Universität Bielefeld
Technische Fakultät
AG Praktische Informatik
CeBiTec

Conveyor

A workflow engine for bioinformatics analyses

Zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften an der Technischen Fakultät der Universität Bielefeld vorgelegte Dissertation

von

Burkhard Linke

August 15, 2011

Supervisors: Prof. Dr. Robert Giegerich
Prof. Dr. Frank Oliver Glöckner
Dr. Alexander Goesmann

Burkhard Linke
Breite Strasse 3
33602 Bielefeld
blinke@CeBiTec.Uni-Bielefeld.DE

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The various fields of the *omics life sciences (genomics, transcriptomics, proteomics to name a few) have shown a very high pace of development in the last decade. New laboratory technologies like next-gen sequencing or high-throughput GC-MS are able to create a tremendous amount of data. On the other hand, the cost and availability of compute power has dropped to an all time low.

Combining life sciences and computer science is the field of bioinformatics, which also has seen a growth without comparison over the last decade, and many new algorithms and applications like short read mapping have been enabled by new developments in life sciences.

Keeping pace with the development and data flood from the laboratories imposes a need for agile and flexible software development approaches. Unfortunately, the conventional software development is too slow, even with scripting languages like Perl or Python. This is why workflow systems have been proposed and emerged. They break down a complex task into smaller problems, orchestrate solving the sub-problems and merging the solutions, and to some extend also handle data management.

Conveyor is a workflow engine which uses a novel approach for defining and deploying of analytical workflows. It is based on a full generic object-oriented data model, and allows a data-driven, interface based design of workflows. The engine is extensible by plugins, allowing almost any functionality to be included in a workflow. The applicability of Conveyor to bioinformatics applications is demonstrated in this thesis by several use cases.

Conveyor has been published in the Oxford Bioinformatics Journal ([LGG11]).

CHAPTER 2

Analysis

Bioinformatics has been a fast growing field over the last decade, and new developments and breakthroughs happen almost every year, both on the laboratory side producing data and on the analytical side processing the data. The ultra-fast sequencing methods introduced in the last years have created completely new opportunities for conducting experiments and examining larger genetic samples. One example is metagenomics, the analysis of a complete biotope without sequencing invidual specimen. It has not been feasible beforehand or required extraordinary efforts. With more and more data available, the demand for data storage and data processing have changed in a similar radical way. Flexibilty and agility in setting up analysis pipelines have become an important aspect of modern bioinformatics tools. One way to achieve these goals is using workflow systems instead of complex, monolithic applications.

2.1 A definition of workflows

The term *workflow* is used in many different contexts. It always refers to the process of breaking up a large task into smaller ones, and arranging these smaller units in a logical and temporal order to replace the large task.

A simple example of a workflow is the receipt for baking a cake. Put butter in a dish, add sugar, eggs and flour, stir it up, put the dought into a cake tin, and finally put the tin into an oven for a given time and a given heat.

This example already demonstrates some of the key features of workflows:

- they are built from smaller tasks
- tasks may be arranged in a successive or parallel way (e.g. ingredients may be put into the

dish simultaneously, but the cake tin may only be put into the oven after it was filled)

Another example for workflows are the laboratory protocols used to conduct an experiment. Figure 2.1 shows an excerpt of such a protocol. It also shows another feature of workflows:

- the same step may show up multiple times with different specifications

The final example for workflows are computer programs. They consist of a sequence of simple, reusable instructions taken from a predefined set. There is a well defined order of execution, and multi-threaded programs also parallelize multiple control flows.

While programs itself may be regarded as workflows, the term is used in a different context in computer applications. The steps are usually higher level tasks, e.g. the various steps of a use case in a business application.

In the scope of this thesis, only a special kind of workflow is considered:

- data-driven
- non-interactive (no step requires user interaction)
- batch

2.2 Requirements for bioinformatic analysis workflow systems

Modern data processing and analysis in the field of bioinformatics has to cope with a number of requirements that arise due to the nature of biological data:

Data-driven processing The amount of data being processing in a typical workflow has grow by several orders of magnitude during the latest developments in laboratory equipment. A complete run of a Roche 454 Genome Analyzer FLX generates up to one billion bases per run; an Illumina Genome Analyzer IIx has a yield of up to 70 billion bases. New developments will push the output size even further to 500 billion bases. While sequencing is only one aspect of bioinformatics, new developments in proteomics and metabolomics also have increased the output size by several orders of magnitude.

The different file formats in use complicate the design and implementation of workflows. Data has to be converted between formats, losing information in almost every case. Thus a modern workflow system should be based on data instead of file formats, and only do necessary conversion if interacting with external applications. The data should be defined independent of concrete formats.

