustrated in Figure 25. There is a *SVALI server*, which is SVALI extended with a *service handler* to process CQs, *database* updates, and other SVALI *commands*. The *VisDM client* is LabVIEW extended with *VDFC definitions* for constructing data stream visualizations. It contains a *client API* to communicate with one or more SVALI servers. LabVIEW applications using the VisDM client framework can send commands to the SVALI server, for example to start CQs that filter and transform data stream from one or several *stream sources* accessed through SVALI. The result of a CQ is a *derived stream* which is sent to the VisDM client for visualization. VisDM client applications define data stream visualization by visual data flows, e.g. as in Figure 23.

A stream source can be, e.g., an embedded computer that outputs a data stream from a sensor onto a network, rows read from a data file, or a data stream emanating from a different computer.

A *stream wrapper* is a plug-in to SVALI that continuously converts data received from an external data stream into data structures supported by SVALI. The wrapper may leave all stream handling to an external agent such as Corenet and only retrieve the data from a broadcasting source, or it may have complete

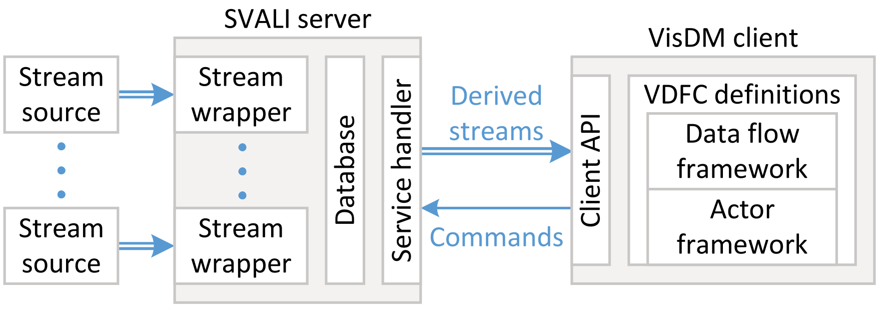


Figure 25: The client-server architecture of the VisDM system.

control of the stream source, setting it up beforehand and shutting it down afterwards. Data stream wrappers can be created visually with VisDM and plugged into SVALI.

The VisDM *client API* is called from the VisDM *data flow framework*, which contains data flow abstractions, dynamic type resolution, error handling, visualization support, etc. This framework is in turn based on the LabVIEW *actor framework*.

The user-generated commands back to the SVALI server can change the behaviour of CQs, e.g. changing a threshold, changing tuple rate, interrupting CQs, etc.

All queries running on a SVALI server are integrated: they run in the same address space and share access to the local database. As the number of streams increases, processing may become too much for a single server, and the execution must be distributed across several servers.

Architecture interfaces

The VisDM system has several layers of interfaces, the exact number of which depends on the nature of the wrappers used and whether any distributed systems are utilized. The schematic of a typical system can be seen in Figure 26.

The *LabVIEW API* provides an abstraction for interfacing the VisDM data flow objects with the *client C API*, which consists of a set of C functions for executing commands and retrieving data from CQs on the server.

The server CQ framework is built around a multiplexing *dispatcher* that receives incoming calls from a client and then issues server commands accordingly. The result streams of CQs are packaged in a special type of *scan*. In general, a scan is a generator that is used to step through iteratively a possibly infinite data stream. VisDM scans allow CQs to run simultaneously in the same server without interfering with each other, and provide general query maintenance.

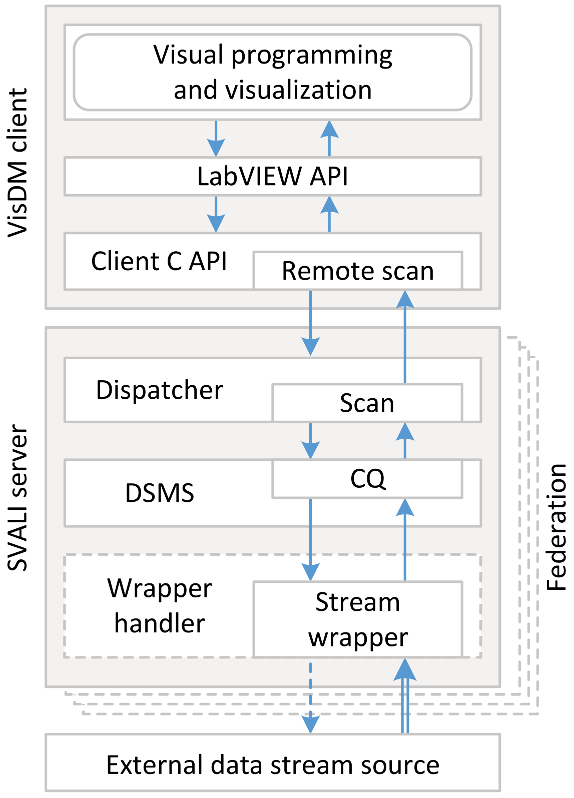


Figure 26: The interface stack of VisDM.

Scans reside on the server, while the client API uses *remote scans* to communicate with the server. They are placeholders for VisDM scans representing derived streams, and mainly add location independence to a scan, the physical location of which can be unknown to the user, and could even be undefined until run time.

A CQ calls its *stream wrapper* directly, or through a *wrapper handler*. The wrapper handler can dynamically load and integrate a stream wrapper with the server.

Every running server becomes part of a *federation*, where it is identified by a unique name. All servers in a federation automatically become aware of each other's existence, and adding new servers becomes as easy as just giving the server a name.

LabVIEW concepts

Before describing the implementation of VisDM in details, some basic LabVIEW constructs need to be explained. The VI in Figure 24 is used as an example and is at the same time explained.

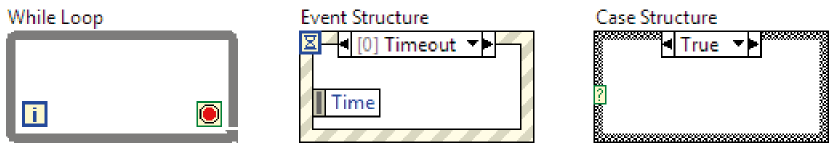


Figure 27: Commonly used control structures in LabVIEW.

Graphical objects in LabVIEW are divided into *controls* and *indicators*, where controls collect user input and indicators present outputs to the user.

Three types of control structures are used in Figure 24, and they are shown separately in Figure 27. A *while loop* executes the sub-diagram inside the frame over and over until a Boolean condition is met[1]. This condition is wired to the termination symbol ◉ and will cause the loop to exit when true. The blue square is a counter holding the current iteration.

An *event structure* will execute a sub-diagram when a certain event is triggered. It can have several sub-diagrams for separate events. It will execute the diagram associated with the triggered event, but only one triggered event will be executed at a time. Each event has a set of labelled attributes holding data pertinent to the event. Events can only be triggered if execution has reached the structure, making it wait for the events.

A *case structure* executes a certain sub-diagram depending on the condition wired to the condition switch ?. The conditions can be true/false, ranged, or enumerated. Event and case structures may contain several layered sub-diagrams, but only one is shown at a time. They can be flipped using the list at the top.

The outer while loop in Figure 24 is needed to restart the diagram after it has been terminated, since there can be several diagrams that are executed independently. The outer event structure waits for the user to press the start button before starting the data stream visualization. There are two controls used in Figure 24 to represent the start and stop buttons. Controls return a value, indicated by the triangle on the right-hand side OK, where the data type is indicated by the colour of the border. Pressing a button triggers a *value change* event that event structures can catch. Each button is located in its corresponding event structure for convenience.

The "Mill Power" node is an indicator that visualizes one or several stream elements. Visualizing a stream requires iteratively retrieving each stream element by a while loop. An iteration over a data stream stops either by the user pushing the stop button or when end-of-stream is reached.

1 As such, the while loop is really a do-until loop.

The START QUERY node initializes the CQ and sends a stream handle to the other nodes. The READ TUPLE node reads the current stream element from the handle in a while loop. The tuple must be converted to a LabVIEW diagram data type by the READ XY GRAPH node before being visualized by the "Mill Power" node. When a CQ ends, the READ TUPLE node will stop the main while loop. At that point there are no more tuples that can be visualized, and the tuple output from the read tuple node is undefined. Consequently, the diagram must be prevented from receiving data, which is accomplished by the last value case structure. It switches out the diagram, preventing it from executing further.

The stop condition's event structure will either wait for a pressing of the stop button, or a timeout. If the button is pressed, the stop query node runs and then the event while loop will exit. Every 500 ms, a timeout event is triggered at which point the structure will execute a sub-diagram polling whether end-of-stream is reached. If so, it will cause the event while loop to exit. Otherwise the loop will start over and the event structure will wait for new events.

4.3 Implementation of VisDM

Data flow programming comes in two flavours: *data-driven* or *demand-driven* [22][34]. With data-driven execution, function nodes produce output whenever data is available from the inputs. With demand-driven execution the nodes can only produce output when it is explicitly requested from a subsequent node. The data-driven approach is well suited for processing CQs, where code is executed whenever data arrives. The message driven operation of actors is equivalent to data-driven operation. Actors send messages at their own discretion regardless of the state of receiving actors. This means actors are well suited for designing a data-driven data flow framework.

In order to support the construction of data flow programs like the one in Figure 23, there are some shortcomings of LabVIEW that need to be solved. Using actors allow us to eliminate all control structures, but there are more issues that need to be addressed. The semantics of wires in VisDM do not match the control driven data transfer semantics of wires in LabVIEW. Also, the static data types in LabVIEW can handle neither CQs nor data flows, as the VisDM wires need to resolve the types of tuples at run-time.

The SVALI server allows CQs to be altered while they are running, by executing separate updates of the local database. For this purpose VisDM supports the creation of application-specific VDFCs that send commands to SVALI for execution.

Furthermore, stream wrappers can be created in LabVIEW and then loaded and executed in a running server, through a wrapper handler that can optionally be called from a CQ.

The RUN QUERY producer node

As illustrated in Figure 28, in VisDM the RUN QUERY node uses an action loop to retrieve tuples from a derived stream specified by a *CQ* that is sent to SVALI through the VisDM client API when the action loop is started. The API returns a handle to a *remote scan*, which is an interface to the derived stream returned from the CQ. The remote scan includes a *query signature* that describes the types of the objects received from the derived stream. It is forwarded to the subsequent actor as part of a *startup message* to describe subsequently emitted tuples. Then the remote scan is started by iterating over tuples as they arrive from SVALI one at a time through messages asking for the next tuple in the remote scan. The tuples are converted to LabVIEW representation according to the query signature and sent as outgoing *tuple messages* to other VDFCs. When there are no more tuples to receive from the stream, the remote scan is closed and a *stop message* is propagated through the data flow to close the VDFCs in the data flow. The termination is triggered by a stop message to the RUN QUERY node itself. When the stop message arrives, it is forwarded to subsequent VDFCs before the *shutdown* function of the RUN QUERY node is called to terminate the actor.

The visualization nodes

The flowchart in Figure 29 describes the operation of a VisDM visualization node. The shared data area is used for buffering tuples used by the display and for storing error messages.

The actor for a consumer node does not have an action loop, but has three message functions:

1) The *startup* function validates that the tuple signature corresponds to the display format and initializes the display. Caught error messages are stored in the shared data area.

2) The *update display* function converts each received tuple according to the format required by the display and refreshes it. LabVIEW requires all points in a visualization diagram to be stored in an array for each refresh. In the case when a tuple contains only a part of the points, the shared data area is used for accumulating points. The visualization can be either *incremental*, where the display canvas is modified for each received tuple, or *non-incremental*, where the canvas is completely redrawn for each new tuple.

3) The *shutdown* function simply terminates the actor.

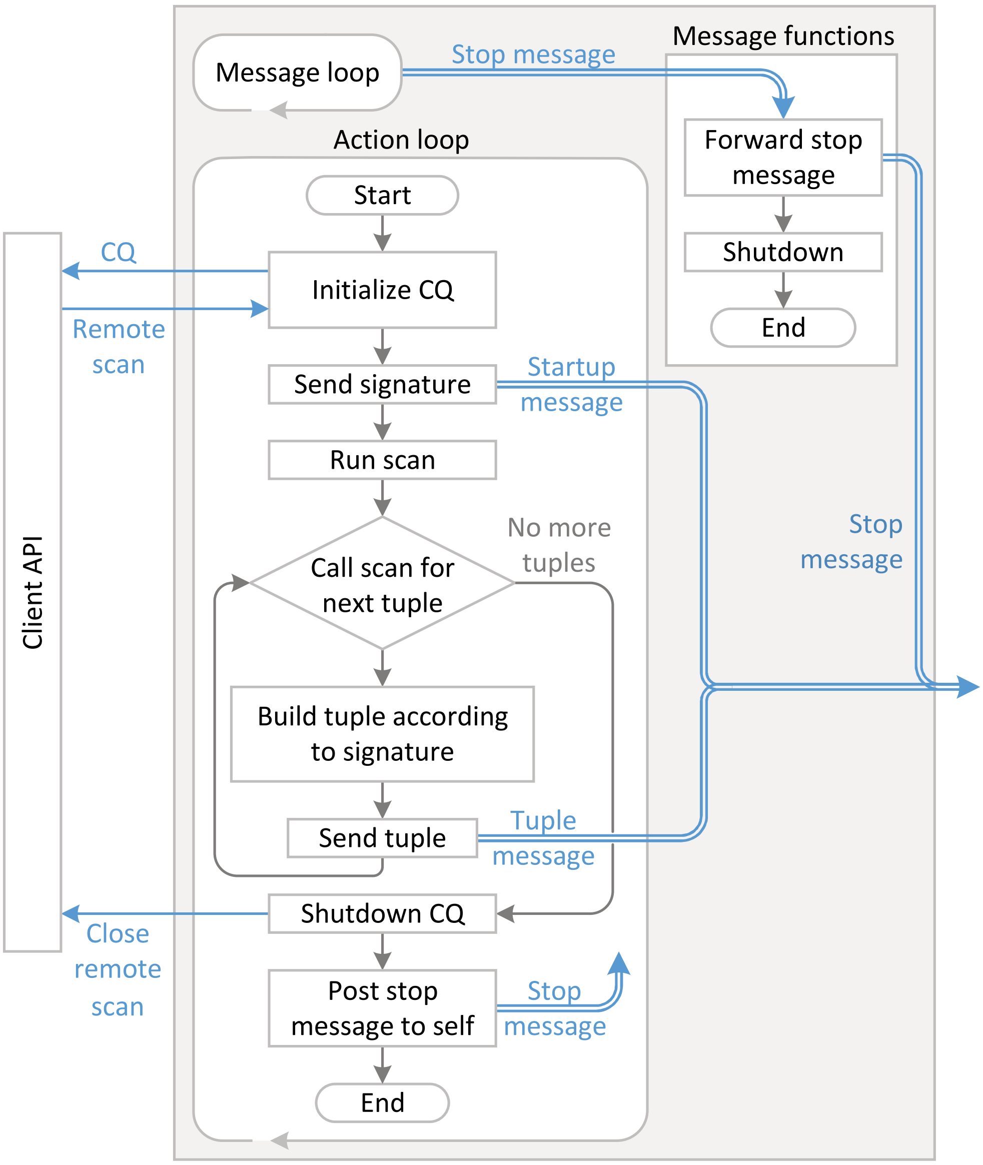


Figure 28: The actor operation in the RUN QUERY producer.

If an error is detected the actor will continue to accept incoming tuples, but will display error information to the user. The message loop may catch initialization errors, the startup function may catch validation errors, and finally the display function will present the caught errors to the user.

Appendix A.1, "Customizing visualization" on page 87 goes through the details of the visualization execution in a display node.

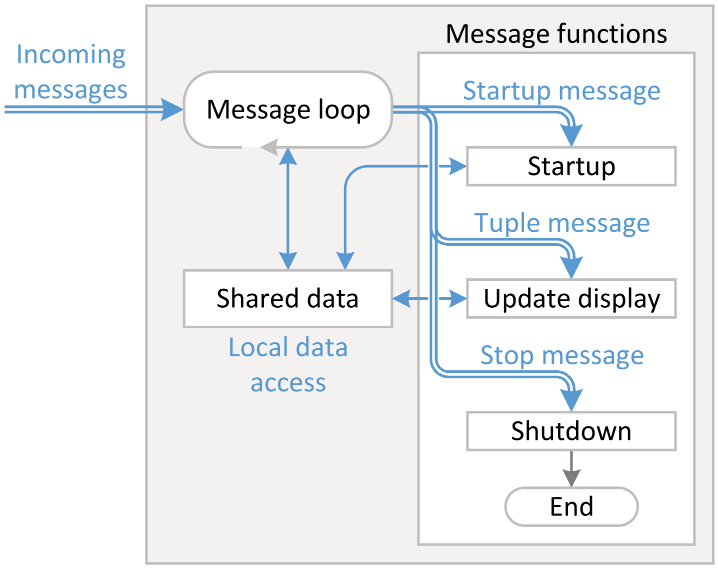


Figure 29: Actor operation for a visualization node.

Constructing the visual data flow in LabVIEW

Actors in LabVIEW communicate through message buffers. An actor sending a message to a receiving actor will put the message in the receiving actor's message buffer. When starting an actor, it will return a reference to its message buffer. In VisDM, each actor is created by a *starter* subVI that starts the actor and establishes the communication with other actors. To allow for several consumer actors to receive messages from a producer actor, a shared LabVIEW queue [63] is created by the starter of the producer and passed to the consumer starters. When a consumer actor is started a reference to its message buffer is added to the queue. For each message buffer in the queue the producer uses the message buffer to send messages to the corresponding consumer actor. When all message buffers are assigned, the producer can start its actor.

For details, see Appendix A.2, "Enqueuer transfer" on page 90 on how a data flow between a producer and a consumer in VisDM is set up.

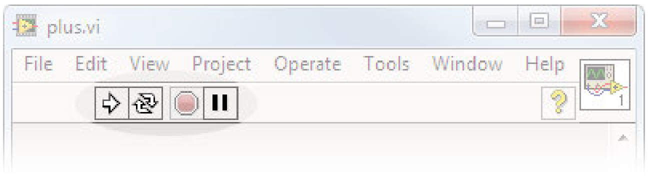


Figure 30: Execution controls of a VI.

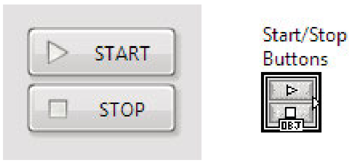


Figure 31: On the left, the front panel appearance of the execution controls. On the right, their appearance in the block diagram.

VisDM execution controls

The use of actors and externally running CQs has implications for how LabVIEW programs behave. The execution controls for a VI, highlighted in Figure 30, that are normally used do not work for VisDM clients. These controls only affect the state of the VI, whereas the execution of a VisDM application depends on many other things:

- Starting and stopping a data flow is dependent on the operation of the CQs.
- Actors run independently of the main VI and are not affected by its execution controls.
- As one VI can contain several data flows, custom controls offer more fine-grained execution control.
- The execution of a VI differs depending on the LabVIEW environment in which it runs, which can be the development environment, or the stand-alone *run-time engine* [61].

VisDM has a control VDFC containing start and stop buttons in a single XControl, shown in Figure 31. It is connected to the VDFCs that it controls. The operational steps for actually running VisDM data flows are as follows:

1) When opening a VI, if it is set to auto-run (as is the case with VIs running in the run-time engine), it will be stopped.

2) When pressing START, it will run the whole VI. Connected VDFCs will be flagged and start running. VDFCs not connected ignore the execution and will not start. Connected VDFCs whose actors are already running will likewise ignore it.

3) Pressing STOP will signal the pertinent scans on the server to interrupt their CQs. Visualization will automatically stop when a CQ has terminated.

Handling type resolution

LabVIEW uses strong typing while the types in the result tuples of a dynamic CQ are not known until run time. There is a possible conflict between the type structure of the tuples a CQ returns and the type structure of the front panel display elements of the VDFC. To validate that the types match, they are resolved in VisDM by the consumer actor's startup function. Furthermore, the update display function dynamically converts each incoming tuple into the format required by the front panel object. To enable the VDFC actor to handle any front panel object as a parameter, it is passed as a reference to the actor by the starter subVI.

There are two instances where the type conversion can fail. The first, more obvious one is when the tuple data types do not match the expected, statically typed output, when checked once by the startup function. The second kind of failure occurs when data can be converted but is otherwise malformed, for example when the order of data is different from what is expected by the visualization. This will not cause the visualization node to fail from running, but the output will become incoherent.

Tuples retrieved from a CQ first have to be converted to a format that can be managed by LabVIEW, and then converted to the native types required by the visualization objects used in the application. These steps are a consequence of *type* impedance mismatch.

Tuples must be copied because LabVIEW cannot manage references to external objects. In Figure 32, how many copies of the same string are there? It is impossible to tell (without referring to detailed LabVIEW memory management documentation at least). It is likely that some form of copy-on-write method is utilized, meaning that both diagrams contain exactly one string instance. Enfor-

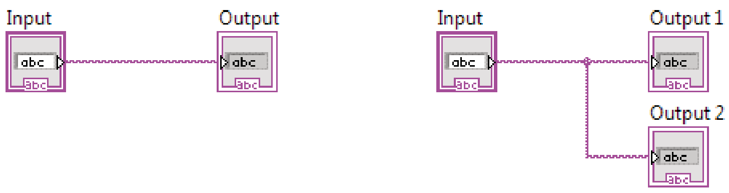


Figure 32: These two diagrams are indistinguishable programmatically.

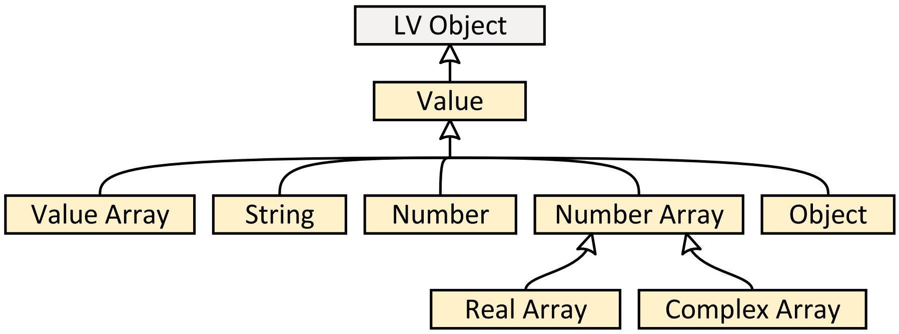


Figure 33: VisDM class structure for storing tuples.

cing some form of reference counting scheme on top of this is counterproductive and will not increase efficiency. Furthermore, LabVIEW classes do not have user-defined destructors [60], making implicit object termination impossible.

LabVIEW supports object-oriented programming and class polymorphism. This enables tuples to be imported and resolved at run-time. In VisDM, tuples are represented as arrays of values, where each value is a specific instance of a child class shown in Figure 33.

"LV Object" is the base class for all classes used in LabVIEW, and all custom classes will automatically inherit it. "Value Array" is the storage container for a tuple, storing any child class instance of the abstract class "Value". This includes other instances of "Value Array", making it possible to store recursive tuples, e.g. tuples containing arrays of points, for plotting diagrams. "String", "Number", "Real Array", "Complex Array", and "Object" are the types expected to be returned from SVALI. Number arrays are dedicated number types in SVALI, and therefore it is more efficient to have separate types for them instead of converting them to tuples. "Object" instances are application-specific objects that can be transferred between different servers, but cannot necessarily be translated to LabVIEW types.

Constructing the data flow

Figure 34 shows the current class hierarchy for the VisDM data flow framework.

All producers need a method "Propagate stop" that forwards a stop message to listening actors. Consumers need a method "Startup" that initializes the VDFC upon receiving the startup message, and a method "Process tuple" that handles each tuple message that arrives. Operators need all three, and the different numbers of methods may increase with future versions of the data flow framework.

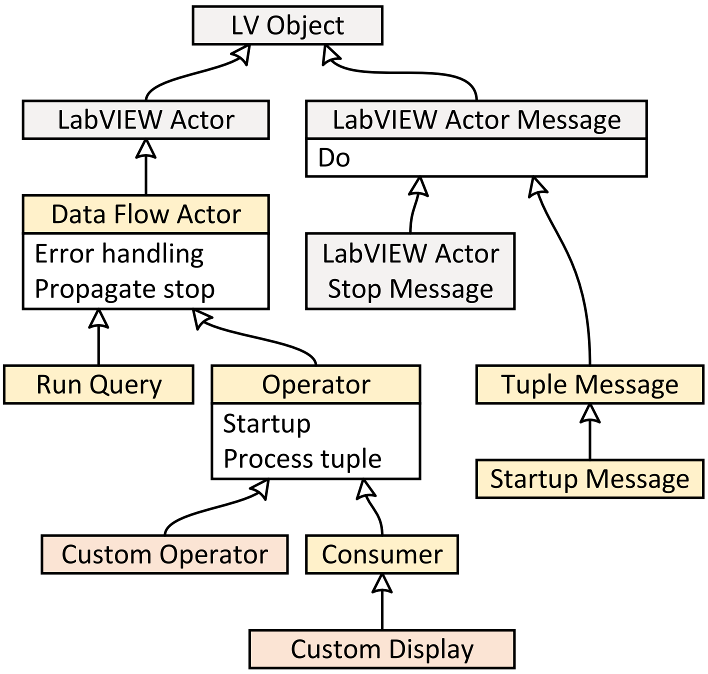


Figure 34: The current actor class structure for data flow emulation. LabVIEW actor framework classes have a grey label, data flow classes are yellow, user-defined classes are pink.

Methods in LabVIEW are VIs that belong to a class, and are therefore called method VIs.

The class structure in Figure 34 is complete for the current incarnation of the VisDM data flow framework. "Run Query" is the only producer class. Anybody adding a user-defined class for a new VDFC will have to implement their own version of the "Startup" and "Process tuple" method VIs.

All message classes must have a method VI named "Do". Whenever an actor receives a message, it will call that message's "Do" method. That VI will in turn call the appropriate message handling function of the actor. In the case of a tuple message, it will always be sent to an operator class, and it will call the "Process tuple" method of said operator. A startup message will be sent to a "Startup" method of an operator in the same way.

4.4 Running update queries

Stream monitoring might last for considerable lengths of time. Therefore a derived stream from a CQ can be altered while it is running by updating the local database through a separate connection. For example, a validation threshold that a CQ depends on may be updated while the CQ is running, which is different from regular database queries, where queries are isolated from updates.

An *update* VDFC specifies an update command sent to a SVALI server and requires as a parameter the name of the server. The command is sent when the user interacts with it. A practical example is the case of validation diagrams for Sandvik Coromant milling machines as illustrated by the front panel in Figure 35. There are four VDFCs: one visualization diagram plotting the mill stream output, the start and stop buttons, an update VDFC for entering a new threshold margin, and an alert indicator showing when the power output deviates outside of the margin. The power output from a machine is expected to stay within a certain margin from a predefined power output level. When power measurements deviate outside of this margin, an alert is signalled to the user. The margin for each machine is stored in the local database. It is altered whenever the user enters a new value to the update VDFC, sending an update statement to SVALI.

The block diagram for Figure 35 is presented in Figure 36. Two separate CQs are started on the server "Mill1", the result streams of which are sent to their corresponding visualization VDFCs: the power stream is emitted to the "Mill Power" diagram and the alert stream to the red LED indicator.

The alert stream runs separately from the mill power stream. Each time the alert status changes, it outputs a tuple with alert information. The stream rate for alerts is usually very low.

"Margin" is a VDFC that handles updates of server "Mill1". When a new number is entered into its text box, it is padded into a proper update command sent to the server. This is done in a subVI that is called from the update VDFC, as illustrated by Figure 37[1], where an update statement is built by the PREP STATEMENT VDFC. The margin parameter is inserted at the question mark placeholder.

The PREP STATEMENT VDFC builds the following update statement when the user enters "0.9":

```
set threshold_margin(42) = 0.9
```

1 "1.23" inside the "Parameter" node icon declares that the input to PREP STATEMENT is a number, "abc" inside the "Statement" node icon declares that the output is a string.

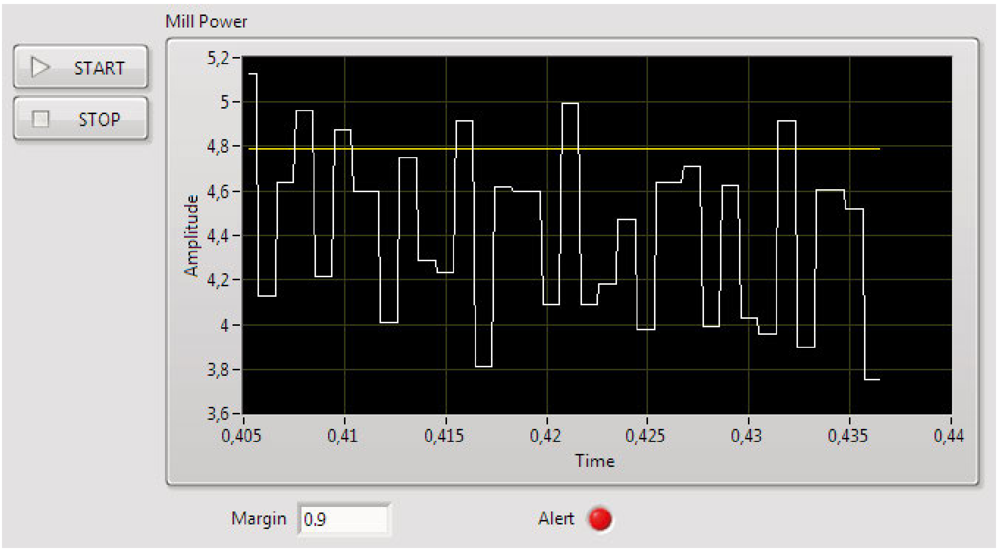
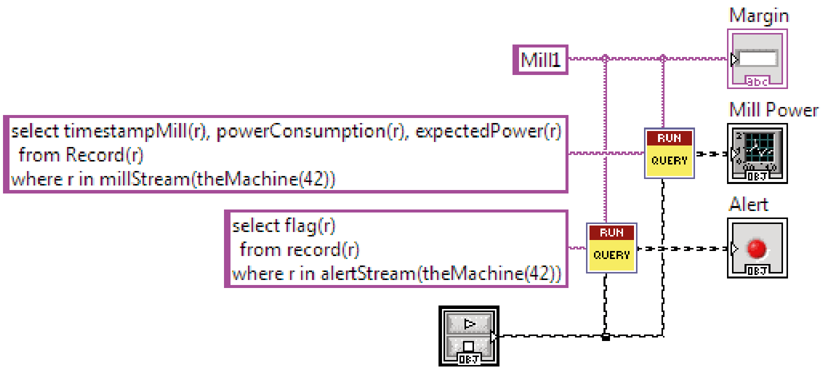


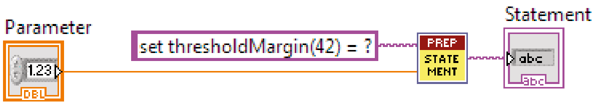
Figure 35: Plotting the mill power for a Sandvik Coromant milling machine. The yellow line marks an expected mill power output. The text box labelled “Margin” allows the user to set a new threshold margin during operation.



```
select timestampMill(r), powerConsumption(r), expectedPower(r)
 from Record(r)
where r in millStream(theMachine(42))
```

```
select flag(r)
 from record(r)
where r in alertStream(theMachine(42))
```

Figure 36: Adding update functionality to a block diagram with two CQ visualizations.



```
set thresholdMargin(42) = ?
```

Figure 37: The subVI for preparing the statement sent to the server.

Writing to a database is a violation of the single assignment [17][86] rule. However, databases are persistent, and they come with their own interfaces and protocols for ensuring safe interaction, and therefore could be considered exempt from the data flow programming rules.

4.5 Visual stream wrappers

A stream wrapper is a program module to handle communication between SVALI and external stream sources of a particular kind. It converts incoming data into streams of tuples emitted to SVALI. In the case of data from the Sandvik Coromant use case where Corenet streams were used, the stream wrapper consists of a set of Python functions that establish and maintain an SSL connection to a Corenet server, and convert the incoming stream elements.

A stream wrapper has three phases: 1) an initialization phase, establishing a connection with a stream source, 2) a retrieval phase where data tuples are continuously received from a source, converted, and emitted to SVALI, and 3) a shutdown phase where the stream is closed.

VisDM includes a framework to enable visual programming of SVALI stream wrappers as virtual instruments in LabVIEW. The VIs are dynamically loaded and executed in the SVALI server. Since VIs in general cannot be called directly from outside LabVIEW, the SVALI server has been extended with a wrapper handler that calls VIs compiled to a dynamic link library (DLL or shared object).

A visual wrapper example

Figure 38 shows a VisDM client display example, showing Fourier transforms of the radio signals collected from the LOFAR prototype antenna. VisDM was connected remotely to the antenna over the Internet, and used to display signal transformations in real time. In this application, both the visualization and the stream wrapper were defined visually with VisDM.

The block diagram is shown in Figure 39. Looking more closely at the CQ, there are some parts that warrant further explanation:

```
select abs(fft(na1)), abs(fft(na2)), abs(fft(na3))
  from number cnt, carray na, carray na1, carray na2, carray na3
 where (cnt, na) in fixstream(vi(“radio.vi”), “u2,ci2[366]”)
   and (na1, na2, na3) = il(na, 3)
```

The stream of tuples emitted consist of three arrays, one from each channel. *fixstream()* is the general VisDM wrapper handler for stream wrappers. The first argument of *fixstream()* is an interface object that loads the LOFAR stream wrapper VI named “radio.vi”. The second argument specifies the type format of the

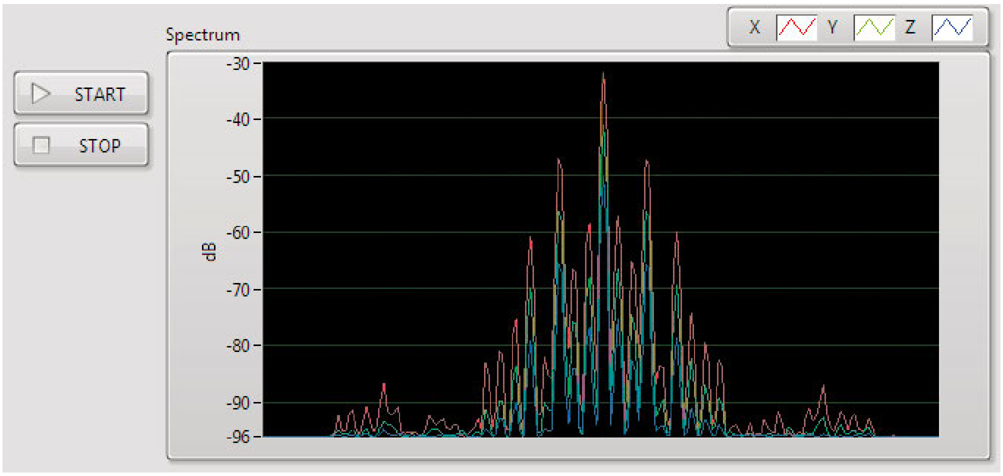
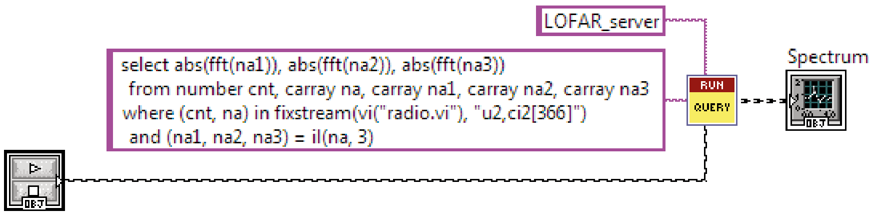


Figure 38: Displaying Fourier transforms of the three LOFAR antenna radio channels.



```
select abs(fft(na1)), abs(fft(na2)), abs(fft(na3))
 from number cnt, carray na, carray na1, carray na2, carray na3
where (cnt, na) in fixstream(vi("radio.vi"), "u2,ci2[366]")
 and (na1, na2, na3) = il(na, 3)
```

Figure 39: Sending the CQ to a SVALI server running LOFAR. Note that this query does not define the x-axis.

data stream, and is used to decode the tuples emitted from the wrapper handler. In this case the first value is a two-byte unsigned integer counter that indexes each tuple, and the second value is a numeric vector consisting of 122 interleaved, four-byte complex integer signal values from three channels. The function *il()* unbraids the separate radio channels, which are then transformed individually and emitted for visualization.

The operation of *fixstream()* is described in detail in Appendix B.1, "The fix-stream() wrapper handler" on page 93.

The block diagrams in Figure 40 to Figure 42 define the three phases of the stream wrapper VI to capture LOFAR data streams. Each phase has a corresponding case in a LabVIEW case structure, numbered from one to three. A VI that acts as a stream wrapper has the case number, "Case", as input and two out-

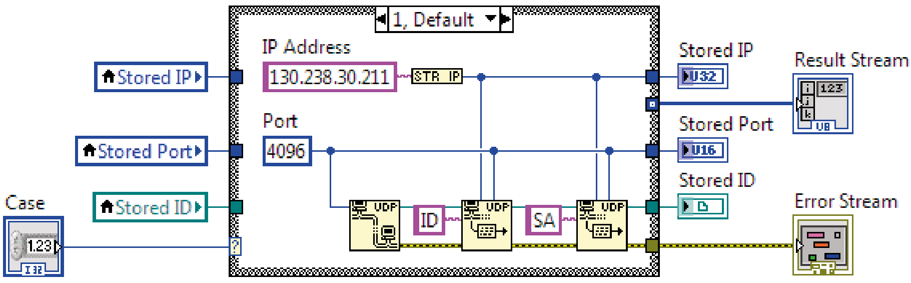


Figure 40: The startup phase of the LOFAR stream wrapper "radio.vi".

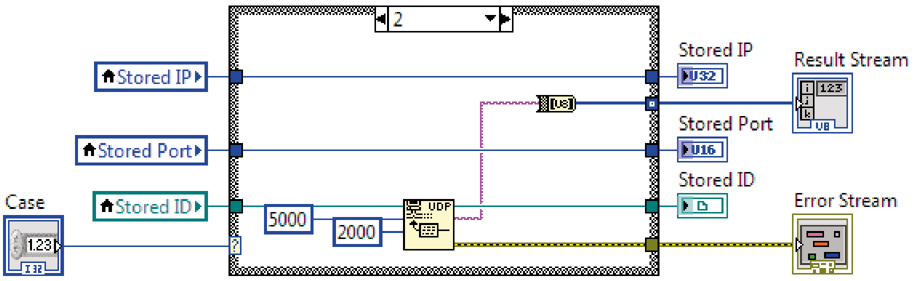


Figure 41: The emit phase of "radio.vi". It is called each time a new tuple is requested from SVALI.

puts, "Result Stream" and "Error Stream". Two streams are emitted to the VisDM wrapper handler: one result stream containing data tuples, and an error stream to signal errors.

The block diagram in Figure 40 shows the first, startup phase of the LOFAR stream wrapper "radio.vi". It establishes a UDP connection to the antenna and sends startup commands. The IP and Port of the antenna are stored in the wrapper as well as an ID of the UDP connection. No result stream elements are emitted.

The second phase in Figure 41 uses the values stored in the first phase. It is called each time a new tuple is requested from the wrapper handler. 5 000 bytes are allocated for each UDP package, which is converted to a byte array and emitted to the result stream, with a 2 000 ms timeout.