Security and discretion The data processed by bioinformatics workflows has to be handled with various levels of discretion. In most cases, unpublished or secret data is involved which should not be accessible for unauthorized third parties. This also includes sending the data to external web services to perform analysis.

Legacy applications integration The backbone of data processing in bioinformatics are legacy applications like BLAST, that perform special operations or implement algorithms. Depending on the application several options exists for executing such tools during workflow

```
Total Bacterial RNA Labeling with Random Hexamers.
Arkady Khodursky, Ph.D.
Email: khodursk@cmgm.stanford.edu
Updated: 2/10/99

20-25ug of total RNA in 10-14 ul of diH20 should be purified using  RNA kit
or hot phenol extraction. After obtaining RNA sample it is important to get
rid of residual DNA with RNase-free DNAse followed by 1x PHOH,
1xPHOH/CHCl3, 2x CHCl3 extraction. Extent of RNA purity achieved by
PHOH-CHCl3 extraction is absolutely critical for successful labeling.

Above indicated amount of RNA should be mixed with 500ng of random pdN6
primer on ice in total volume of 15 ul.

Incubate at 65 C for 10’.

Incubate on ice for 2’.

Add 3 ul of  1 mM FluoroLink Cy3 (orCy5) dUTP (Amersham Cat# PA53022).

Add 11.6 ul an mix everything by pipeting 3-4 times on ice of reverse
transcription mix:
0.1 DTT - 3ul
5x 1st strand buffer - 6 ul
dNTPs (25 mM dATP, 25 mM dCTP, 25 mM dGTP, 10 mMdTTP) - 0.6 ul
SuperscriptII - 2 ul
(All from GibcoBRL Cat.# 18064-014)

Incubate the complete mixture for 10’ at RT.

Transfer to 42 C for 110’.

Stop by adding 1.5 ul of 1n NaOH for 10’ at 65 C.

Neutralize with 1.5 ul of 1M HCL.
```

Figure 2.1: Laboratory protocol example for bacterial RNA labeling, taken from the Stanford website. In addition to other steps it contains several incubation steps with varying time spans and temperatures.

processing:

local processing The application is executed on the local host only. This option is favourable with respect to data security, since no data is passed over a network. Its drawback is the limited processing power even larger computer system offer today. On the other hand it allows using special hardware like GPUs oder FPGAs for faster analysis.

local cluster A local compute cluster can be used for distributing applications over several compute hosts. In most cases a complex infrastructure is required for local clusters, starting with the compute hosts, and ending with shared network storage for data and application. Depending on the application, the performance of the clustered approach may scale well with the size of the cluster. Since the cluster is part of the local infrastructure, the security aspect of this solution depends on its setup and configuration. Special purpose hardware like the aforementioned GPUs and FPGAs may also be made available in a local cluster.

cloud (remote cluster) The much hyped *clouds* are compute clusters operated by third parties. Clients may rent machines in various configuration and sizes, running their custom software on them. After an initial setup, clouds are easy to use and do not require maintenance by the clients. While a local cluster requires a regular investigation of time and money, cloud-based computing is paid by the amount of computing power actually used.

The drawback of cloud computing is the uncertainty with respect to security aspects. Hosts in a cloud run outside the control of the client. Depending on the setup, several different clients may share the same physical machine in case of virtualized instances. Although communication with and between the cloud hosts may be encrypted, the security of the host itself cannot be supervised by the client.

(remote) web service Web services offer a standardized way of providing services over the network. Service discovery, specification, and invocation are defined by standard bodies to allow interoperability between different platforms and programming languages. A number of large bioinformatics related institutes offer access to their data and applications by web services.

Several drawbacks make web services the least favourable solution for executing applications within workflows. They add a noticable overhead to execution time due to the time needed for formatting the request, sending it over the network, decoding and processing it, and finally sending a formatted result back to the client. For bioinformatics applications, several different implementations like BioMoby[WSK$^+$08] or SoapLap[SRBU09] exist with different data formats and different discovery methods. Direct interoperability between different service implementations is not possible in most cases.

Security aspects are the major drawbacks of web services. Some discovery methods like BioMoby do not even offer to select encrypted services or use trusted services only. Authentication of the service is optional for most services.

Extensibility New data formats and new applications are constantly created in the fields of bioinformatics, so a workflow system has to be able to adopt to changes.

Integration The workflow system itself is used as a tool for processing data. Data management and result visualization may not be part of the workflow system and applications that handle these aspects already exist. The workflow system should be easily integratable into existing software, being able to access their data and passing the results for storage and visualization.

The requirements are gathered with a large data amount, fast data processing, security concerns, and flexibility in mind. The next section presents selected existing systems and evaluates them with respect to these requirements.

2.3 Existing workflow systems for bioinformatics analysis

A number of workflow based applications and frameworks have already been published and presented for bioinformatics workflows. These applications offer different approaches, varying functionality, and multiple solutions to the requirements pointed out in the previous section. The following subsections describe such applications in order to examine whether they fulfill these requirements.