The third phase in Figure 42 sends a shutdown command to the antenna and then closes the connection.

Figure 43 shows the data path through the system. The stream wrapper and the compiled VIs run in an instance of the LabVIEW *run-time engine* [61], which is dynamically loaded into the SVALI server process by the VI wrapper handler. In

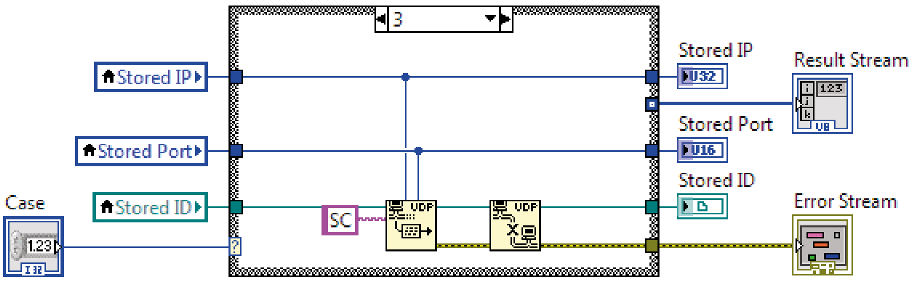


Figure 42: The shutdown phase of “radio.vi”, shutting down the connection.

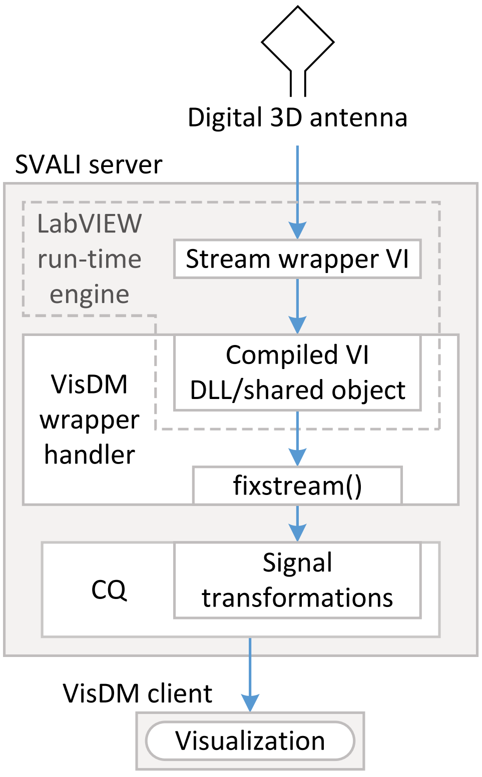


Figure 43: Radio visualization data path.

order to retain data values between the different phases of a wrapper, even when the same wrapper VI is used for more than one stream, the VI execution mode is set to *preallocated clone reentrant execution* [64]. This mode causes LabVIEW to create a separate instance of the VI for each separate call.

Setting wrapper parameters

There are two ways to set parameters for a visual stream wrapper:

1) **Server-side**. A VI can be set to open its front panel when it is loaded. Any parameters can then be entered directly into the stream wrapper. The advantage is that the wrapper can be altered directly while it is running and collecting data. The disadvantage is that this has to be done on the server.

2) **Client-side**. Any value for a control in a LabVIEW VI can be set remotely from another VI running in the same process, as long as the name of the control is known. This allows parameter input to be scripted, and controls can also be updated while running, through a dedicated update VDFC.

4.6 Server and API details

Both SVALI and LabVIEW can be embedded in other programs. Furthermore, both can in turn call embedded components. A bare-bones SVALI DLL has been embedded in the VisDM client to aid with peer communication, details of which can be explored in Appendix B.2, "Interfacing LabVIEW with embeddable components" on page 95.

Most APIs that are used for accessing queries base their operation on a *scan* entity. While scans encapsulate queries from the user, they do not by themselves encapsulate queries from each other. This is the purpose of a special type of *coroutine* [90][69][19], on which the VisDM scans are built. By using coroutines, queries can be scheduled so that their operation do not come into conflict with each other. But they also need a mechanism by which they avoid blocking the system when doing something that is peripheral to the query operation, such as waiting for data input. Coroutines are discussed in detail in Appendix B.3, "Coroutines" on page 96.

Remote scans add connection transparency to scans. A remote scan is used by the client to connect to a scan on a server, but it can also be used in more general terms as a logical entity that represents a physical entity located elsewhere. A remote scan behaves exactly the way a scan does, and effectively eliminates perceived physical distances in a computer network.

Scans and remote scans are discussed further in Appendix B.4, "Scans" on page 99.

The coroutines inside scans as well as stand-alone coroutines run in tandem using cooperative multitasking, on the multiplexing server. For details on how this works, see Appendix B.5, “Server structure” on page 101.

4.7 Evaluation

As previously shown, VisDM has successfully been tested in two real-world applications, one industrial at Sandvik Coromant and one academic at the Ångström Laboratory, Uppsala University.

Initial performance studies of the VisDM system have been carried out, evaluating the practical limits of its visualization capabilities.

All applications are defined as producer-consumer pairs, matching a CQ data stream source to a visualizer data stream sink.

Sandvik Coromant machine tool monitoring

In this use case [93], the task is to provide a portable system for easily customizable visualisation of CQs, where the operator can start new client windows and make changes to those already running, depending on the operational parameters. To enable this, the VisDM system was deployed to visualize machining data streams, connecting a laptop directly to monitored machines in production on the factory floor. Setting up a running system required only a few minutes of VisDM configuration, consisting mostly of mapping data in arrays to corresponding variable names.

There can be many machines running at different sites. Each machine may be unique and perform several different tasks [77]: turning, threading, milling, drilling, boring, etc. The number of sensors varies and there can be many parameters measured, up to 40 for some machines. The input used is machine control system data currently sampled at ~200 Hz[1]. The derived streams that are calculated from the input streams can have any frequency, down to fractions of a hertz for alarm streams. Visualization of streams from each machine runs independently from the others, which is well supported by VisDM’s data flow primitives.

Since the result of a Corenet stream is visualized as a sliding window, the visualization is incremental.

1 Data from external sensors such as dynamometers are often in the range of 2–5 kHz, but they are not covered in this use case.

LOFAR antenna unit

LOFAR consists of about 20 000 antenna units operating in tandem, in effect becoming a very large radio telescope. All units are identical and their operation is unlikely to change during their run time. Instead, operation is focused on intensive calculations over high-rate, possibly massive data streams. The radio signals are received as signal vectors consisting of interleaved, complex numbers from three channels. As specified by the CQ on page 62, an FFT is applied to each channel and then visualized non-incrementally by VisDM.

In this case, the visualization update speed is determined mainly by two things: the radio signal bandwidth, and the size of the signal vectors. Higher bandwidth means more data, but bigger vector size for higher resolution means slower update speeds. Running the receiver at 44.1 kHz bandwidth generates ~367 UDP data packages per second. The call to *fixstream()* in Figure 39 on page 63 converts each received byte array from the "radio.vi" stream wrapper VI to a tuple consisting of a counter and a vector of 366 complex numbers. The carrier wave frequency does not affect the data speed, because the carrier wave is subtracted in the antenna hardware before transfer. Displaying data packages as they arrive requires the VisDM client to update display objects at a rate of ~2.7 milliseconds, or 370 hertz.

Evaluation of VisDM visualization performance

The test applications in their current state do not toe the limits of VisDM by any means. Furthermore, it is of little practical use to display values faster than the eye can perceive or the computer screen can muster. The 370 hertz update frequency in the LOFAR case is much higher than is needed for presentation purposes, and this is also true for the 200 hertz frequency in the Sandvik Coromant case.

Displaying tuples as they arrive can be a convenience, as it can save some configuration time and generally increase the system flexibility. The chart in Figure 44 shows maximum update frequencies in kilohertz for a query running in VisDM, returning a stream of arrays consisting of complex, double values (16 bytes), and having them plotted in VisDM. The chart specifically shows how the maximum update frequencies relate to the tuple size. Testing was done on an Intel Core i7-4770 @ 3.40GHz running Windows 7. ■

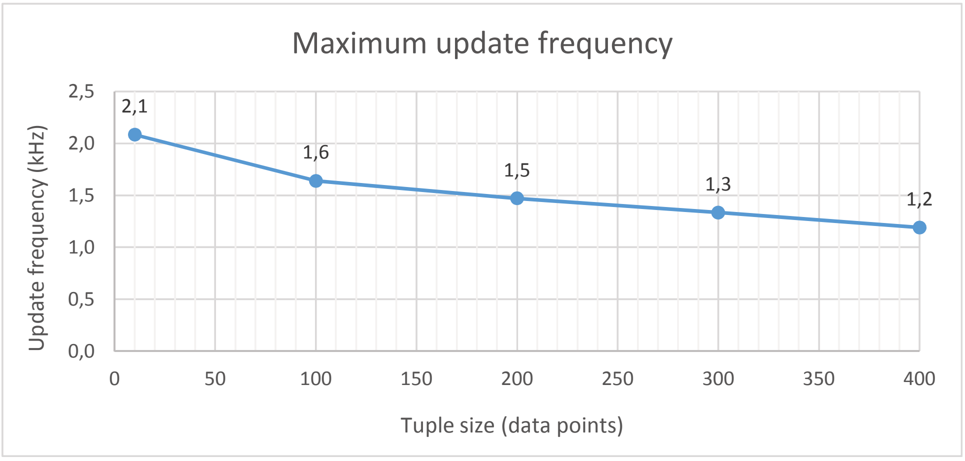


Figure 44: Relation between tuple size and update frequency.

5 Related work

"You seriously planned all this mad scientist stuff? I mean, when was this hopeless black fantasy supposed to happen? When were you planning to do it?"

"'Do it?' Dan, I'm not a comic book villain. Do you seriously think I'd explain my master-stroke if there remained the slightest chance of you affecting its outcome? I did it thirty-five minutes ago."

—Alan Moore, *Watchmen*

The power of VisDM is its ability to combine powerful, visually defined data stream visualization with a state-of-the-art DSMS in one versatile and expressive working system by using an existing, commercial, and general visual programming language extended with data flow primitives to provide powerful customizable visualization components.

Visual data flow solutions with strong visualization capabilities are becoming more widespread. The platforms described below combine data stream management, data visualization, and data flow programming to various degrees. They are usually based on a client-server structure with a point-and-click interface. They are marketed to the public, either as commercial systems or downloadable function libraries. There are experimental systems that have been developed by various computer science research groups, but they usually have limited scope, or they are not being developed or maintained anymore [34][15].

A basic prerequisite of any system that is useful for data stream management is the ability to handle data streams with disparate stream rates. One way many solutions do that is by imposing a predefined structure or schema on the tuples in a stream, the simplest form being to prepend a time stamp to tuples [23]. Others that perform pattern matching need tuples to contain type and structure metadata [75].

SVALI does not impose requirements on the nature of data streams. It is up to the implemented model, through the schema it utilizes, to impose restrictions for how data can be accessed and processed.

5.1 Data streaming examples

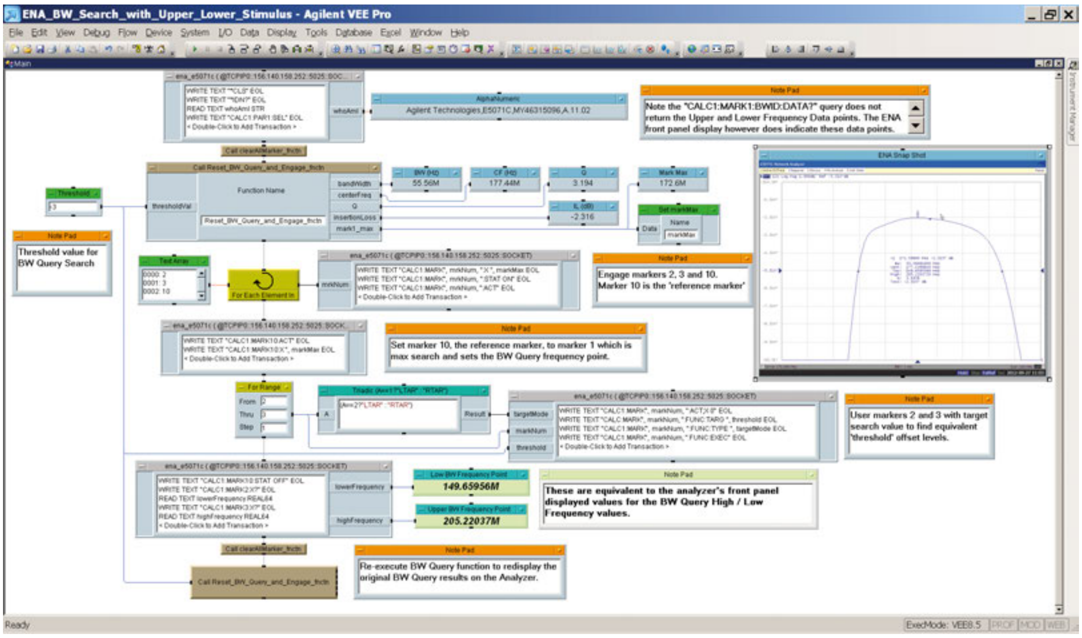
There are platforms that try to provide complete solutions for all data streaming tasks, such as *IBM Streams*[1] [12] and *SQLStream Blaze*[2] [82]. Both have their own comprehensive query languages, and support point-and-click visualization through a web-based interface. However, the visualization is based on JavaScript templates, and customization – to the extent that it is available – must be programmed manually in JavaScript. By contrast, all VisDM visualization is defined visually, including any customization, using a readily available and comprehensive toolkit.

Complex event processing (CEP) provide tools and techniques for analysing and controlling complex series of interrelated events [45][16]. A central part of CEP is pattern matching and actions over complex event sequences rather than analysing streams of measurements. Some prominent examples of platforms for handling CEP are *Tibco StreamBase*[3] [87], *DataWatch Desktop*[4] [21], *ZoomData*[5] [97], and *Software AG Apama*[6] [80]. Function extensibility ranges from non-existent (DataWatch Desktop) to very good (StreamBase). These systems have dedicated, non-generic visualization components that can be sophisticated (zooming, information extraction, etc.) but are much more limited compared to the functionality VisDM offers through LabVIEW.

Rickshaw[7] and *Plotly*[8] are function libraries for visualizing streaming data. They are programmed in JavaScript – Plotly is available for other languages as well – and can be used with various data stream sources. Programming is text-based and supports only visualization without any primitives for connecting to or managing data streams.

Comparing VisDM with the systems mentioned above, VisDM leverages on the rich visualization and data stream programming capabilities available in LabVIEW to provide support for advanced engineering and scientific applications.

1 http://ibm.com/software/products/ibm-streams
2 http://sqlstream.com/blaze
3 http://streambase.com
4 http://datawatch.com/products/datawatch-desktop
5 http://zoomdata.com
6 http://techcommunity.softwareag.com/ecosystem/communities/public/apama/products/apama
7 http://code.shutterstock.com/rickshaw
8 https://plot.ly



```
ena_e5071c ( @TCPIP0:156.140.158.252:5025:SOC...
WRITE TEXT "*CLS" EOL
WRITE TEXT "*IDN?" EOL
READ TEXT whoAmI STR
WRITE TEXT "CALC1:PAR1:SEL" EOL
< Double-Click to Add Transaction >
```

whoAmI: Agilent Technologies,E5071C,MY46315096,A.11.02

Note the "CALC1:MARK1:BWID:DATA?" query does not return the Upper and Lower Frequency Data points. The ENA front panel display however does indicate these data points.

Threshold: -3

Threshold value for BW Query Search

Reset_BW_Query_and_Engage_fnctn

BW (Hz): 55.56M | CF (Hz): 177.44M | Q: 3.194 | IL (dB): -2.316 | Mark Max: 172.6M

Text Array: 0000: 2, 0001: 3, 0002: 10

```
ena_e5071c ( @TCPIP0:156.140.158.252:5025:SOCKET)
WRITE TEXT "CALC1:MARK", mrkNum, ":X ", markMax EOL
WRITE TEXT "CALC1:MARK", mrkNum, ":STAT ON" EOL
WRITE TEXT "CALC1:MARK", mrkNum, ":ACT" EOL
< Double-Click to Add Transaction >
```

Engage markers 2, 3 and 10.
Marker 10 is the 'reference marker'

```
ena_e5071c ( @TCPIP0:156.140.158.252:5025:SOCK...
WRITE TEXT "CALC1:MARK10:ACT" EOL
WRITE TEXT "CALC1:MARK10:X", markMax EOL
< Double-Click to Add Transaction >
```

Set marker 10, the reference marker, to marker 1 which is max search and sets the BW Query frequency point.

For Range: From 2, Thru 3, Step 1

```
Triadic (A==1?"LTAR" : "RTAR")
(A==2?"LTAR" : "RTAR")
```

```
ena_e5071c ( @TCPIP0:156.140.158.252:5025:SOCKET)
WRITE TEXT "CALC1:MARK", markNum, ":ACT;X 0" EOL
WRITE TEXT "CALC:MARK", markNum, ":FUNC:TARG ", threshold EOL
WRITE TEXT "CALC1:MARK", markNum, ":FUNC:TYPE ", targetMode EOL
WRITE TEXT "CALC1:MARK", markNum, ":FUNC:EXEC" EOL
< Double-Click to Add Transaction >
```

User markers 2 and 3 with target search value to find equivalent 'threshold' offset levels.

```
ena_e5071c ( @TCPIP0:156.140.158.252:5025:SOCKET)
WRITE TEXT "CALC1:MARK10:STAT OFF" EOL
WRITE TEXT "CALC1:MARK2:X?" EOL
READ TEXT lowerFrequency REAL64
WRITE TEXT "CALC1:MARK3:X?" EOL
READ TEXT highFrequency REAL64
< Double-Click to Add Transaction >
```

Low BW Frequency Point: 149.65956M
Upper BW Frequency Point: 205.22037M

These are equivalent to the analyzer's front panel displayed values for the BW Query High / Low Frequency values.

Re-execute BW Query function to redisplay the original BW Query results on the Analyzer.

Figure 45: A sample VEE program. Note the yellow elements which are iterator functions, and the beige elements which are calls to external functions.

5.2 Visual data flow programming

A visual programming platform which shares many similarities with LabVIEW is *VEE*[1] [41], from Keysight Technologies. Just as with LabVIEW, its main intended application is to serve as a frontend to, and work in conjunction with, various measurement instruments. And just as with LabVIEW it can be programmed to perform all sorts of different tasks. The data flow programming model in VEE is much closer to the data flow programming paradigm than LabVIEW, and programs are not dependent on control structures for their operation. Instead they depend on *iterators*, *switches*, and *merges* [20][38].

Programs in VEE are read from left to right and from top to bottom. Wires connecting to the left and right-hand sides of a node are data wires, and the ones on the top and bottom are sequence wires. For example, the yellow iterator node with a circular arrow in Figure 45 is connected on all four sides. The leftmost wire contains loop parameters, the rightmost wire outputs an iteration value each time the topmost wire transmits a trigger, and the bottommost wire outputs a trigger on each occasion it happens.

1 http://keysight.com/en/pc-1000003078%3Aepsg%3Apgr/agilent-vee

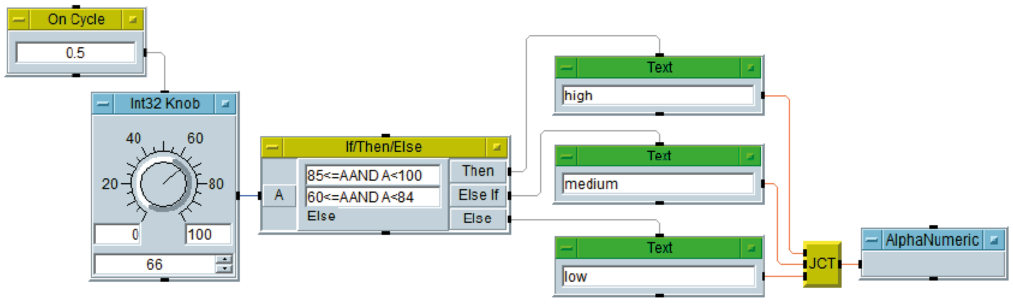


Figure 46: A VEE program with a condition entity.