2.3.1 Mobyle

Mobyle[NMM+09] is a web based environment for accessing bioinformatics tools and data management. The software is freely available, and several institutes provide free guest access to their server.

Data processing is based on files. A tool is described by its input format(s), the output format(s), and configuration settings. The result of one tool can be directed into another if output and input formats match. A processing pipeline can thus be built by concatenating a number of tools. In case of non-matching formats a conversation is necessary (if available).

Many standard bioinformatics tools are already supported by Mobyle. Extending the repository is a matter of writing an application description and providing a commandline implementation of the tool.

Since Mobyle is freely available, the security requirements can be easily fulfilled by setting up a private instance. Execution of external applications is configurable, and compute clusters are supported as execution backend.

Due to the web based nature of Mobyle, an integration into existing software is not possible.

2.3.2 Taverna

Taverna[HWS+06] is the workflow solution of the *myGrid* project[1], "a multi-institutional, multi-disciplinary internationally leading research group focussing on the challenges of eScience". It is a freely available, Java-based application. Workflows designed with Taverna can be published and exchanged using myExperiment[DRGS08], a researcher community being part of myGrid.

Taverna is built around web services, and comes with a number of preconfigured service repositories, for example BioMOBY([WSK+08]) or SoapLap([SRBU09]) based registries. The myGrid project developed an ontology of data formats and mapped each native service format to that ontology. This allows for using services from different sources and different registries in the same workflow, as long as their data formats map to compatible ontology entries. Conversion services may be used in case of incompatible formats.

The default registries already contain almost all standard bioinformatics tools; in many cases there are even multiple services providing the same tools, e.g. for BLAST[AMS+97] based services in BioMOBY. To extend an available services, a new bioinformatic tool can be deployed as a web service, and registered with Taverna or one of the registries Taverna uses. Functionality may also be implemented locally as a part of Taverna, to a limited extent.

Since Taverna relies on external web services, creating a secure and discrete environment requires setting up a local registry and local web services for all necessary functions. Although possible, this setup requires a huge effort, which may not be feasible under most circumstances.

2.3.3 Pegasus WMS

Pegasus WMS[DSS+05] is a workflow management system designed for distributed computing. It is freely available, and binary packages for various Linux distributions are offered on the corresponding website.

Workflows are file based and built on external applications. Instances of files and applications are defined in an abstract workflow definition, including their relations.

Pegasus itself does not handle applications. The abstract workflow definition refers to application names, which are either mapped to executable described in the same file, or a site-specific catalog is used.

To execute a Pegasus workflow, the abstract workflow definition is converted to a physical workflow definition, based on the architecture of the execution environment, the availability of the executable, and other constraints. Pegasus is able to handle local execution, complex grid computing environments, and cloud-based setups. Necessary extensions like file transfer between compute nodes or cloud-instances are added to the physical workflow definition on demand. Execution itself is handled by the integrated Condor workload manager[TTL05]. Security and data discretion aspects depend on the setup of Pegasus and the execution environment, and the documentation available for Pegasus and Condor describe how to build a secure setup.

[1] http://www.mygrid.org.uk.

A number of graphical user interfaces for workflow design and application definition are built on Pegasus, and libraries for Java and Python are bundled with the Pegasus distribution that allows creating workflows programmatically.

2.3.4 Kepler

Kepler[ABJ+04] is a generic workflow engine with applications in various different fields of research. It is a freely available standalone java application.

Workflows are built from components called *actors* and include definitions how their actors interact with each other. The actors in turn are implemented as Java classes and bundled into libraries. A large collection of predefined actors exists, focussing on many aspects of numerical data analysis. A small set of actors related to bioinformatics is already implemented but a large number of standard tools are missing from the available actor collections.

Kepler provides different *directors* that orchestrate the actors during execution and implement different ways to execute a workflow. There exists directors for single actor execution, parallel execution, or time-triggered execution, just to name a few.

A workflow is executed on the local machine as long as no external data source or actors accessing external services are involved. Security and discretion aspects are thus reduced to selecting the right actors and directors for a workflow.

Kepler is based on an underlying framework named *Ptolemy II*([EJL+03]) for defining actors. This framework provides a type system for data passed between actors. It includes various numerical types (integer, float, double etc.), string and derived types like *XMLToken*[2], booleans and opaque objects. Relations between types express the ability to (losslessly) convert one type to another. Compound types like arrays or records are also exported.

In addition to data processing actors, Kepler also includes a number of actors executing control flow tasks like branching or workflow control. Actors may also be composed of nested actors, allowing workflows to be arbitary complex.

A graphical user interface for designing and executing workflows is part of the Kepler distribution.

2.3.5 Galaxy

Galaxy[GNT+10] is a data management and processing system. It allows both ad-hoc processing and workflow-based processing. It is a web-based application and freely available.

Data is managed in a file-based way, using file name extensions or file introspection to define the type of a file. Workflows are built from *tools*, and files are passed between the various tool instances of a workflow. Tools are based on external applications and described in an XML format. A large number of predefined tool definitions are bundled with Galaxy, with a focus on

[2]a string with restrictions on valid characters

next generation sequencing data processing. Adding new tools requires providing an external application or script, and writing a tool definition file.

Since Galaxy is based on local applications, security aspects can be answered by installing a local instance.

The Galaxy web interface is very intuitive and easy to use, and a lot of documentation exists.

2.3.6 Conclusion

The workflow systems outlined in the previous sections represent the state-of-art in bioinformatics, and are supported by a number of larger international collaborations. Table 2.1 lists the set of requirements defined in section 2.2 and states which are supported by a given workflow system. Mobyle is a nice tool for simple processing pipelines, but it uses a file-based data management. Taverna is an excellent tool for integrating data and processing facilities from various sources, but it cannot be used with confidential data. Pegasus is probably the most outstanding system with respect to its distributed processing facilities, but it also offers only file based operations. Kepler is the most promising tool, but its type system is mainly focussed on primitive values and lacks support for complex types like classes, and only a small number of bioinformatics tools are already implemented. Finally, Galaxy is the most user-friendly tool[3], and allows users to quickly analyse their data and design workflows. Again, the file-based concept does not fulfill the requirements.

Requirement	Mobyle	Taverna	Pegasus WMS	Kepler	Galaxy
Data driven processing	-	-	-	+[4]	-
Security and discretion	+[5]	-	+[6]	+	+
Legacy applications					
locally	+	-[7]	+	-	+
local cluster	+	-	+	-	-[8]
cloud	-	-	+	-	-
web service	-	+	-	-	-
Extensibility	+	+	+	+	+
Integration	-	+[9]	+[9]	-	-

Table 2.1: Comparison of the existing workflow systems described in the previous section. Each row lists one of the requirements defined in section 2.2 and which system fulfills that requirement.

[3]personal opinion of the author
[4]limited with respect to features available
[5]web interface access only, but designed for running external tools on local infrastructure
[6]security aspects depend on the method of execution used for Pegasus
[7]focussed on web services, limited support for local functionality
[8]tool wrappers in Galaxy may be setup to use a local cluster
[9]based on command line tools for workflow execution

As shown in the table, none of the presented workflow-systems fulfill the complete set of requirement. While all systems allow to be extended with new functionality, integration into existing application is only possible by command-line clients for executing a workflow. A direct transfer of data between an application and the workflow is only possible by the use of files in the Pegasus case.

One of the most important aspects is data-driven processing. As stated in the introduction, the amount of data to be processed has grown of the last years, and is expected to further increase within the future. With file and web interface based setups, attempts to work with large datasets becomes infeasible.

CHAPTER 3

Design of the Conveyor workflow system

This chapter describes the design of the *Conveyor* workflow system based on the requirements defined in chapter 2:

- data-driven processing
- extensibility
- integration with existing applications
- secure and discrete data handling
- support for legacy applications

Although the design has to take all these requirements into account simultaneously, the different aspects have a varying degree of influence on it. As pointed out in chapter 2, the way to execute external applications also defines which security aspects have to be considered. These are discussed in a separate section of chapter 4.

The following sections describe the user roles involved in Conveyor, the overall system design, the representation of workflows, and associated components.

3.1 System design

The overall design of Conveyor is split into three parts as shown in figure 3.1:

Plugins The plugins define the workflow steps and data entities available to the workflow system.

Definition and Processing This subsystem contains the component for managing the plugins, the representation of an actual workflow, and the component for executing a workflow.

Client This component uses the definition and processing subsystem to create workflows and execute them. It may be outside the scope of Conveyor itself, e.g. a third party application that uses Conveyor to process data.

This aspect of the design is mainly guided by two requirements defined above for workflows: extensibility and integration with other applications. Using a component-based approach allows Conveyor to be embedded into third party applications, either by directly linking the components, or by encapsulating the Conveyor components into a web service.

The interfaces used for communication between the components also encourage extension of the Conveyor system. Components may be exchanged with new implementations, or additional providers may be added for supporting new processing steps and data entities.