The program in Figure 46 shows the use of an iterator, a switch and a merge. Every half a second, the knob control (the iterator) outputs a value, and depending on the value the "If/Then/Else" box (the switch) outputs different triggers. The "JCT" box (the merge) collects the data flows, and the "AlphaNumeric" text box shows the resulting text.

Unfortunately, VEE has insufficient extensibility for complete integration of the system with a DSMS as VisDM requires. Polymorphism is also not supported, meaning that there is no way of handling dynamic typing for tuples.

Apache NiFi[1] [3] (Figure 47) is a browser-based visual data flow programming language. It can be connected to external data streams and to visualization toolkits, but completely lacks those capabilities by its own.

Streams, Blaze, and StreamBase all provide their own data flow programming environment. An example in StreamBase can be seen in Figure 48. In StreamBase, writing to a database is a side effect of some of the nodes, which is a deviation from pure data flow programming, because the operation of the function nodes is not restricted to the nodes themselves.

5.3 Platform comparison

In Figure 49, the most important features of all these systems have been collected in a single table. There are some notes to be aware of:

- The systems mentioned that have a DSMS also have a built-in query language. This does not have to be the case; there could be only an API available.
- Visually defined visualization can still mean that extending the visualization requires manual, text-based coding, in the cases where comprehensive visualization tools do not exist.

1 https://nifi.apache.org

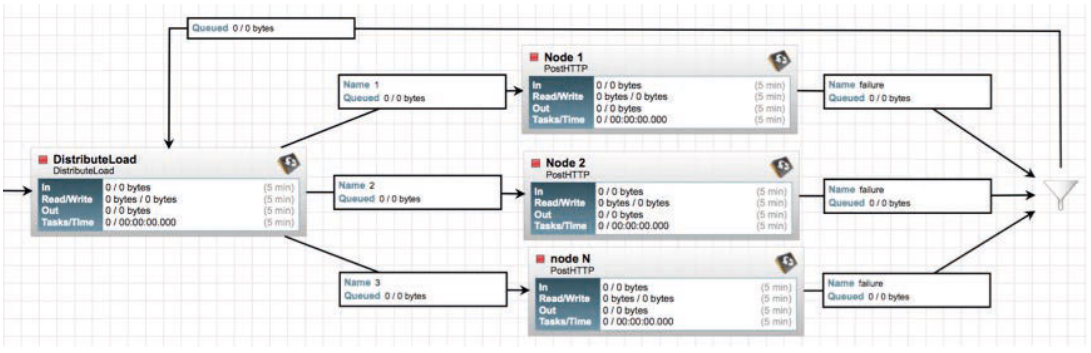

Queued 0 / 0 bytes
DistributeLoad
DistributeLoad
In 0 / 0 bytes (5 min)
Read/Write 0 bytes / 0 bytes (5 min)
Out 0 / 0 bytes (5 min)
Tasks/Time 0 / 00:00:00.000 (5 min)
Name 1 Queued 0 / 0 bytes
Name 2 Queued 0 / 0 bytes
Name 3 Queued 0 / 0 bytes
Node 1 PostHTTP
Node 2 PostHTTP
node N PostHTTP
Name failure Queued 0 / 0 bytes

Figure 47: Apache NiFi example application.

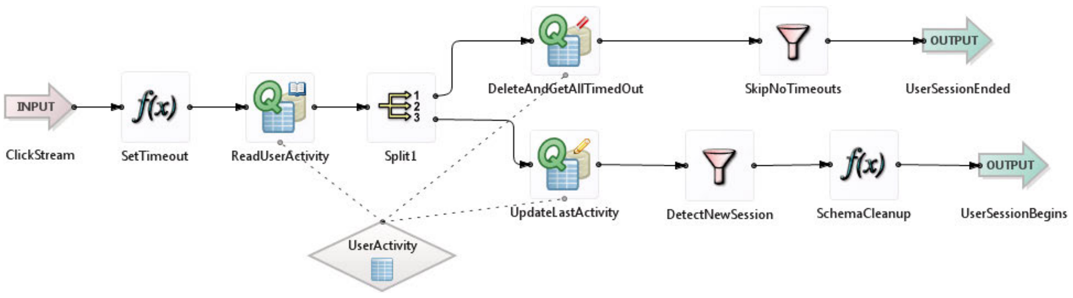


Figure 48: An event flow diagram in StreamBase.

- “Extensible stream sources” means that the platform can be extended to connect to new stream sources. In the case of CEP platforms, they are still limited to certain types of sources.
- Supporting streaming data does not necessarily mean that the streams are real-time and high-rate. It is often the case that the streaming is done from some kind of repository, like a data warehouse.
- All platforms that have visualization based on JavaScript naturally have browser-based visualization. LabVIEW provides browser-based visualization through a dashboard based on Microsoft Silverlight. This dashboard is available for Android, iOS, and Windows 10 as well.
- Continuous updates are updates to data stream filtering while an application is running. Only VisDM provides updates to CQs running on a server.

	VisDM	IBM Streams	SQLStream Blaze	TIBCO StreamBase	Software AG Apama	ZoomData	DataWatch Desktop	Rickshaw	Plotly	Keysight VEE	Apache NiFi
Built-in query language	✓	✓	✓	✓	✓						
Visual, declarative data flow applications	✓	✓	✓	✓						✓	✓
Visually defined visualization	✓	✓	✓	✓	✓	✓	✓			✓	
Extensible visualization	✓	✓	✓	✓		✓		✓	✓	✓	
Comprehensive visualization tools	✓			✓						✓	
Integrated DSMS	✓	✓	✓	✓	✓						
Stream source types	All	All	All	CEP	CEP	CEP	CEP				
Extensible stream sources	✓	✓	✓	✓	✓	✓					
Real-time data streams	✓	✓	✓	✓							
Browser-based visualization	✓	✓	✓	✓		✓		✓	✓		
App-based visualization	✓										
Continuous updates	✓		✓			✓					

Figure 49: Features comparison.

5.4 Visual query builder

In most cases, continuous queries are programmed in a text-based language. In contrast, [10][47] describe the *SmartVortex Visual Query System* used for SVALI [Paper III]. It is aimed towards letting inexperienced programmers build complex queries. With a point-and-click interface, the user can visually define any query.

Unlike the other platforms presented in this chapter, this is not a complete, stand-alone system. Its editor, shown in Figure 50, capitalizes on many advantages with visual programming:

- The interface significantly flattens the learning curve for people who are unfamiliar with query programming.
- Wiring is strictly hierarchical and eliminates most, if not all, risks of syntax errors.
- The two-dimensional programming style supports code reuse, at least to a limited extent, e.g. type definitions can be shared.

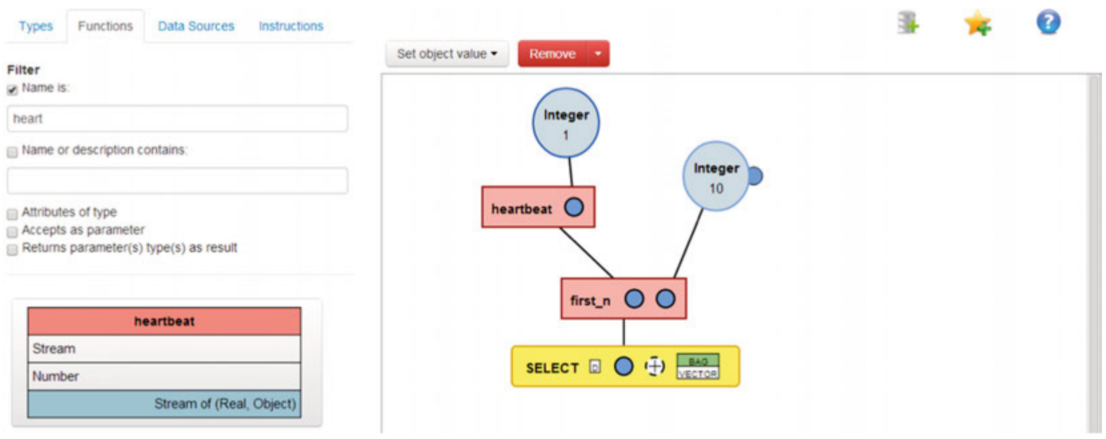


Figure 50: The query editor for the visual query builder.

The programming style of the builder shares many properties with data flow programming – strict function hierarchies notwithstanding – and is somewhat similar to DFQL [18], PICASSO [42], and other visual query languages, where focus is on the graphical representation of an already established query language and increasing the usability of said language, without adding any new technologies or semantics to the language. ■

6 Summary

"Er, what if … if I'm not in front of one when it tries to hit me? What if it is in fact behind me?"
"Ah, well, I am afraid that in that case sir has to go back and start all over again, sir."
"And, er, how do I do that?"
"Being born is traditionally the first step, sir."

—Terry Pratchett, *Thud*

Visual Data stream Monitor (VisDM) is a platform for online analysis and visualization of data streams. It has a stream-oriented client-server architecture and utilizes visual data flows for connecting continuous query results with appropriate real-time visualization displays. VisDM provides easy data flow specification to specify continuous visualizations of CQ results. The data flow specification is simple and straight-forward for visualizing each data stream, since it does not require detailed specification of the execution. Visual data flows enable declarative specification of application programs visualizing data streams defined as CQs to a DSMS.

VisDM integrates a visual programming language with a data stream management system (DSMS) to support the construction, configuration, and visualization of data stream applications. To achieve this, the LabVIEW visual programming platform has been adapted to support the easy specification of continuous visualizations of CQ results. LabVIEW comes with many different graphical objects for visualizing data, both in 2D and 3D. They are mainly intended for statistical and signal processing data, but can handle other types of data as well, and in the case that they should not be sufficient, new customized visualization tools can always be created.

To enable visual specification of interfaces to external data stream sources, VisDM includes a framework to enable visual programming of SVALI stream wrappers as virtual instruments in LabVIEW, which can then be loaded and executed in a running server, through a wrapper handler that can be called from a CQ.

LabVIEW has been extended with a toolbox, Visual Data Flow Components (VDFCs), which enable declarative visual specification of visual data stream applications as visual data flows. The set of VDFCs is extensible, so that adding new components when needed is easy. The declarative, data flow centric programming with VDFCs does not rely on control structures the way regular LabVIEW programs do. Thus VisDM extends LabVIEW with a data flow framework on which the VDFCs have been created. The data flow framework utilizes the actor framework of LabVIEW. With actor-based data flows, visualization of data stream output becomes more manageable, avoiding the procedural control structures used in conventional LabVIEW programming while still utilizing the comprehensive, built-in LabVIEW visualization tools.

VDFCs are divided into producers, operators, consumers, and controls. Producers are the source of data flows, typically a CQ that runs on a server. Consumers are the end points of the data flows, typically displaying the CQ results. Operators perform manipulations of data flows, most commonly extracting values from tuples. Controls accept input from the user, like sending commands to the server (update VDFCs) and starting/stopping a data flow. The VDFCs allow the user to focus on what they want done, while minimizing the trouble with how to do it.

To implement the VisDM system, LabVIEW was interfaced with the SVALI (Stream Validator) data stream management system. In order to operate as a query server, SVALI was extended with components for handling multiple CQs concurrently. For client-server access to continuous queries, a special type of scan primitive and a dispatcher for switching between running CQs have been added to SVALI. Furthermore, it has been extended to dynamically incorporate visually programmed stream wrappers, for the handling of external stream sources.

VisDM has been applied on two different real-world problems in order to evaluate its effectiveness: one on industrial machining and one on processing high-volume radio telescope data streams. For both applications, data is visualized in real-time, and VisDM is capable of sufficiently high update frequencies for processing and visualizing the streaming data without obstructions.

The strength of visual data flow programming is the expressive power, making data stream management and visualization easy and intuitive, even for users who are unfamiliar with the concepts. Using symbolic building blocks interconnected with wires, a program definition becomes the program itself. The visual programming tools of LabVIEW provide a foundation for visualization and programming of data streams. Combined with the SVALI, we get a well-rounded and very flexible solution for all sorts of data stream management tasks.

Data streaming applications and visualizations are easy to define, configure, and deploy using VisDM. It works well both for inexperienced programmers as well as experienced ones, depending on the application. It offers strong visualization capabilities without limiting the unique data streaming capabilities of SVALI.

A unique strength of VisDM is its complete extensibility. No other system offers the same capabilities for adapting to such a vast range of data streaming and visualization tasks.

6.1 Discussion

While LabVIEW works adequately for providing a base for VisDM, it is far from ideal in many aspects. Much of the loss of performance lies with XControls. Most of that loss comes from having to use references for visualization. Using references in LabVIEW instead of the actual objects is inefficient to varying degrees, depending on the type of reference. XControls are by themselves relatively resource heavy just because of how they are integrated with the run-time environment. Furthermore, typecasting of objects is inefficient compared to other object-oriented languages.

Still, the advantages that LabVIEW brings to the table may render all other issues inconsequential. There is seemingly no other solution that can offer this combination of sophisticated visual programming, visualization tools, and extensibility.

LabVIEW graphical objects and in particular diagrams more often than not will require extra infrastructure to operate properly. For example, diagrams only accept arrays as input, containing the complete plot data. It is up to the programmer to facilitate these arrays. This kind of customization is difficult to encapsulate without using XControls.

Static typing in LabVIEW becomes a roadblock, because it hinders the adoption of a system-wide schema.

LabVIEW graphical objects are not customizable the same way that widgets are in other language frameworks. For one thing, they do not have support for callback functions, i.e. user defined functions that are called by the framework. The details of setting up an example front panel object using an XControl is presented in Appendix A.1, “Customizing visualization” on page 87.

LabVIEW XNodes

Many LabVIEW functions have connector panes that can be customized in many ways. Connections to a subVI on the other hand cannot be altered without having to make several different versions of the VI.

The three boxes to the left in Figure 51 show how the array indexing function can be extended to retrieve several values at the same instance from an array. The two rightmost boxes show the function for flattening data to a string, which can accept literally anything that can be transferred through a wire.

This flexible behaviour can be created using XNodes[1][2]. They provide a form of meta-programming, where the behaviour and appearance of an XNode is defined by a set of VIs that the programmer provides.

This can be particularly useful for type resolution, where the dynamic conversion of tuples can be solved using overloading in an XNode, the appearance of which can be completely adaptable to different applications.

The reason that XNodes are currently not included in any solution is because they are not officially supported, and subject to change at any moment. Documentation is quite insubstantial and only a few tutorials exist. They are generally not recommended for use.

6.2 Future work

Several topics lie ahead, warranting further investigations. The current system is merely a prototype; there are both issues with the architecture that need to be solved, and practical applications that need to be investigated.

Client-side management of a distributed DSMS. A possible future work is to extend VisDM with VDFCs that visually define data flows deployed across several distributed compute nodes, executing CQs or other stream computations in parallel. The VDFC implementation should automatically parallelize the execution across any number of compute nodes, making node management transparent to the user.

For massively parallelizing heavy computational processes [95], a parallel computation could be specified graphically by dropping VDFC nodes on the block diagram.

Generalized and streamlined system pipeline. Most industrial machines today that require automated monitoring come with embedded computers that collect and distribute sensor data. If the machines are equipped with embedded,

1 http://labviewwiki.org/XNodes [unavailable at the time of writing.]

2 https://lavag.org/files/category/10-xnodes

Figure 51: Examples of the versatility of functions.

off-the-shelf single board computers running Linux or similar, then each board can run an instance of SVALI, which can be represented by function nodes in VisDM. This way, the whole data streaming architecture from machine to user can be administered from a single VisDM client.

Common for all systems utilizing custom components is that there is always an initial step where raw data is produced which must then be converted to a format that can be handled by subsequent platforms. This adds extra steps to the data stream pipeline, and requires more resources. Basing the entire pipeline on VisDM eliminates these extra steps. Sensor data still need stream wrappers with this model, but they can be included as light-weight functions in SVALI, instead of being separate processes.

Pluggable visual query builders. Constructing queries in a visual programming environment, instead of manually typing them, offers many of the same advantages that visual data flow programming provides: a flattened learning curve, elimination of many syntax errors, making the code more manageable, etc. The extensibility of both SVALI and LabVIEW makes it easy to attach a visual query builder. The SmartVortex Visual Query System [28] is intended to be used in conjunction with SVALI, but other extensions may be incorporated as well. DFQL [18] is one such system, PICASSO [42] is another one.

Pluggable backends. It should be investigated whether data streaming platforms such as IBM Streams and SQLStream Blaze may serve as backends to VisDM.

Visual stream operators. SVALI can be extended with a framework for dynamically loading and executing visually defined streaming functions, by using VisDM. These functions may work as functors [85].

Automated stream monitoring. There are currently no mechanisms for having a server and a client automatically exchange information about their states. Ideally, a client should be able to convey the nature of desired data stream input, and the server should if possible cater to those requests. There are some issues that may benefit from this communication:

- Tuple buffering. Automatically adjusting buffer size and timeout according to changes in stream rates can influence perceived visualization behaviour, and decrease the load on computer resources.
- Data stream health status. Automated supervision and monitoring of the various components of a data stream management application can be very beneficial to a user, who may not have a clear understanding of what can go wrong if a system suddenly becomes unresponsive.

Visualization can become a heavy load for the client and might slow it down so much that it cannot retrieve data as fast as it is produced. In those cases, some form of load shedding protocol is needed. There are various strategies for load shedding [83][7][53], but they can all be divided into two categories:

1) **Pruning.** Filtering functions are applied that discard less interesting data and hopefully preserves the more interesting bits. Pruning works well, at least in theory, with the current system architecture, since it can easily be incorporated in a distributed data stream management system as filtering functions.

2) **Aggregation.** Statistical functions are applied to extract the essence of the data, without needing to preserve the actual data.

While pruning seems like the better choice at face value, the fact is that aggregation has some appealing properties for visualization:

- The data that is to be visualized has probably already been pruned in one way or another, and the alternative left is simply to discard data that cannot be visualized in time.
- The data to visualize may already be the output from aggregation, e.g. a running average, and thus only needs minimal adjustments. Which leads to the next point:
- Aggregation works well together with visualization, because the reduction rate can be quite significant, and one can make flexible solutions such as the one shown in Figure 52, where the aggregation is done in two steps with intermediate database storage, with the two functions or queries running in parallel. The daemon process is handled through the VisDM server extensions.

Preferably, load shedding should be automatic. One possibility is to have the client operators informing the server of their largest acceptable delivery rate. This data can then be picked up by the data stream management system, allowing it to inject the appropriate aggregation or pruning functions when processing the streaming data. ■

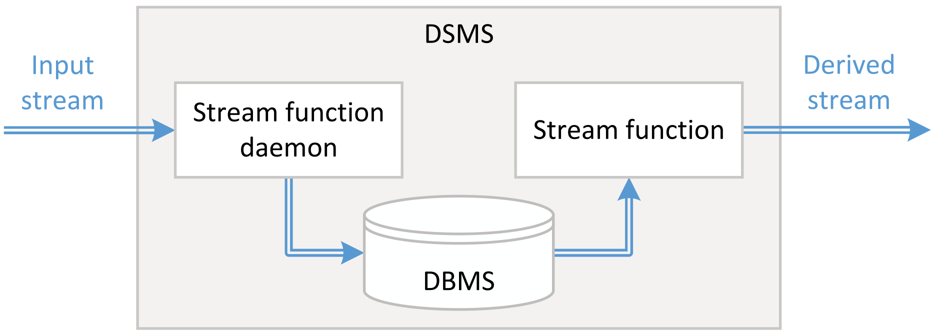


Figure 52: Balancing stream rate by running two asynchronous stream functions with intermediate database storage.