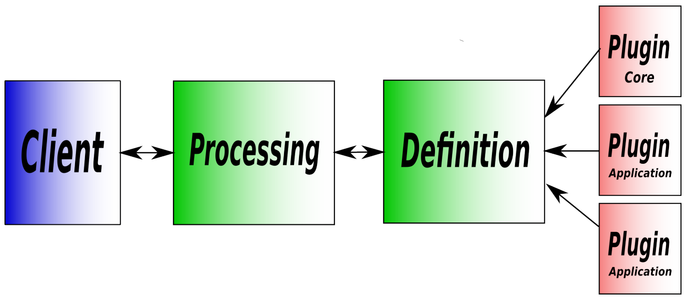


Figure 3.1: Design of the Conveyor workflow engine, with different colors depicting different functional components. Three different components interact: the plugins (red), the client (blue), and the processing and definition component mediating between them.

3.2 User roles

Within the *Conveyor* system a suer might fulfill different roles. The following list introduces these roles and gives a short description of them:

Administrator The administrator's task is to setup Conveyor and provide all necessary tools and libraries for running a Conveyor based application. It is also his responsibility to maintain Conveyor, install updates, etc.

Plugin developer The plugin developer has a background in a specific domain or application and software development. He creates the plugins that allow Conveyor to handle data of that domain and application, providing a specific functionality.

Workflow developer The workflow developer is a specialist for a certain domain. He designs workflows for domain-specific tasks.

Workflow user The workflow user finally applies an already designed workflow to solve a specific task.

As an example, we consider a workflow running BLAST for a number of protein sequences. The administrator has to ensure that the BLAST executables are installed, and corresponding databases are available. The plugin designer implements the functionality for running BLAST and analysing the BLAST output. The workflow developer designs the workflow, including the handling of sequences, passing them to BLAST, and processing the results. The workflow user defines which sequences and which database should be used.

A user may take any of the roles mentioned, e.g. a workflow developer is also a workflow user in most cases.

A role not mentioned above is the role of the *Conveyor developer*. Instead of emphasing on specific functionality, the Conveyor developer focusses on core components and the internal mechanics of Conveyor itself.

3.3 Workflow

A workflow contains the complete description of a task, including all processing steps and how the processing steps are connected to each other. Within Conveyor, a workflow is represented as a directed graph. Each node in the graph defines a processing step; the edges connecting the nodes define how data is passed between nodes. Each processing step defines how many ingoing and outgoing connections are required for each node. An example workflow is shown in figure 3.2.

The design allows simple processing pipelines, built by concatenating processing steps, or complex workflows with logic control, branches, loops and parallel flow of data. The nodes within a Conveyor graph act independently of each other. Data objects are passed one by one from node to node, and each node starts processing when its requirements are fulfilled. This model allows parallel execution of a workflow, and thus utilizes multiple cores or computers.

3.4 Plugins

A plugin is a collection of node and data types for the Conveyor system. Instead of a list of contained types maintained by the plugin developer, Conveyor uses reflection and introspection to scan the plugin and extract all necessary information at runtime. Metadata fields defined in the source code of the plugin allow the developer to add more information, e.g. a description of a node type. These metadata fields are usually embedded in the source code of the referring entity itself, thus encouraging their use already during development.

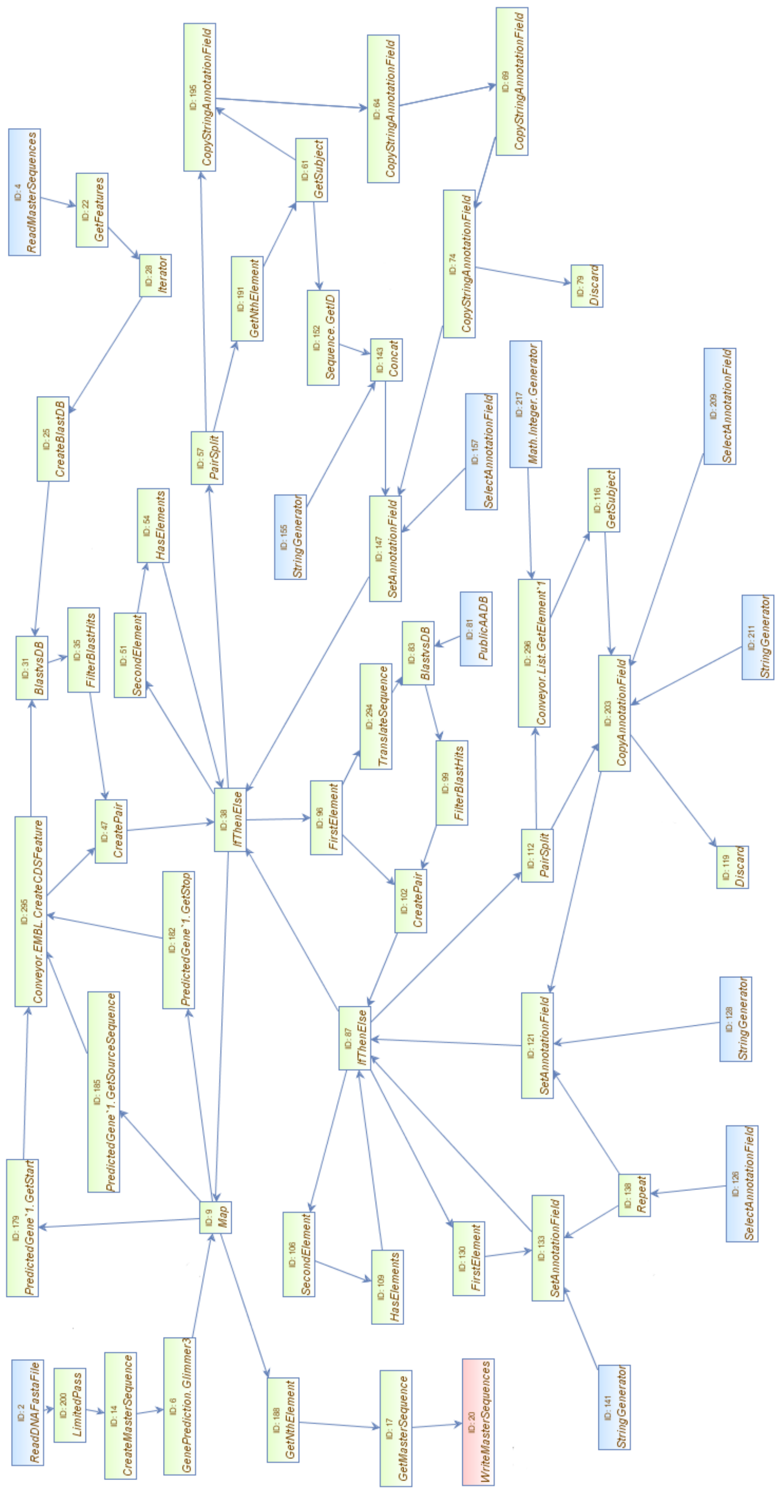


Figure 3.2: The image shows an actual Conveyor workflow for creating a reference based genome annotation (see section 5.1 for more details). The boxes are instances of various node types, and the arrows connecting the boxes indicate the flow of data between node instances. Different shades are used for input nodes (blue), processing nodes (green) and output nodes (red).

3.5 Type system

Conveyor is all about data and processing nodes of a workflow graph. Both concepts are designed around an object oriented type system providing statically strong types. Many features of type systems found in modern programming languages are supported by Conveyor:

Inheritance In particular the node types (explained in more detail below) use inheritance to create a class hierarchy and share many implementation aspects with super-classes.

Interfaces For a data-driven approach to workflows, using interfaces for expressing common concepts is crucial: Interfaces make certain aspects independent of an explicit implementation and allow exchanging the implementation without further changes. Interfaces are important for data types, since they allow using the same node type with different data types instead of creating a node type for each data type implementation.

Generics Generic types are the basis for complex, type-safe data structures. They implement a type whose concrete characteristics depend upon one or more type parameters. Instances of generic types are defined by binding these type parameters to concrete types. Well-known examples of generic types are generic lists, that abstract from the type of element they store. Applying generic types reduces the need for type checking and type casting dramatically.

Type constraints A type constraint defines a minimal requirement for a type parameter in an instance of a generic type. The constraint usually refers to an interface a type has to implement, or a base class it has to be derived from.

The combination of last three concepts give a very powerful and expressive type system for node and data types. Generic node types further encourage flexible usage and independence of concrete data types, while allowing to narrow acceptable data types e.g. by defining a type constraint for an interface.

3.6 Data types

Since Conveyor's design is following a data-driven approach, the data type design is the most crucial part. It should only inflict as little restrictions as possible to applications. To achieve this goal, a data type is defined by a class implementing an empty marker interface. No further restrictions apply, and a plugin developer is free to implement any functionality within the data type. Using the marker interface for type constraints in generic classes, the resulting type will work with any data type supported by Conveyor. The marker interface also allows an easy identification of data types. By scanning all public types for those implementing the interface, Conveyor can easily build a list of all available data types.

3.7 Node types

Besides the data types, a plugin may also contain node types that define the processing steps of a Conveyor workflow. While the data types are defined by an interface, the node types are derived from abstract base classes. They already contain most of a node type's logic and default implementations for many aspects of node types like life cycle and execution control. Similar to the data types, the available plugins are scanned for classes being derived from the abstract node types' base classes, creating a list of node types. Node types may be decorated with certain meta data attributes, that contain a description of the node type, or define special requirements for using it. The metadata is also used in the node type information presented to a workflow designer.

3.7.1 Node fields

The use of reflection to obtain information about plugins also applies to the node types itself. Almost all important aspects of node types like connections to other nodes or configurations are expressed by fields of the node type. These fields are scanned and information is extracted that result in the formal description of a node type. Using typed fields also eliminates a whole class of programming errors. Instead of evaluating the name of a connection at runtime, its name is the name of a type field and thus can be checked by the compiler at compile time. Type errors and casts also are avoided by this practice.