Appendix A – LabVIEW programming

This appendix goes into details and minutiae of LabVIEW-specific solutions to programming issues. Certain knowledge of how LabVIEW programming works and of available functions is required.

A.1 Customizing visualization

Design of a visualization object and its operation are divided between an XControl and a class inherited from the "Consumer" class in Figure 34 on page 59. This class becomes the driver of the visualization for the XControl.

The front panel presented in Figure 53 shows a number box embedded in a façade VI. Every XControl has this VI, and the canvas of its front panel becomes the face of the XControl when used inside another VI.

Figure 54 shows the corresponding block diagram for this façade VI. It handles the brunt of all XControl operations, which in this case is not so much, since the diagram will only run once, upon receiving the tuple stream object. The important part is the "Display Init" method VI, which is part of the "Consumer" class. It associates an instantiated display class object with the tuple stream and the graphical object. All inputs have a red triangle, indicating a type cast. A black triangle in the upper left corner of an entity indicates a custom type definition, which is causing the topmost red triangle to appear for the "Display Init" subVI. In the two other cases the red triangles indicate a type cast to a more generic class: The "Number" child class is casted to the "Consumer" class, and the "Digital" class reference is casted to a generic control reference.

The block diagram in Figure 55 shows the operation of the "Do" method VI, the message handler of the "Tuple" class. Any actor receiving this message will call this VI. The actor must be a descendant of the "Operator" class, as it is to this class the actor will be cast. All "Operator" descendants must have a "Process Tuple" VI which will handle the received tuple.

Note that the "Actor out" output has a red triangle. That is because the casted "Operator" object will be casted back to a generic actor.

The "Process Tuple" block diagram in Figure 56 is called for each tuple for the number box. It performs two operations: 1) Casting the stored reference to the right type. The VI has a dummy number box which serves as the type target for

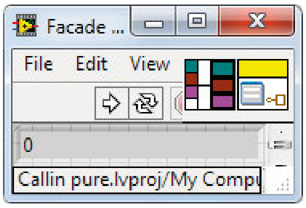


Figure 53: The front panel of a façade VI that has a number text box embedded. The connector pane is predefined in the XControl template.

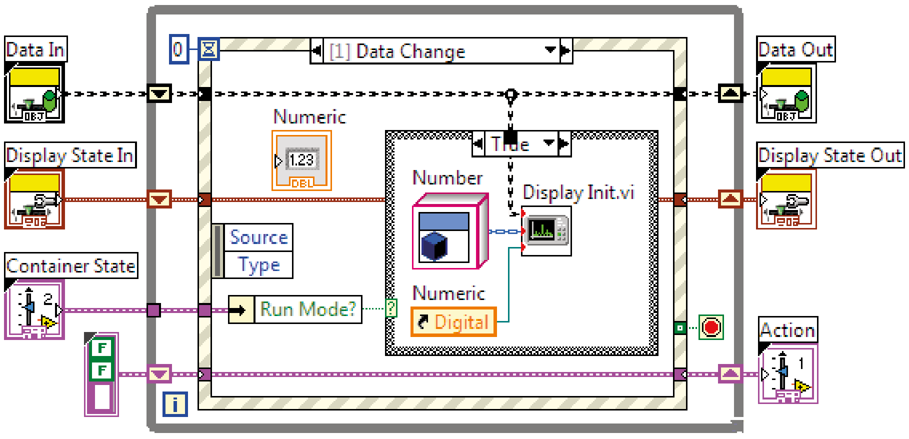


Figure 54: The block panel of the façade VI. The event structure and all objects outside of it are part of the XControl template, and must not be altered.

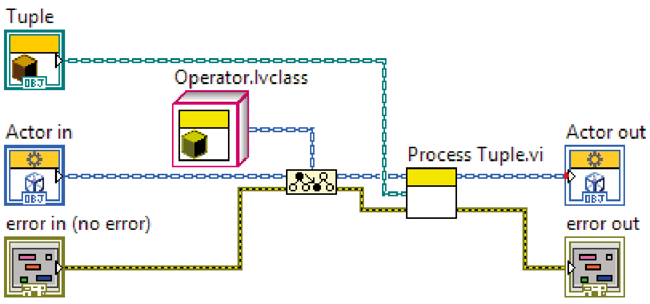


Figure 55: The "Do" method VI of the "Tuple" class. All it does is call the "Process Tuple" VI of the recipient actor, casted to an "Operator" class.

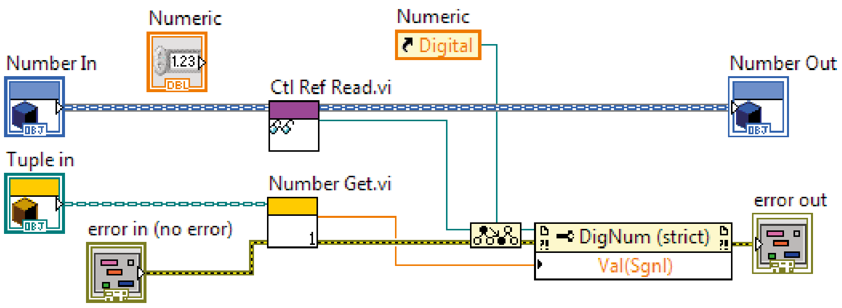


Figure 56: This is the "Process Tuple" block diagram for the "Number" class. It will be called each time a tuple message is received.

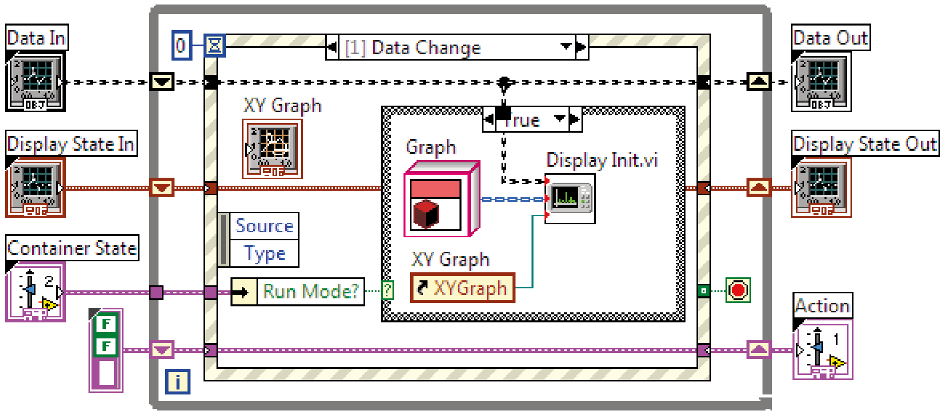


Figure 57: The XControl for a scatter or line plot. Note that this XControl is structurally identical to the one in Figure 54.

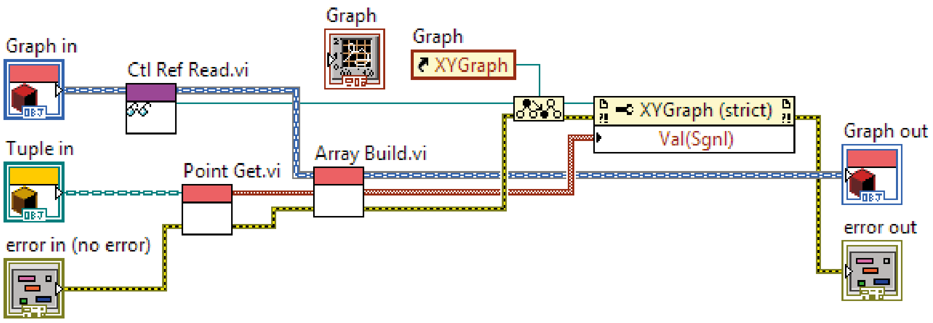


Figure 58: The "Process Tuple" VI for a scatter/line plot.

casting the control reference. This is not necessary, but increases the efficiency of the subsequent method call. 2) The tuple member VI "Number Get" converts the first element of the tuple to a double precision number. Using the casted control reference, this number is presented in the number box. A slight improvement can be made by moving the reference type cast to the "Startup" method VI, which runs during initialization.

The example shown in Figure 57 and Figure 58 is slightly more complicated, because it requires data to be preserved between calls.

The main difference in the "Process Tuple" block diagram shown in Figure 56, compared to the one Figure 58, is the handling of data. The "Point Get" VI handles the type conversion: it reads the first two elements of the tuple and returns them as a two-dimensional point. The "Array Build" VI reads data stored in the actor, adds the point to the array, and then writes data back to the actor. An actor may store any data, and by including the data in the actor returned from the "Do" method VI, its proper storage is ensured.

A.2 Enqueuer transfer

The RUN QUERY subVI – and any other producer or operator that sends tuple messages to a subsequent actor – cannot both send a queue reference and wait to receive an item from that queue; the subVI must have finished running before it can return any items. The solution is to start an asynchronous VI to do the waiting, and then returning the reference.

The block diagram for the RUN QUERY subVI, shown in Figure 59, consists basically of three parts: 1) starting the asynchronous VI (the function with the sideways triangle), 2) waiting for the VI to start running (the while loop), and 3) returning a tuple stream object with the queue name.

The subVI is set to preallocated clone reentrant execution, which means it has a unique clone name. This name is used to uniquely identify the queue as well. "Run Query Loop.vi" is the name of the VI that will be started independently. A reference is opened to a clone of the VI which is then set to run.

The while loop will check every tenth of a second if the VI is running, and exit when it does. This is needed because of what is possibly a bug in LabVIEW: If a VI finishes running too quickly after it has started an asynchronous VI, then that VI will never run. The while loop will ensure that it is running, before leaving the VI.

Afterwards, the reference is closed, and any error that may have appeared (through the yellow and black wire) is returned with the tuple stream object.

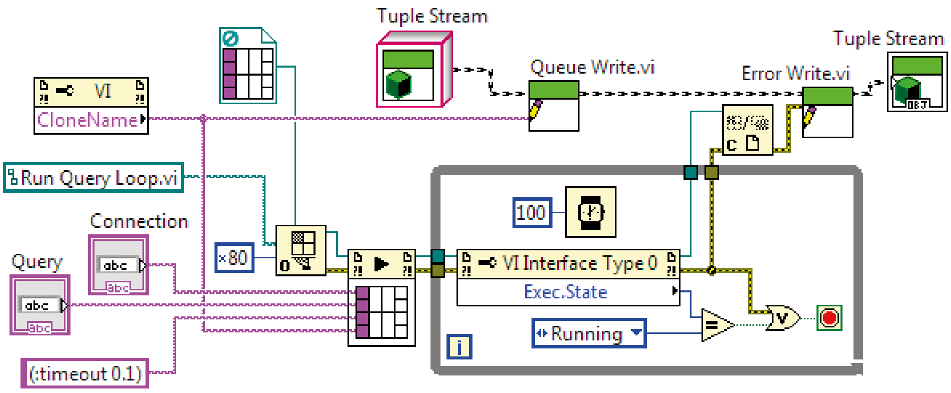


Figure 59: Inside the RUN QUERY subVI.

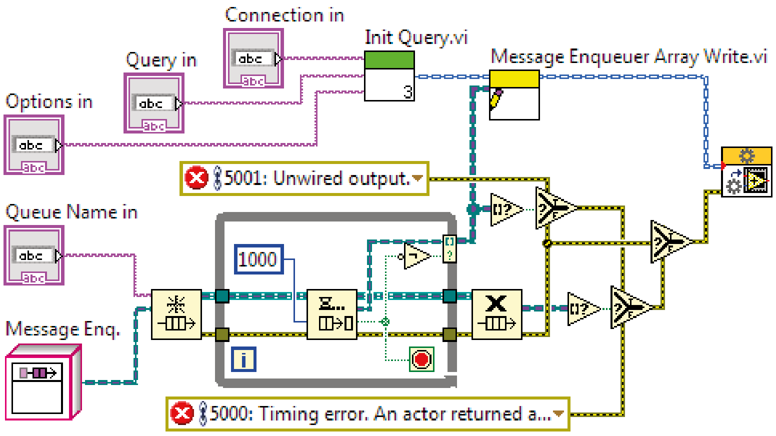


Figure 60: Enqueuer reception inside the "Run Query Loop.vi" VI.

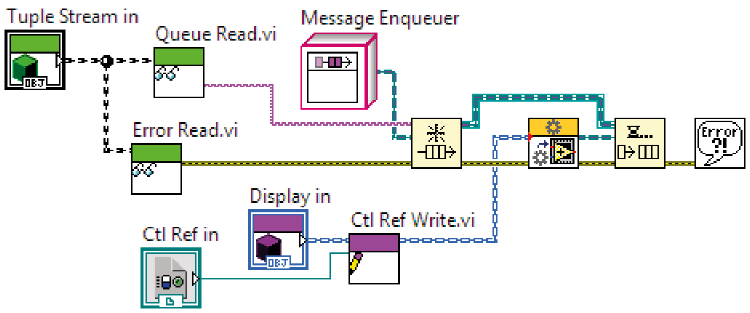


Figure 61: Sending the enqueuer inside the "Display Init" VI.

There are three queue operations carried out by the "Run Query Loop.vi" block diagram in Figure 60: 1) Before going into the while loop, the queue is obtained (it is created if it does not exist). 2) Inside the loop, the dequeue function will wait for a message enqueuer to arrive for 1000 milliseconds. If an enqueuer arrives, it will loop over and wait for a new one, otherwise it will time out and exit. 3) Afterwards, the queue is no longer needed and will be deallocated.

The enqueuers that were retrieved from the queue are packed in an array and written to the query actor object, which is then launched.

This way of enqueuer retrieval is necessary, because the number of enqueuers is unknown. This also means that it is possible, however unlikely, that initialization can fail, because of the timeout. A longer timeout means a smaller chance of failure, but increased waiting time for the user.

There are three possible types of errors, of differing precedence. A queue operation error has highest priority, because if an operation fails, everything else fails as well. It is possible that the queue receives an item after it has timed out, causing the timing error. The third and most likely error is an unwired output, causing the enqueuer array to be empty.

A fourth possible error is a subsequent actor trying to send an enqueuer to a queue that has been closed, but that is handled elsewhere.

It is certainly possible to start the actor first, and then have the actor wait for the enqueuers. This would eliminate much of the code needed above. However, it is more desirable to have all enqueuers in place before starting the actor; it makes the design more robust to changes, and makes error handling easier.

"Display Init" (Figure 61) is the subVI that is used inside a consumer to retrieve the tuple stream object and initialize the consumer. "Display in" is the child class that handles visualization, "Ctl Ref in" is a reference to the display object that receives the visualization output. The VI obtains the queue for the queue name retrieved from the tuple stream object. It then starts the consumer actor and puts the returned enqueuer in the queue. ■

Appendix B – Server building blocks

The SVALI server has several components that may be interesting for a closer look.

B.1 The *fixstream()* wrapper handler

Continuing the example with the digital antenna prototype, a typical query will be based on this query template:

```
select counter, array from number counter, carray array
 where (counter, array) in fixstream(vi(“radio.vi”), “u2,ci2[366]”);
```

This query returns a stream of tuples, where the first value is a number and the second is an array of complex numbers (data type *carray* in SVALI). They are converted from a two-byte unsigned integer package counter and an array of 366 complex, two-byte, signed integers, as defined in the second parameter of the *fixstream()* function. This parameter also declares the return signature of the function. The default declaration of *fixstream()* is:

```
stream of object
```

When added to the query above it automatically gets the type:

```
stream of (number, carray)
```

In effect, this adds static type checking to a dynamic context.

There does not exist any semantic coupling between the wrapper function output and the wrapper handler input – just as there is none between data streams and wrappers – meaning that it is up to the user to verify that the type string actually matches the contents of the byte array. This is a problem with all wrapper programming. A possible solution is to add some form of schema propagation. By associating a data stream with a schema, it becomes possible to automate type checking of the *fixstream()* function. At the time of writing, there does not exist any mechanism for defining schemas.

Wrapper handler operation is divided into two threads, in order to do as much work in parallel as possible. The child thread performs the two tasks that can be done outside of the system: waiting for a data package to arrive, and converting the data to SVALI data types. The execution flow is illustrated in Figure 62.

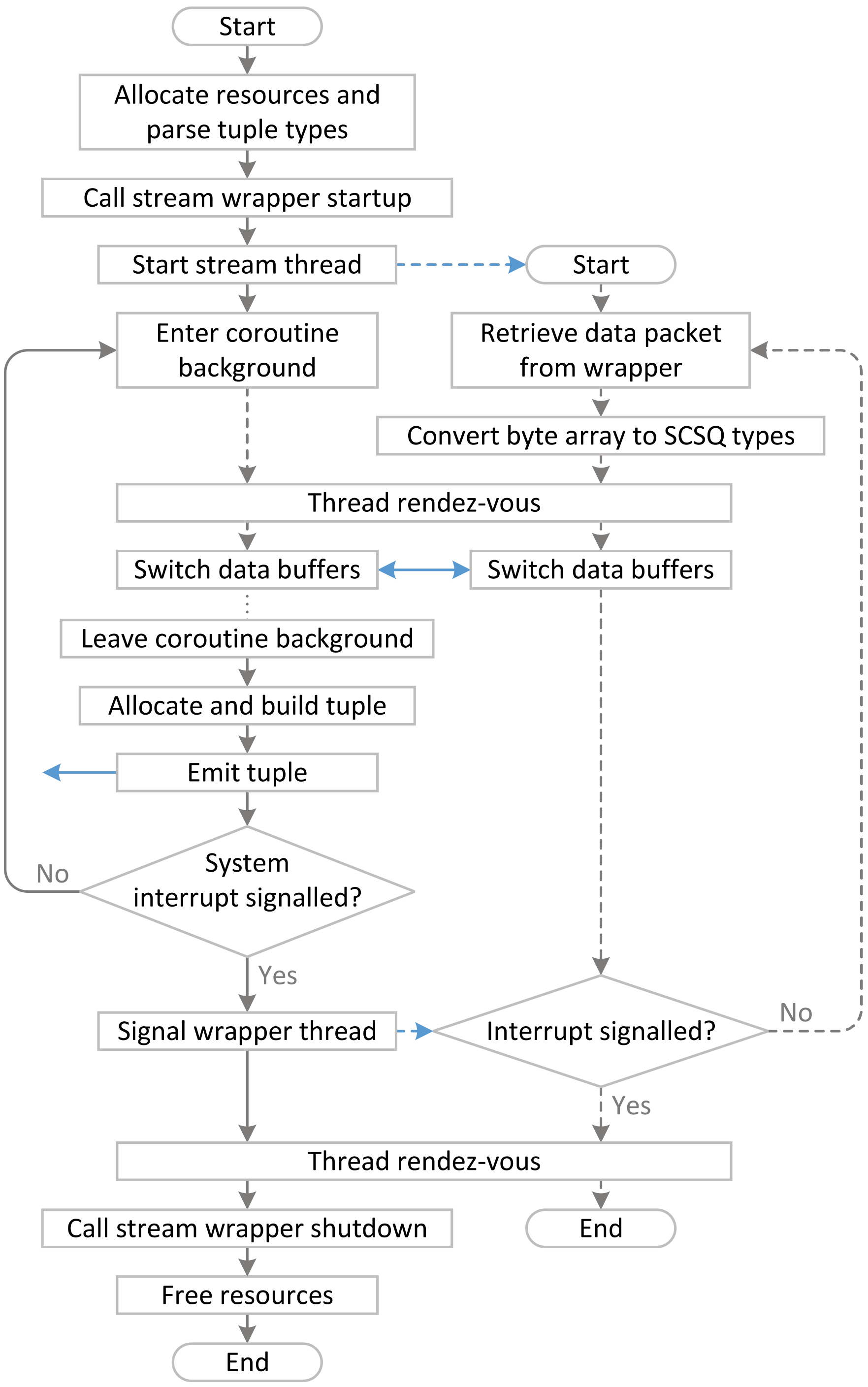

Start
Allocate resources and parse tuple types
Call stream wrapper startup
Start stream thread
Start
Enter coroutine background
Retrieve data packet from wrapper
Convert byte array to SCSQ types
Thread rendez-vous
Switch data buffers
Switch data buffers
Leave coroutine background
Allocate and build tuple
Emit tuple
System interrupt signalled?
No
Yes
Signal wrapper thread
Interrupt signalled?
No
Yes
Thread rendez-vous
Call stream wrapper shutdown
End
Free resources
End

Figure 62: The execution flowchart for the *fixstream()* wrapper handler.