The following types of fields are supported by node types and detected by Conveyor:

Endpoints A node may have severral endpoints, which is a data exchange point either consuming or producing data. A Conveyor workflow is build by creating node type instances and connecting their endpoints thus modelling the data flow in the graph. An endpoint itself is defined as generic data type with the type parameter being the type of data that is exchanged via the endpoint. As a result, the node class implementation does not need to use type casts to work with data elements exchanged at an endpoint. The number of endpoints is not limited by the design, but at least one endpoint is required to make a class usable at all. Similar to the node class itself, an endpoint may be decorated with a description via attributes that are visible to the workflow designer. Two types of endpoint fields are available, one for outgoing and one for incoming data.

Configuration fields These fields define configuration values that may influence the processing within a node class. Configuration fields are visible to the user and for mandatory ones, the user is required to enter valid information prior to running the graph. A node class may have any number of configuration fields and it is the responsibility of the class itself to check that all fields contain reasonable values. Fields may be decorated with attributes, providing a description visible to a workflow designer, a default value and a flag to mark them as optional.

Settings In contrast to configuration fields mentioned before, settings are static objects that refer to a node class instead of a single node instance. They can be used to define setup and policy dependent values like the paths to executables or default values. Table 3.1 shows the various kinds of settings available in node classes. The value of a setting cannot be manipulated by the user. Instead, settings are configured by the administrator or automatically detected at runtime. Setting fields are not restricted to node types only. Other types available in a plugin may also contain settings, e.g. parsers for file formats.

Type	*Description*
Boolean	a boolean value
Directory	a directory in the file system, with optional check for existence of the directory and entries in the directory
Double	a floating point value
Executable	a binary in the file system, with optional scanning of the default search path for the binary
Integer	a 32 bit integer
String	a UTF-8 string
StringList	a list of UTF-8 strings
Type	a class implementing a given interface
TypeList	a list of classes implementing an interface

Table 3.1: Setting types available in Conveyor. Static fields using these types allow for an easy configuration of plugins and the core system by external configuration files.

3.7.2 Generic type support

Node types are ideally suited to be implemented as generic types, propagating the type parameters to the endpoint fields. Type constraints associated with the type parameters define the minimum data type of endpoints, and allows retrieving and sending concrete data objects. In addition to be independent of concrete data types (as mentioned above), generic node types also mimic *higher order functions* known from functional languages. In this connection, functions are treated as data entities, and higher order functions employ functions given as parameters in their execution. Generic node types may implement a similar scheme by endpoints that are expected to form a loop. Instead of the functional programming language case of calling a function, passing parameter to that function, and retrieving its result, a generic node type simply passes data to one or more outgoing endpoints, and expects the results at one or more incoming endpoints. The node library developed for Conveyor already contains a number of node types following this approach to provide functionality like sorting, mapping elements of a list or searching for elements in a list.

3.7.3 Node constraints

While the previously mentioned aspects focus on the definition of a node type itself, node constraints define certain aspects of the semantics of a node type.

As an example consider the *map* functionality found in many functional programming languages. A list of elements is converted to another list of (maybe different) elements by invoking a given function for each element of the source list and collecting the function's results in the target list. A node type example of the *map* function is shown in figure 3.3. Node *A* sends lists to the map node, the map node passes each element to the loop built by the *C* and *D* nodes, the results are collected in the map node and passed to the *B* node after a complete list has been processed.

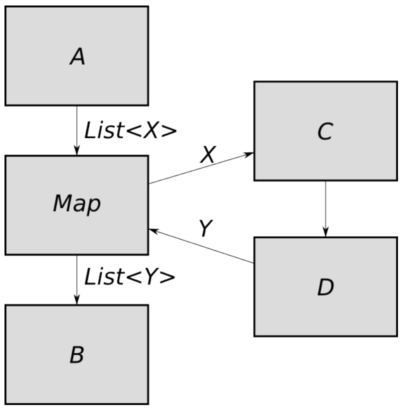


Figure 3.3: Example for node constraints: the *Map* node requires that a path exists from its processing output to the processing input (*Map* $->$ *C* $->$ *D* $->$ *Map*), and no nodes on that path violate the cardinality and order constraints.

Since map applies a function to each element of a list, the map node has to rely on a number of semantic requirements. A path has to exist in the direct graph connecting map's element output to map's element input. Furthermore, for each element passed to the processing output, an element has to be made available on the result input. Finally, the order of elements resp. their derived elements has to be preserved.

These requirements are expressed by using *constraints*. A constraint is an attribute used for decorating a node type class. It usually contains the names of two endpoints the constraint has to be applied to[1]. Node types that violate the semantics of a constraint are marked with corresponding (mostly empty) interfaces.