The parent thread will most likely run in a coroutine, since the query is expected to run on a server. It is possible though to run the query in a dedicated process, and in that case the steps for entering background execution and returning to the foreground will not have any effect. When running in a coroutine, it will wait for data from the child thread in the background, allowing other queries to run in the meantime. Allocating and building a tuple must be done in the foreground. The child thread meanwhile will continue to retrieve the next data package.

B.2 Interfacing LabVIEW with embeddable components

SVALI has a C application programming interface (API) through which a programmer can send commands and retrieve query results. Correspondingly, LabVIEW supports calling external functions in C[1] by loading a DLL/shared object (Figure 63), the functions of which can be called through a LabVIEW function named *call library function node* (Figure 64). Using the node, functions become individual entities in a block diagram.

The node in Figure 64 corresponds directly to a C function in the client API, which is loaded with the DLL/shared object:

```
int32_t __declspec(dllexport) lv_nextrow(a_scan scan, a_tuple tpl)
{
  return a_nextrow_basic(scan, tpl, TRUE);
}
```

The function *a_nextrow_basic()* is in turn part of the SVALI peer API that supports communication between peers, both clients and servers. *a_scan* can be either a scan object or a remote scan object.

LabVIEW does not check the validity of any function calls. Instead it is up to the programmer to make sure that function compatibility is maintained. For each function call, a list of LabVIEW variables are matched against the function parameters. In the case of a type mismatch, it is up to the external function to handle type conversion. LabVIEW provides a library of C functions for creating and handling native types.

The main purpose of an application programming interface is to provide access to a different programming environment. The second most important aspect is logical independence. An API is expected to behave in a certain way that is pre-

1 Other languages can be used as well, as long as they support functions with C headers.

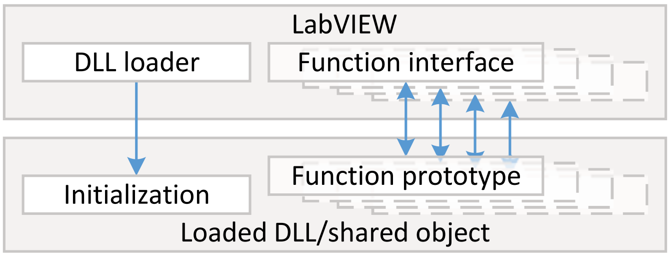


Figure 63: A schematic overview of external function calls in LabVIEW.

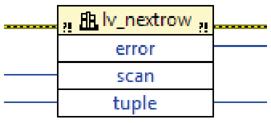


Figure 64: A library function node for retrieving a tuple from a scan.

dictable and conformant to the environment in which it is used. This behaviour need not emulate the underlying system, and it must not change with updates of said system.

B.3 Coroutines

SVALI coroutines are asymmetric and stackful, but not first-class [52]. They cannot be transferred to a different processing context from which they were started, nor can they be saved to a file.

An asymmetric coroutine must always yield a value at some point. This is the expected behaviour with function calls, and is a natural match with query execution.

A stackful coroutine can yield a value at any point in its execution, whereas stackless ones can yield values only in the main coroutine function. Recursive coroutines must be stackful, to mention an example, but the handling of queries does in itself not require stackfulness.

The basic functionality of a coroutine is that of a child process running alongside the main process, sharing the same resources. Both processes cannot run at the same time, because that will cause conflicts. System resources must only be accessed by one process at a time. This is a form of cooperative multitasking, where each process at some point allows the other process to run.

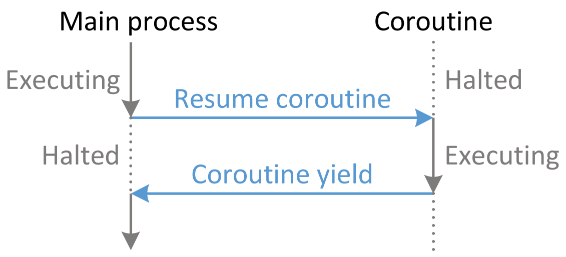


Figure 65: Execution flow of a basic coroutine.

All asymmetric coroutines follow the flowchart in Figure 65 in one way or another. Upon resuming a coroutine, the main process may pass data to it, just as with a regular function call.

For streaming queries, this basic operation would be sufficient for continuous execution and returning tuples in a timely manner, if they were only contingent on processing power. Letting all running queries yield tuples through a coroutine, one after another, would not impede the operation of any single query. However, continuous queries also depend on operations outside of the system, the main concern of which is the reading of data from external data streams. A coroutine that waits for data to arrive on a stream will block the operation of all other processes. It is therefore important that the coroutine can wait for data while relinquishing control of the system. A coroutine that has system control is said to run in the *foreground*. A coroutine that has relinquished system control but is still running is said to have entered the *background*. As illustrated in Figure 66, a coroutine that wants to enter the foreground again has to wait for the main process to call it.

A coroutine that is waiting, either for data input or to return to foreground execution, will be suspended by the operating system and thus not consume any processing power.

A coroutine that enters the background returns the symbol **BUSY** to the main process, to indicate its changed state, shown in Figure 66. For background execution, coroutines must be stackful, because entering the background is in effect equivalent to a yield, but it can happen at any call depth during the coroutine's execution. When about to leave the background, the coroutine can signal the main process. This allows the main process to wait for one or many coroutines to become available for resuming operation.

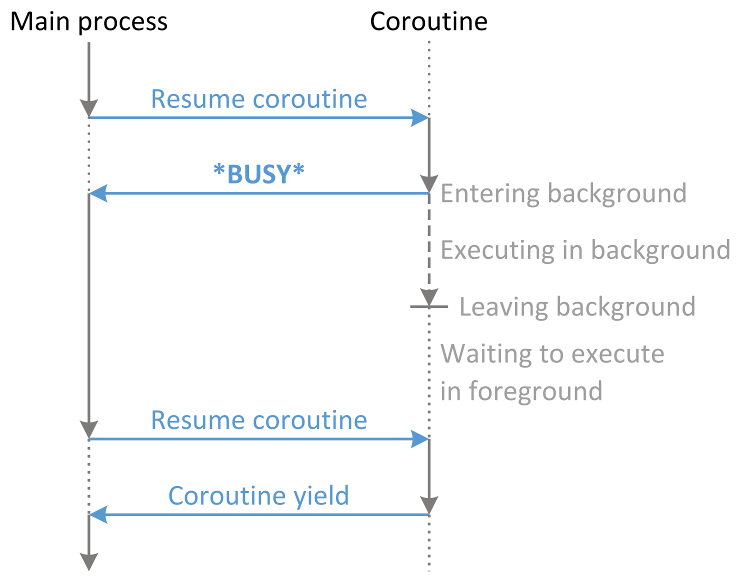


Figure 66: Execution flow of an extended coroutine supporting background execution.

Coroutines are the basic building blocks of the server. They offer the ability to switch out queries and to run them in the background. Each query runs in its own coroutine, and the main process of the server calls each coroutine in turn. Entering the background lets a query continue to run without blocking the server, and thus allowing other queries to run.

When a coroutine enters background execution, its state becomes unknown, because the coroutine's state is stored in its system variables, and those are not available until the next time the coroutine enters the foreground, including the query that it is running. One consequence is that a background coroutine cannot be stopped or deleted. Any process trying to do so will halt until the coroutine leaves the background. A coroutine does share some variables with its calling process, and coroutines running in the background can be flagged, causing them to terminate themselves upon reentering the foreground.

Coroutines running in the background must not access system resources, e.g. they must not allocate or manipulate variables. Some operations that do not alter the state of the system – like reading the value of a variable – are still possible, but unsafe. They can still access resources that exist independently of the system. A stream source is typically handled by the coroutine that accesses it, which means that the state of the coroutine becomes irrelevant for communicating with the source.

B.4 Scans

A scan is, at the very least, an entity that provides procedural access to the results of a query execution. SVALI scans do a bit more, as they encapsulate queries and provide abstractions for their operation. From a user point of view, the two main points are: 1) an interface for handling the query in a way that feels logical and is generic across different applications, and 2) fallback routines that guarantee the graceful shutdown of a query, should it be interrupted or fail.

At initialization, a query is assigned a coroutine and a buffer. The coroutine allows a query to wait for data, call functions outside of the system efficiently, and return data opportunistically through stateful operations, the last of which it does by sending data to the buffer.

A scan is the meeting point between the external programming language used for running a query, and the internal goings-on of the query engine, either a server or a stand-alone program. As such, scans provide logical independence between the application layer and the system kernel.

All running servers have a single program loop that handles all requests. Figure 67 shows the interaction between the server loop and the coroutine for a scan that has been sent to the server. The coroutine does not return data to the loop. Instead the data is put in the buffer, which is shared by the scan object and the coroutine.

The point of using a buffer in a scan is to transmit data efficiently over a network and between processes, and there are two settings for the buffer:

- A fixed buffer size. The coroutine thread will not yield until the buffer is filled.
- A fixed buffer size with timeout. In order to increase responsiveness to scan operation, each time the buffer has a tuple added, a timer is checked to see if a timeout has been reached. If it has, the coroutine will yield and the buffer will be stored in the scan object. A scan will not time out if it is empty, but wait for at least one tuple.

On top of these two options, a scan can have a *rate*, which is a rule or set of rules for how responsive a scan should be depending on the output rate of the tuples. This rate may change over the course of query operation.

Tuples are retrieved from a scan one at a time, regardless of buffer size. Only when the buffer is empty is the coroutine called again and query operation recommenced.

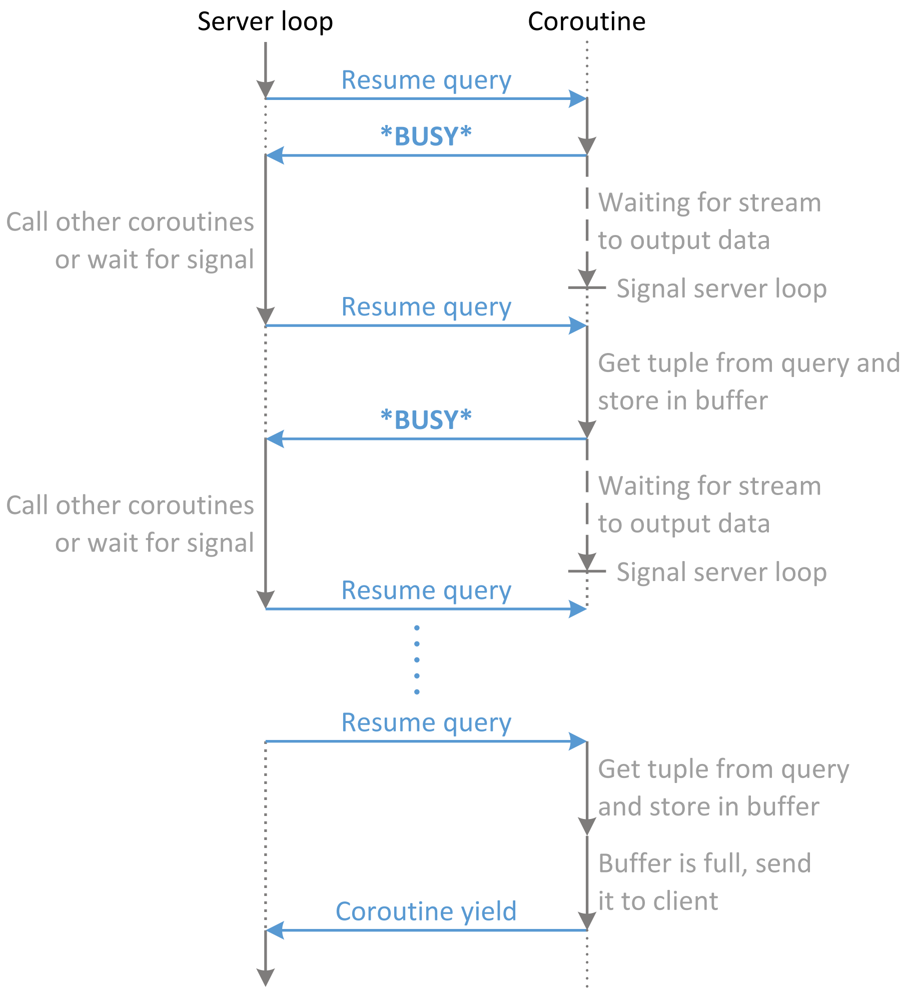

Server loop
Coroutine
Resume query
BUSY
Waiting for stream to output data
Call other coroutines or wait for signal
Signal server loop
Resume query
Get tuple from query and store in buffer
BUSY
Waiting for stream to output data
Call other coroutines or wait for signal
Signal server loop
Resume query
Resume query
Get tuple from query and store in buffer
Buffer is full, send it to client
Coroutine yield

Figure 67: Coroutine operation inside a scan in the server structure.

Remote scans

A remote scan is a handle to a scan residing on a server. Each time a function call is made for a remote scan, a request is made to the scan on the server. Every SVALI process has a process global hash table [44] that stores server-side scans. A server-side scan has the same life span as the coroutine it contains; once the coroutine has finished running, the scan is deleted, along with all components.

A remote scan provides transparency. For all intents and purposes, it is a scan, but its physical location is not in the process with which it is associated. Its location could be unknown to the user, and even switch places during its run time[1].

B.5 Server structure

A DSMS client – it can be written in LabVIEW, Java, C/C++, or whichever language one chooses – will likely support several concurrent calls to the server, or there can be several clients calling a single server. This posed a problem, because Amos II and its siblings did not originally support independently running queries in a single process; a query running in a server process would prevent other queries from running. This becomes a problem when a streaming query waits for input from a stream source, because the query will block while it is waiting. This is where coroutines with support for background execution come to the rescue in SVALI, because they will allow a query to wait without blocking the system. In effect, there will still only be one query running at any moment, but it runs within a flexible, multiplexing server structure, where queries switch themselves out of the system at opportune moments.

Usually, waiting for data input means waiting for a data stream source to send data, but it could be user input as well. In other cases queries are expected to read and process data at certain intervals, and during the time not spent doing that, the query should not obstruct other queries from running.

When waiting for data stream input, a query will automatically enter the background. It cannot return to the foreground until permitted to do so by the server loop, as described in Figure 66. A query that does not read data from an external stream may not enter the background unless programmed to, and will only relinquish control of the server upon returning tuples to the client. This may not be a problem if the query returns tuples faster than other running queries, otherwise it will slow down the other queries to its own pace.

Each running server will always have one special type of coroutine called a *dispatcher*, and at server startup it is the only thing that is running in the server loop. The dispatcher listens to incoming calls and runs server functions in response to those calls. It does not discern between types of calls, as each call by itself defines the server function to execute. In order to run queries in conjunction with each other, they cannot be executed immediately when called. Instead the dispatcher will add them to a list with all the queries running on the server, including the dispatcher itself. The server loop calls each coroutine in turn. This is necessary in order to maintain the multiplexing nature of the server.

1 Provided it is a first class object.

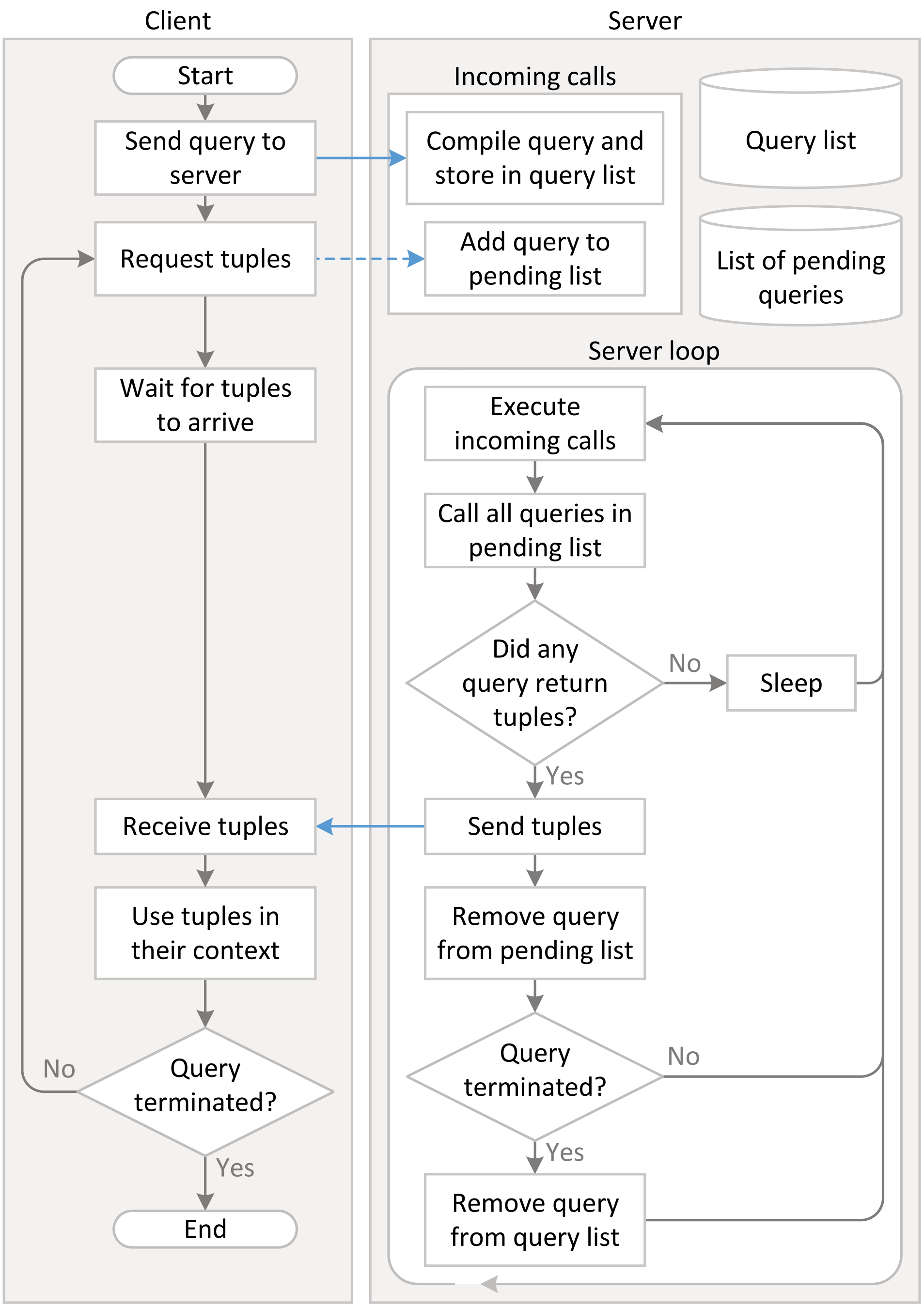

Client
Server
Start
Send query to server
Request tuples
Wait for tuples to arrive
Receive tuples
Use tuples in their context
Query terminated?
No
Yes
End
Incoming calls
Compile query and store in query list
Add query to pending list
Query list
List of pending queries
Server loop
Execute incoming calls
Call all queries in pending list
Did any query return tuples?
No
Sleep
Yes
Send tuples
Remove query from pending list
Query terminated?
No
Yes
Remove query from query list

Figure 68: A client running in a separate process interacting with a server, when running a query.

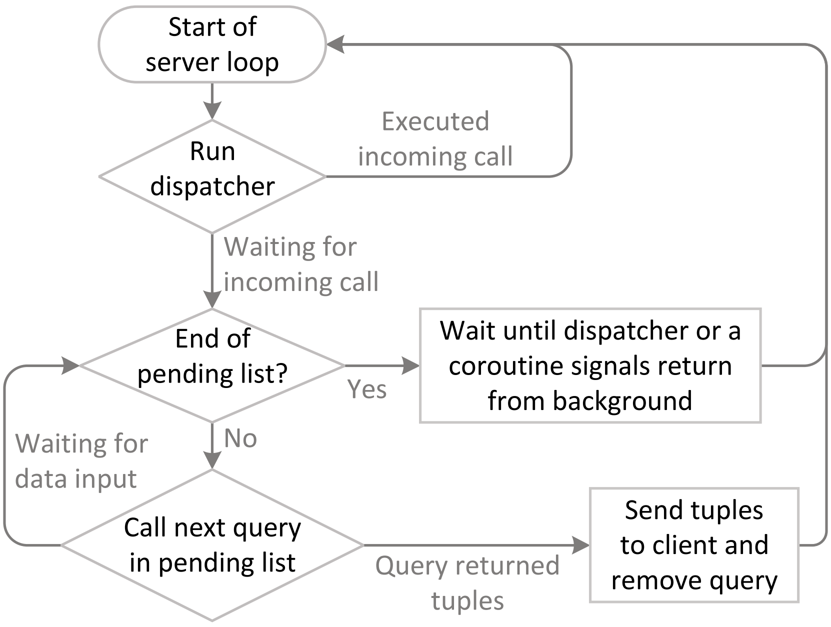

Start of server loop
Run dispatcher
Executed incoming call
Waiting for incoming call
End of pending list?
Yes
Wait until dispatcher or a coroutine signals return from background
No
Waiting for data input
Call next query in pending list
Query returned tuples
Send tuples to client and remove query

Figure 69: Calls to dispatcher and queries in the server loop.

The flowchart in Figure 68 shows the client-server operation of a single query coroutine in conjunction with the dispatcher. Often, the dispatcher will wait for incoming calls while all queries currently running are likewise waiting for incoming data. When that happens, the server loop will become suspended and wait for any coroutine to leave the background and signal the loop to continue running.

The server can also – on top of the dispatcher and queries – run daemon functions in stand-alone coroutines that are not wrapped by a scan. These coroutines are called by the server, but their operational control is handled either by a programmer or by script automation. They are useful for handling scheduled tasks and similar.

There are two query lists shown in Figure 68: the query list that contains all the queries sent to the server, and a list with pending queries. The query list is a mix of scans – those are the queries – and coroutines for the dispatcher and any daemon functions. A pending query is a scan that has received a request for more tuples, and the list stores the coroutines of those scans.

A scan will collect several tuples at a time, to increase efficiency, meaning that a query will wait for data stream input several times without returning any results.

The dispatcher and all queries will be called at the same time, because it is likely that most, if not all, will wait in the background for input to arrive; a coroutine will only start waiting for input when called, so it makes sense to call

all coroutines as early as possible. In the case of the dispatcher, it will wait for input from any connected client; a remote scan will not call its server counterpart directly, but through the dispatcher, in order to maintain the responsiveness of the server.

All operating systems have some form of condition variables, which are used for suspending the execution of threads, and to signal their state to a different thread. The dispatcher and all queries each run in their own threads, and each has two condition variables for signalling their execution state: one for suspending operation, and one for signalling background execution. In the case when all queries plus dispatcher are waiting for input, the server will wait on each variable simultaneously. Figure 69 shows the order of operations for that situation to appear.

At the moment the dispatcher or a query receives stream input, it will want to return to the foreground by signalling the server loop to continue. That thread will in turn start to wait on another condition variable, which the server loop thread will signal when it in turn is ready to wait.

Only one query can run in the foreground at a time, and it will do so at its own leisure, until it returns tuples or reenters the background. Only then can a different query continue its operation. This cooperative multitasking allows queries to switch execution states in a fairly deterministic manner, and does not raise all the timing issues that are related to preemptive multitasking. It does however mean that much of the operational time of queries is serial, and for this reason queries running on a single server do not scale well; the operational time required by a server is approximately the sum of the operational time of all queries running on that server. Instead, for highly scalable processing the parallelization primitives provided by SCSQ [95][96] have to be used. ■

Appendix C – Tangential issues

This appendix describes some things that are related to topics that were presented in this Thesis, and which may be interesting, but nonetheless did not merit inclusion in the main text.

C.1 More issues with data flow programming

There have been a few mentions of when data stream management in a data flow programming environment can fail, but unfortunately there are more issues to consider.

Solving wire branches and wire merges

A user may want to send the output from one function node to more than one input of subsequent nodes. How this is accomplished for a particular data flow framework is up to the developers of that platform. Branching may even be prohibited. The data flow programming paradigm is not affected by whichever solution is chosen for wire branches. For example, *Max*[1] from Cycling 74 has implicit wire branching (the left diagram in Figure 70), while StreamBase (the right diagram) only allows explicit wire branching using a splitting function node. It should be noted that of the two, only StreamBase is intended for handling data streams, as Max relies on a global tick counter for program operation. It makes sense to use a splitting node for data streams, because it is convenient to be able to attach a broadcasting function to the node, offering options for filtering the different output streams.

LabVIEW does not have any way of handling wire branches. Likewise, it is impossible to tell whether a subVI output is wired or not. There is however a way to circumvent this problem in this case, thanks to the queue used to transfer an actor message buffer. By setting a timeout when waiting for a queue to receive a message buffer, any amount of message buffers can be handled and consequently any type of wire branch can be handled as well.

1 http://cycling74.com/products/max

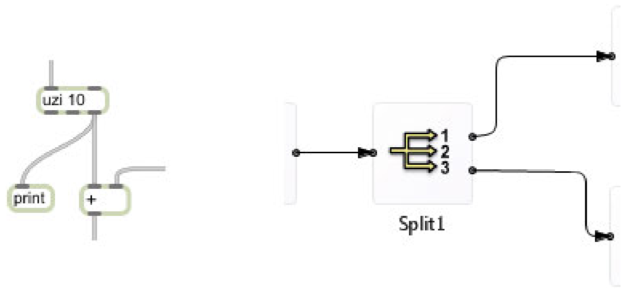


Figure 70: To the left, branches in Max are handled automatically (top-to-bottom). StreamBase has a specific function node for creating wire branches (left-to-right).

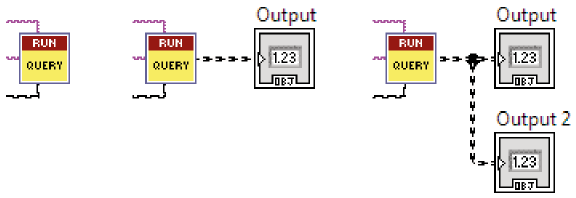


Figure 71: Three different connection wirings that need to be discerned.

If the three block diagrams in Figure 71 were ordinary LabVIEW programs, the wiring would be impossible to discern for each diagram from within a program. For the purpose of data streaming, the leftmost diagram should return an error, because it does not have a consumer that receives tuples. The other two diagrams should be fine, but in order for the data flow framework to function properly, they need to be distinguishable. If only explicit wire branching were allowed through the use of a function node, the problem would be solved. It does however not stop users from drawing wire branches in LabVIEW anyway, causing the program to fail.

The details of how this is solved in VisDM was covered in Appendix A.2, "Enqueuer transfer" on page 90.

In contrast, wire merges should *not* be allowed. The rules of data flow programming do not forbid wire merges, but there are situations when they are not feasible. LabVIEW does not have support for merging wires, as this would conflict with the behaviour of its programming model. Looking at the diagram in Figure 72, the two RUN QUERY nodes would send their queue references simultaneously to the "Output" XControl, which becomes impossible to resolve.

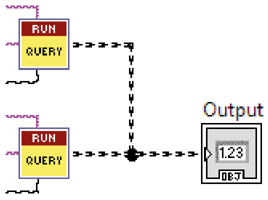


Figure 72: Merging function node output. This works fine in a data flow setting in general, but not when handling data streams. This way of merging wires is not possible in LabVIEW.

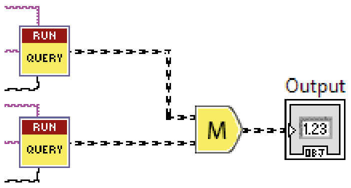


Figure 73: The correct way of handling data stream merges, using a merge function to ensure the validity of the merged stream.

With other data flow programming platforms (for example Max) wire merges are supported, without the need of merge operators. In these cases, the elements of the merged data flow retain their operational context. Elements of a data stream, in contrast, lose their context when mixed with other data flows/streams, as would be the case in Figure 72. A data stream can generally not be chopped into bits, as would happen with a simple wire merge that will interleave the streams. And even if it can be divided, the order of the merged tuples is usually important as well. First of all, there is the question whether tuples should be chained together, or merged physically into new entities. Merging data streams always requires a merging function, usually a join operation [50]. Figure 73 shows an example of how this may appear in a block diagram.

A merge operator can provide monitoring and error handling services, which can be very useful, particularly if the streams have widely different transfer rates.

Interestingly, actors do not prohibit wire merges. Merges merely translate to the receiving actor collecting messages from more than one transmitting actor. The messages may need additional data about their point of origin, which is easy to add, or each actor may have its own set of message types.

Issues with LabVIEW data flow programming

A wire in LabVIEW does not represent a data flow, it represents a data *transfer.* In a data flow programming language, data flows are what drives the execution. If execution is not driven by data flows, it is questionable whether it is a data flow programming language.

Case in point, think of how a data flow programming language would handle the press of a button. In Max, the pressing of a button sends an event over a wire representing an event flow. It is a data flow that may only produce a single value over an application run time, but it is a data flow nonetheless. In Max, events are called "bangs", and are identified by the wire through which they are transmitted. Bangs do not contain any information by themselves, other than indicating an event. More complex events can be built by wiring data, as can be seen in the example shown in Figure 74.

In LabVIEW, the pressing of a button will not cause an event unless an event structure is waiting for the event to occur. And even then, it will only trigger once unless the event structure resides in a loop structure. If a button is pressed before the event structure has had time to start waiting for the event, it will never trigger for that event.

Figure 75 shows the basic layout for handling events; an event structure residing inside a loop structure. When program execution reaches the event structure, that particular thread will wait for the button to be pressed. When pressed, any code residing inside that particular event tab will execute. Afterwards, any functions wired to the structure will execute, and after that the loop structure will start a new iteration and a new event will be waited for. This shows very clearly the procedural nature of LabVIEW.

A control structure need not run counter to data flow paradigms. The LabVIEW case structure creates a branch of execution, where the data transfer passes through one out of two or more diagrams. The number of inputs and outputs will be equal for all diagrams, because they are in effect stacked, only one of them being shown at a time. In a data flow program, all diagrams would be visible at the same time, and a switch [20][38], or something similar, would redirect the data flow through the proper diagram instead.

Figure 76 shows a very simple case structure with two options. The value of the "Option" input controls which control tab to run.

The code to run for a particular case must reside completely within the structure frame. The case structure is functionally equivalent to a data flow merge [20] [38] function. An example of how such a function may look in LabVIEW, were it to exist, is shown in Figure 77.

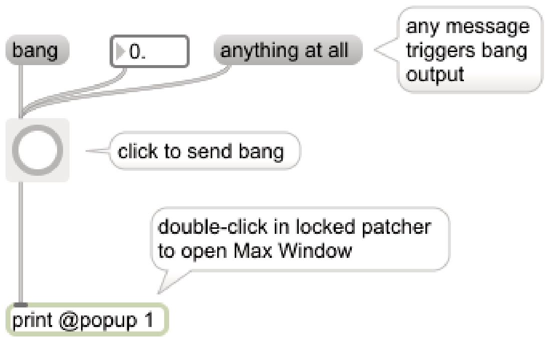

bang
0.
anything at all
any message triggers bang output
click to send bang
double-click in locked patcher to open Max Window
print @popup 1

Figure 74: Button example from the Max help.

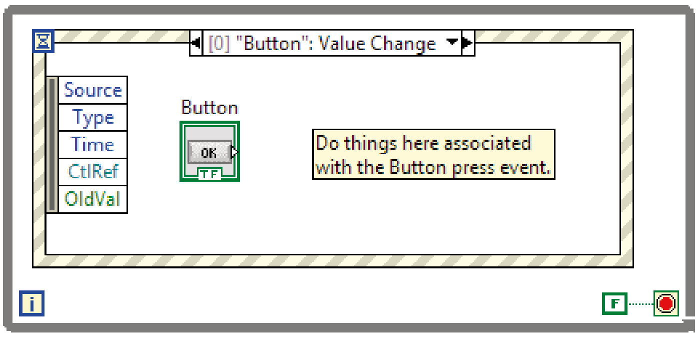

[0] "Button": Value Change
Source
Type
Time
CtlRef
OldVal
Button
OK
TF
Do things here associated with the Button press event.
i
F

Figure 75: Handling a LabVIEW button press event.

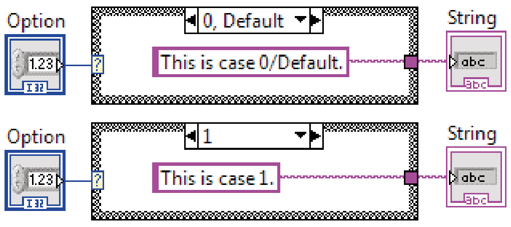


Figure 76: A simple LabVIEW case structure setup. The two parts of the block diagram show the same program, but with different cases.

Note that the execution of different cases can be quite different when using a merge, depending on the data flow model used. When using a demand-driven [20] model, the correct case will be evaluated only when an option is set and incorrect cases will not be evaluated. This operation is equivalent to how the case structure operates in LabVIEW. In a data-driven model however, all cases may

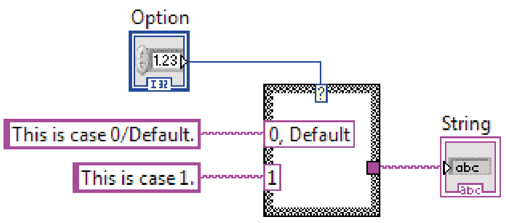


Figure 77: A data flow conditional merge in LabVIEW pseudo-“code”, equivalent to the case structure in Figure 76.

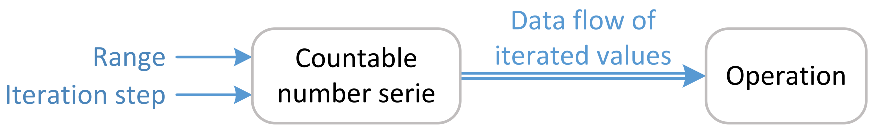


Figure 78: A function node that produces a data flow of values for certain parameters.

be evaluated before the option is set. Once the correct option is set, the corresponding result will be propagated through the diagram, and the other results will be ignored. This will cause extra resources to be spent evaluating operations that will be discarded, but on the other hand the data flow system may respond faster.

When actor-based data flow programming fails

Problems with actor based data flow programming can be summarized with one word: congestion. Actors must be able to process data at least as fast as it arrives. This may not always be the case, and it may be hard to predict when such a situation appears. Temporary congestion is usually not a problem, thanks to message buffering, but issues appear for prolonged durations.

In lieu of a loop construct, a data flow may have a function node that produces a series of numbers within a certain range (Figure 78). Producing these numbers is computationally cheap, and the data flow will very quickly fill up with messages containing the numbers. Consequently, there must be a mechanism for regulating the flow of numbers. In Max, this is accomplished with a global tick counter. In VEE, iteration functions must have a connected sequence trigger input.

Iteration can be implemented by using a feedback loop (see Appendix C.2, “Feedback loops using actors” on page 111), when such are available. Feedback-based iteration can eliminate congestion issues without relying on triggers.

C.2 Feedback loops using actors

A visual data flow programming language may or may not support feedback loops. Including feedback support does not violate any rules for data flow programming, instead it depends on the intended application of the platform. Feedback loops are interesting because they can be used both for iteration and recursion.

StreamBase is intended for stream processing and aggregation, and therefore has no need for feedback. Max is more generalized, and does have feedback support. One reason feedback works well in Max is because of the global tick counter, causing all output inter-operations to become well defined and trivial. Figure 79 shows a Max program example which has a feedback loop being fed by an iterator.

The feedback source is connected after the result output widget. This means that the initial value for the feedback is the content currently contained in the output display. LabVIEW indicators do not have any wired output, and thus feedback behaves differently, as shown in Figure 80.

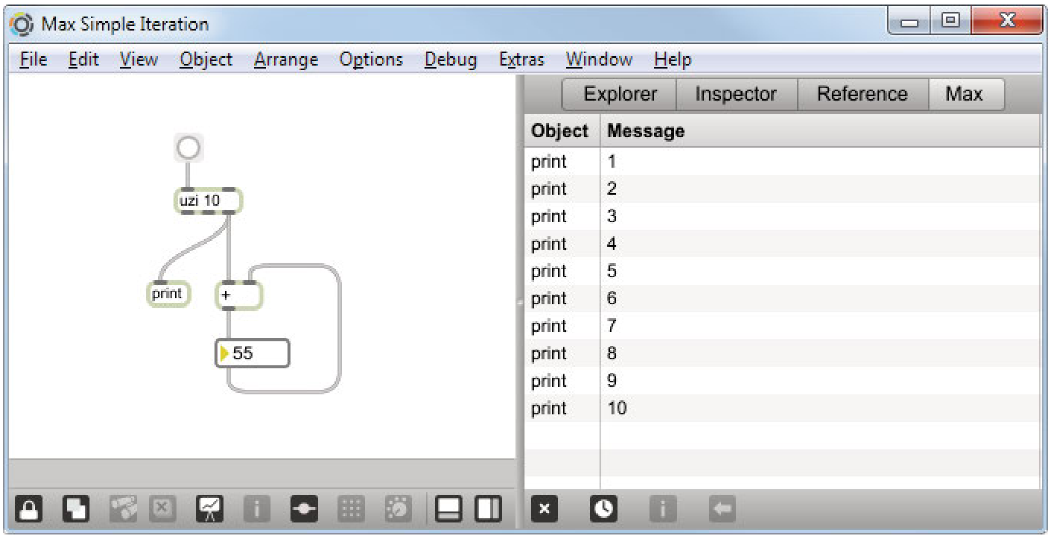
LabVIEW supports feedback inside a loop structure context by using the feedback node, which simply delays the value transfer for one iteration. LabVIEW does not support cyclic diagrams as Max does.

However, when moving to a solution using actors, the feedback node will not work. The problem with the diagram in Figure 81 is that the wire is not a data flow, but merely facilitates the setup of the flow. The feedback node will allow the design of a program with feedback, but the design will be incorrect and cause a run-time error.

Actors do not directly support the construction of feedback loops, because at some stage of construction an actor must be created before an output enqueuer can be assigned to it. In Figure 82, this happens in the rightmost feedback node. Its actor is created first, then later the actor for the leftmost feedback node is created, but the input enqueuer from the leftmost actor becomes the output enqueuer for the rightmost actor. This is easy to solve manually though: The rightmost actor receives a dummy enqueuer, which is then replaced after creating the second actor.

The feedback solution in Figure 82 circumvents the restriction of cyclic wiring by introducing a feedback node that has two distinct functions: either as a feedback source or as a feedback sink. By giving both nodes the same identifier, they get a logical connection without having to wire them together. The type of operation is determined by whether any input is wired: A node that has two unwired inputs will only receive default (empty) tuple stream objects, and thus become a sink.

This solution does not account for any custom initial tuple. ■



```
Object  Message
print   1
print   2
print   3
print   4
print   5
print   6
print   7
print   8
print   9
print   10
```

Figure 79: A simple feedback loop in Max.
The iterator function is named "uzi".

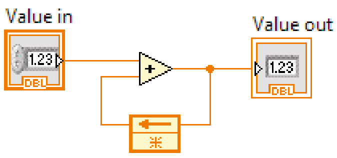


Figure 80: Example of a feedback loop in LabVIEW. The star symbol in the feedback node represents the default value, either a default wire value (initially zero), or a connected wire input.

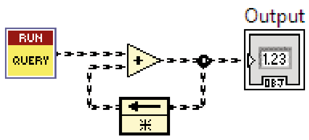


Figure 81: A faulty diagram using actors.

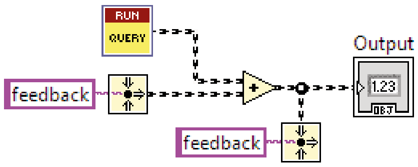


Figure 82: A better diagram. The left copy of the feedback node is the sink, the right one is the source.

References

[1] Gul A. Agha
ACTORS: A Model of Concurrent Computation in Distributed Systems
1985

[2] Allen L. Ambler, Margaret M. Burnett
Visual Forms of Iteration that Preserve Single Assignment
Journal of Visual Languages and Computing 1, 1990, pp 159–181

[3] Apache NiFi Team
Apache NiFi Overview, 8 December 2015
https://nifi.apache.org/docs.html

[4] the Apache Software Foundation
Apache Flink, 2015
https://flink.apache.org

[5] the Apache Software Foundation
Apache Storm, 2015
http://storm.apache.org

[6] the Apache Software Foundation
Spark Streaming, 14 October 2015
http://spark.apache.org/streaming

[7] B. Babcock, M. Datar, R. Motwani
Load shedding for aggregation queries over data streams
Data Engineering, Proceedings. 20th International Conference on, 2004, pp 350–361

[8] Y. Bai, H. Thakkar, H. Wang, C. Zaniolo
Time-Stamp Management and Query Execution in Data Stream Management Systems
Internet Computing, IEEE 12(6), 2008, pp 13–21

[9] Ed Baroth, Chris Hartsough
Experience Report: Visual Programming in the Real World
Visual Object Oriented Programming, edited by MM Burnett, A. Goldberg & TG Lewis, Manning Publications, Prentice Hall, 1995

[10] E. Bauleo, S. Carnevale, T. Catarci, S. Kimani, M. Leva, M. Mecella
Design, realization and user evaluation of the SmartVortex Visual Query System for accessing data streams in industrial engineering applications
Journal of Visual Languages and Computing 25, 2014, pp 577–601

[11] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, M. Raulet
OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems
ACM SIGARCH Computer Architecture News 36(5), 2009, pp 29–35

[12] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, C. Moran
IBM InfoSphere Streams for Scalable, Real-Time, Intelligent Transportation Services
in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10), New York, USA

[13] Robert H. Bishop
LabVIEW 2009 Student Edition
Prentice Hall Press, 2009

[14] M. Brettel, N. Friederichsen, M. Keller, M. Rosenberg
How Virtualization, Decentralization and Network Building Change the Manufacturing Landscape: An Industry 4.0 Perspective
International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering 8(1), 2014, pp 37–44

[15] Marat Boshernitsan, Michael Downes
Visual Programming Languages: A Survey
Computer Science Division, University of California, Berkeley, 2004

[16] Sharma Chakravarthy, Raman Adaikkalavan
Events and Streams: Harnessing and Unleashing Their Synergy!
Proceedings of the second international conference on Distributed event-based systems, ACM, 2008, pp 1–12

[17] Donald D. Chamberlin
The "single-assignment" approach to parallel processing
AFIPS '71 (Fall) Proceedings of the November 16–18, fall joint computer conference, 1971, pp 263–269

[18] Gard J. Clark, C. Thomas Wu
DFQL: Dataflow query language for relational databases
Information & Management 27(1), July 1994, pp 1–15

[19] Melvin E. Conway
Design of a separable transition-diagram compiler
Communications of the ACM 6(7), July 1963, pp 396–408

[20] Arvind Culler, David E. Culler
Dataflow Architectures
Annual Review of Computer Science 1, June 1986, pp 225–253

[21] DataWatch
Literature, 2016
http://datawatch.com/explore/literature

[22] Alan L. Davis and Robert M. Keller
Data Flow Program Graphs
Computer 2(15), 1982, pp 26–41

[23] P. Dekkers, M. van Vliet, P. E. M. Ligthart, H. de Man, G. Ligtenberg
Complex event processing
Master's Thesis, Radboud University Nijmegen, Nijmegen, Netherlands
October 2007

[24] Ramez Elmasri, Sham Navathe
Database Systems: Models, Languages, Design, and Application Programming
Pearson, 2011

[25] Nesimi Ertugul
Towards Virtual Laboratories: a Survey of LabVIEW-based Teaching/ Learning Tools and Future Trends
International Journal of Engineering Education 16(3), 2000, pp 171–180

[26] Peter C. Evans, Marco Annunziata
Industrial internet: Pushing the boundaries of minds and machines
General Electric, 2012

[27] Gustav Fahl, Tore Risch, Martin Sköld
AMOS: An Architecture for Active Mediators
University of Linköping Institute of Technology, Computer Science Department, 1993

[28] A. Gal, S. Keren, M. Sondak, M. Weidlich, H. Blom, C. Bockermann
Grand Challenge: The TechniBall System
Proceedings of the 7th ACM international conference on Distributed event-based systems, 2013

[29] T. Ghanem, M. Hammad, M. Mokbel, W. Aref, A. Elmagarmid
Incremental Evaluation of Sliding-Window Queries over Data Streams
Knowledge and Data Engineering, IEEE Transactions on 19(1), January 2007, pp 57–72

[30] Lukasz Golab, M Tamer Öszu
Data Stream Management
Morgan & Claypool, 2010

[31] T. Grabs, R. Schindlauer, R. Krishnan, J. Goldstein, R. Fernandéz
Introducing Microsoft StreamInsight
Technical report, 2009

[32] M. P. e. a. van Haarlem, et al.
LOFAR: The LOw-Frequency ARray
Astronomy & Astrophysics 556, August 2013

[33] Carl Hewitt
What is computation? Actor model versus Turing's model
A Computable Universe: Understanding and Exploring Nature as Computation, Singapore, World Scientific, 2013, pp 159–185

[34] Daniel D. Hils
Visual Languages and Computing Survey: Data Flow Visual Programming Languages
Journal of Visual Languages and Computing 3, 1992, pp 69–101

[35] William Humphrey, Andrew Dalke, Klaus Schulten
VMD: Visual Molecular Dynamics
Journal of molecular graphics 14(1), 1996, pp 33–38

[36] Christopher Ireland, David Bowers, Michael Newton, Kevin Waugh
A Classification of Object-Relational Impedance Mismatch
Advances in Databases, Knowledge, and Data Applications, DBKDA'09. First International Conference on. IEEE, 2009

[37] M. Jergler, C. Doblander, M. Najafi, H-A. Jacobsen
Grand Challenge: Real-time Soccer Analytics Leveraging Low-Latency Complex Event Processing
Proceedings of the 7th ACM international conference on Distributed event-based systems, 2013

[38] Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar
Advances in Dataflow Programming Languages
ACM Computing Surveys (CSUR) 36(1), 2004, pp 1–34

[39] V. Kakkad, S. Attar, A. E. Santosa, A. Fekete, B. Scholz
Curracurrong: a stream programming environment for wireless sensor networks
Software: Practice and Experience 44(2), February 2014, pp 175–199

[40] Scott Kennedy
Made in China 2025
Center for Strategic & International Studies, 1 June 2015
http://csis.org/publication/made-china-2025

[41] Keysight Technologies
VEE Pro 9.32 Data Sheet
http://literature.cdn.keysight.com/litweb/pdf/5990-9117EN.pdf

[42] H. J. Kim, H. F. Korth, A. Silberschatz
PICASSO: a graphical query language
Software: Practice and Experience 18(3), March 1988, pp 169–203

[43] Jay Lee, Behrad Bagheri, Hung-An Kao
A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems
Manufacturing Letters 3, 2015, pp 18–23

[44] Witold Litwin
Linear Hashing: a new tool for file and table addressing
VLDB 80, 1980

[45] David Luckham
The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems
Springer, 2008

[46] E. Lunca, S. Ursache, O. Neacsu
Graphical Programming Tools for Electrical Engineering Higher Education
iJOE 7(1), 2011, pp 19–24

[47] A. Malagoli, M. Leva, S. Kimani, A. Russo, M. Mecella, S. Bergamaschi, T. Catarci
Visual Query Specification and Interaction with Industrial Engineering Data
G. Bebis et al. (Eds.): ISVC 2013, Part II, LNCS 8034, 2013, pp 58–67

[48] Kudlur Manjunath, Scott Mahlke
Orchestrating the Execution of Stream Programs on Multicore Platforms
ACM SIGPLAN Notices 43(6), 2008, pp 114–124

[49] Lars Melander
SensorGui – a digital radio data graphical interface
Uptec IT 04031, 2004

[50] Priti Mishra, Margaret H. Eich
Join Processing in Relational Databases
Journal ACM Computing Surveys (CSUR) Surveys Homepage archive 24(1), March 1992, pp 63–113

[51] Matt Morley
JSON-RPC 2.0 Specification
JSON-RPC Working Group <json-rpc@googlegroups.com>, 04 January 2013
http://jsonrpc.org/specification

[52] A. L. de Moura, R. Ierusalimschy
Revisiting Coroutines
ACM Trans. Program. Lang. Syst. 31(2), Article 6, February 2009, pp 31

[53] Barzan Mozafari, Carlo Zaniolo
Optimal Load Shedding with Aggregates and Mining Queries
Data Engineering (ICDE), 2010 IEEE 26th International Conference on, IEEE, 2010, pp 76–88

[54] Brad A. Myers
Taxonomies of Visual Programming and Program Visualization
Journal of Visual Languages and Computing 1, 1990, pp 97–123

[55] P. J. Napier, A. R. Thompson, R. D. Ekers
The Very large Array: Design and Performance of a Modern Synthesis Radio Telescope
Proceedings of the IEEE 71(11), 1983, pp 1295–1320

[56] National Instruments
Actor Framework, 28 July 2011
http://ni.com/actorframework

[57] National Instruments
BridgeVIEW™ and LabVIEW™—G Programming Reference Manual, January 1998
http://ni.com/pdf/manuals/321296b.pdf

[58] National Instruments
Creating New Front Panel Objects with LabVIEW XControls, 30 Mars 2012
http://ni.com/tutorial/3198/en

[59] National Instruments
G Dataflow (G)
http://ni.com/documentation/en/labview-comms/latest/g-prog/dataflow

[60] National Instruments
LabVIEW Object-Oriented Programming: The Decisions Behind the Design,
21 May 2014
http://ni.com/white-paper/3574/en

[61] National Instruments
LabVIEW Run-Time Engine Compatibility, 26 November 2014
http://digital.ni.com/public.nsf/
allkb/800E68EBF895BD9686257077005IFF36

[62] National Instruments
LabVIEW Tools Network
http://ni.com/labview-tools-network

[63] National Instruments
Queue Operations Functions, June 2014
http://zone.ni.com/reference/en-XX/help/371361L-01/glang/queue_vis

[64] National Instruments
Reentrancy: Allowing Simultaneous Calls to the Same SubVI, June 2014
http://zone.ni.com/reference/en-XX/help/371361L-01/lvconcepts/
reentrancy

[65] National Instruments
Tutorial: SubVIs, 1 April 2015
http://ni.com/white-paper/7593/en

[66] National Instruments
Virtual Instrumentation, 4 February 2013
http://ni.com/white-paper/4752/en

[67] National Instruments
White Papers, 2016
http://ni.com/white-papers

[68] Mehmet A. Orgun, Wanli Ma
An overview of temporal and modal logic programming
Lecture Notes in Computer Science 827, 1994, pp 445–479

[69] W. Pauli, M. L. Soffa
Coroutine Behaviour and Implementation
Software: Practice and Experience 10(3), 1980, pp 189–204

[70] John Plaice, Blanca Mancilla, Gabriel Ditu
From Lucid to TransLucid: Iteration, Dataflow, Intensional and Cartesian Programming
Mathematics in Computer Science 2(1), 2008, pp 37–61

[71] Pedro Ponce-Cruz, Fernando D. Ramírez-Figueroa
Intelligent Control Systems with LabVIEW
Springer, 2010

[72] T. Risch, S. Badiozamany, D. Heutelbeck, L. Karlsson, M. Löfstrand, L. Melander, K. Orsborn, T. Truong, D. Wedlund, C. Xu, M. Johansson
D5.1: SMART VORTEX DSMS report and specification
SMART VORTEX –WP5-D5.1, FP7-ICT-257899, July 2014

[73] Tore Risch, Vanja Josifovski
Distributed Data Integration by Object-Oriented Mediator Servers
Concurrency and Computation: Practice and Experience J. 13(11), John Wiley & Sons, September 2001

[74] Tore Risch, Vanja Josifovski, Timour Katchaounov
Functional Data Integration in a Distributed Mediator System
Published in P.M.D.Gray, L.Kerschberg, P.J.H.King, and A.Poulovassilis (eds.): Functional Approach to Computing with Data, Springer

[75] David B. Robins
Complex Event Processing
University of Washington, Redmond, WA, 6 February 2010

[76] Sandvik Coromant
Knowledge
http://sandvik.coromant.com/en-gb/knowledge/pages/default.aspx

[77] Sandvik Coromant
Tools for metal cutting
http://sandvik.coromant.com/en-gb/products/pages/tools.aspx

[78] J. Sermulins, W. Thies, R. Rabbah, S. Amarasinghe
Cache Aware Optimization of Stream Programs
ACM SIGPLAN Notices 40(7), 2005, pp 115–126

[79] Will Schroeder, Ken Martin, Bill Lorensen
Visualization Toolkit: An Object-Oriented Approach to 3D Graphics
Kitware; 4th edition, 1 December 2006

[80] Software AG
White Papers, 2016
https://softwareag.com/corporate/res/wp/Default.asp

[81] Masahiro Sowa, Tadao Murata
A Data Flow Computer Architecture with Program and Token Memories
IEEE transactions on computers C-31(9), September 1982

[82] SQLStream
Resources
http://sqlstream.com/resources

[83] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker
Load shedding in a data stream manager
VLDB '03 Proceedings of the 29th international conference on Very large data bases, 2003, pp 309–320

[84] M. Tawfik, E. Sancristobal, S. Martin, G. Diaz, M. Castro
State-of-the-Art Remote Laboratories for Industrial Electronics Applications
Technologies Applied to Electronics Teaching (TAEE), 2012, pp 359–364

[85] Robert D. Tennent
Functor-category semantics of programming languages and logics
Springer Berlin Heidelberg, 1986

[86] L. G. Tesler, H. J. Enea
A language design for concurrent processes
AFIPS '68 (Spring) Proceedings of the April 30–May 2, spring joint computer conference, 1968, pp 403–408

[87] TIBCO
Special Reports, White Papers and Datasheets
http://streambase.com/news-and-events/reports-and-surveys

[88] Daniel D. Traficante
Impedance: What It Is, and Why It Must Be Matched
Concepts in Magnetic Resonance 1, 1989, pp 73–92

[89] M. de Vos, A. W. Gunst, R. Nijboer
The LOFAR Telescope: System Architecture and Signal Processing
Proceedings of the IEEE 97(8), 2009, pp 1431–1437

[90] Arne Wang, Ole-Johan Dahl
Coroutine sequencing in a block structured environment
BIT Numerical Mathematics 11(4), December 1971, pp 425–449

[91] Kirsten N. Whitley and Alan F. Blackwell
Visual Programming in the Wild: A Survey of LabVIEW Programmers
Journal of Visual Languages and Computing 12(4), 2001, pp 435–472

[92] Yingjun Wu, David Maier, Kian-Lee Tan
Grand Challenge: SPRINT Stream Processing Engine as a Solution
Proceedings of the 7th ACM international conference on Distributed event-based systems, 2013

[93] C. Xu, D. Wedlund, M. Helgoson, T. Risch
Model-based Validation of Streaming Data
Proceedings of the 7th ACM international conference on Distributed event-based systems, 2013

[94] H. Yviquel, E. Casseau, M. Wipliez and M. Raulet
Efficient multicore scheduling of dataflow process networks
Signal Processing Systems (SiPS), IEEE Workshop on, 2011, pp 198–203

[95] Erik Zeitler
Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams
Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology, 2011

[96] Erik Zeitler, Tore Risch
Processing high-volume stream queries on a supercomputer
ICDE Ph.D. Workshop 2006, Atlanta, GA, April 2006

[97] ZoomData
Streaming Data: Why It Makes Sense and How to Work With It
2015

65. *Mats Ekman*: Modeling and Control of Bilinear Systems. Application to the Activated Sludge Process. 2005.
66. *Milena Ivanova*: Scalable Scientific Stream Query Processing. 2005.
67. *Zoran Radovic´*: Software Techniques for Distributed Shared Memory. 2005.
68. *Richard Abrahamsson*: Estimation Problems in Array Signal Processing, System Identification, and Radar Imagery. 2006.
69. *Fredrik Robelius*: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. *Anna Davour*: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. *Magnus Ågren*: Set Constraints for Local Search. 2007.
72. *Ahmed Rezine*: Parameterized Systems: Generalizing and Simplifying Automatic Verification. 2008.
73. *Linda Brus*: Nonlinear Identification and Control with Solar Energy Applications. 2008.
74. *Peter Nauclér*: Estimation and Control of Resonant Systems with Stochastic Disturbances. 2008.
75. *Johan Petrini*: Querying RDF Schema Views of Relational Databases. 2008.
76. *Noomene Ben Henda*: Infinite-state Stochastic and Parameterized Systems. 2008.
77. *Samson Keleta*: Double Pion Production in $dd \rightarrow \alpha\pi\pi$ Reaction. 2008.
78. *Mei Hong*: Analysis of Some Methods for Identifying Dynamic Errors-invariables Systems. 2008.
79. *Robin Strand*: Distance Functions and Image Processing on Point-Lattices With Focus on the 3D Face-and Body-centered Cubic Grids. 2008.
80. *Ruslan Fomkin*: Optimization and Execution of Complex Scientific Queries. 2009.
81. *John Airey*: Science, Language and Literacy. Case Studies of Learning in Swedish University Physics. 2009.
82. *Arvid Pohl*: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
83. *Anna Danielsson*: Doing Physics – Doing Gender. An Exploration of Physics Students' Identity Constitution in the Context of Laboratory Work. 2009.
84. *Karin Schönning*: Meson Production in pd Collisions. 2009.
85. *Henrik Petrén*: η Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
86. *Jan Henry Nyström*: Analysing Fault Tolerance for ERLANG Applications. 2009.
87. *John Håkansson*: Design and Verification of Component Based Real-Time Systems. 2009.
88. *Sophie Grape*: Studies of PWO Crystals and Simulations of the $\bar{p}p \rightarrow \bar{\Lambda}\Lambda, \bar{\Lambda}\Sigma^0$ Reactions for the PANDA Experiment. 2009.
90. *Agnes Rensfelt*. Viscoelastic Materials. Identification and Experiment Design. 2010.
91. *Erik Gudmundson*. Signal Processing for Spectroscopic Applications. 2010.
92. *Björn Halvarsson*. Interaction Analysis in Multivariable Control Systems. Applications to Bioreactors for Nitrogen Removal. 2010.
93. *Jesper Bengtson*. Formalising process calculi. 2010.
94. *Magnus Johansson*. Psi-calculi: a Framework for Mobile Process Calculi. Cook your own correct process calculus – just add data and logic. 2010.
95. *Karin Rathsman*. Modeling of Electron Cooling. Theory, Data and Applications. 2010.