The following constraint/interface combinations exists:

ConnectedLoop Both given endpoints have to be connected in a way that a path exists from one endpoint to the other. This constraint relies on analysing the workflow graph itself, and no interface is accompanied to it.

[1]At this point the developer has to address the fields by their name, since their content may not be available yet.

OrderConstraint The order of elements passed along the path between the endpoints may not change.

CardinalityConstraint The number of elements sent to a loop and the number of elements read from the loop may not differ. No elements may be removed from or inserted into the data stream.

FlowConstraint All nodes along the path between the endpoints are expected to process data immediately, and may not block processing or collect data for batch processing. In contrast to the other constraints, the corresponding interface defines a method to disable blocking in affected nodes. in particular node types using batch processing may fall back to process data immediately, although this may cause a severe performance loss.

The processing component checks all constraints by traversing the referred sub graph, and searches for nodes that implement the corresponding interface. If a constraint is not fulfilled, the execution of the workflow is denied.

3.7.4 Life cycle

Figure 3.4 depicts the life cycle of a node in a Conveyor workflow. Transition between the states of the life cycle is done by invoking one of the methods below:

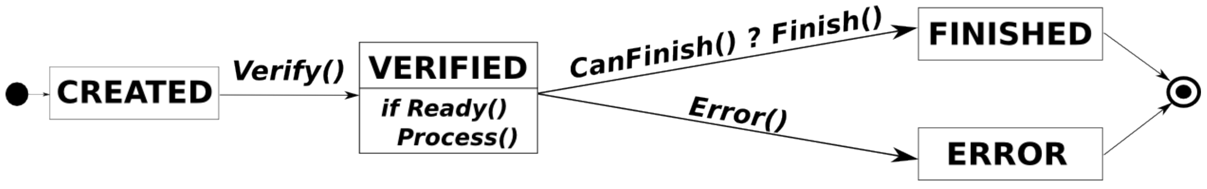


Figure 3.4: Lifecycle of Conveyor nodes: After creation and configuring the node (e.g. setting configuration values and links to other nodes), the *Verify()* method validates the configuration. Upon successful validation the node enters the VERIFIED state and is able to process data using the *Ready()* and *Process()* methods. The lifecycle is terminated either by calling *Finish()* to indicate normal termination and changing to the FINISHED state, or by calling *Error()* in case of an exception during processing.

Verify() Checks that a node in a workflow is properly configured and ready for execution. This includes a) that all endpoints of the node are connected, b) the data types defined for the endpoints are compatible, and c) all mandatory configuration fields are set. Derived node types may overwrite and extend this method, e.g. to check their configuration fields for reasonable values, open files, connect to a database, or to perform any operation necessary to setup a node.

Ready() Checks that the node may process data.

Process() This method does the actual data processing by reading data from incoming endpoints and writing results to outgoing endpoints.

CanFinish() Checks whether a node should terminate.

Finish()/Error() Being the last steps in the life cycle, these methods are used to release resources claimed by a node instance.

The abstract base classes for node types already contain a default implementation for all methods except *Process()*. The default *Verify()* method checks that a) all endpoints of the node are connected, b) the data types defined for the endpoints are compatible, and c) all mandatory configuration fields are set. In case of *Ready()*, a node is considered ready for execution if data is available at all incoming endpoints. In a similar way, *CanFinish()* terminates the execution of a node if any of its incoming endpoints will not provide data anymore, or any of its outgoing endpoints will not consume data anymore. *Finish()* and *Error()* only handle the state transistion and necessary cleanup.

No default implementation of *Process()* is given in the abstract base classes. A minimal node type only needs to define its endpoints and implement the *Process()* method if the default implementation of the other method fits its semantics.

3.8 Definition component

As described in the previous sections, Conveyor is designed around reflection and introspection for collecting information about types at runtime. The definition component bundles the functionality used to gather type information, and provides information about available node and data types.

In addition to descriptions and structural information, the node type representation defined by the definition component also creates the node instances used in an actual workflow. This allows a separation between the node type information and the actual class implementing the node type. An example is shown in chapter 4.

3.9 Processing component

The processing component serves two different tasks:

Setup and information Initializes Conveyor, loads all configured plugins, and provides methods to retrieve information about available data and node types.

Execution Instantiates workflows, starts their execution, and monitors its progress.

Clients do not refer to plugins or the definition component directly, but use methods provided by the processing component. The details about plugin loading are hidden by this component.

To allow Conveyor to adopt to different use cases, another layer hides the details of the workflow execution from the processing component. An interface defined by this layer is used to exchange messages. Which implementation is used to execute a workflow is either controlled by a factory in the processing component, or controlled by the client using it. The design of the node life cycle described above and the processing component itself enables various processing models to be used in Conveyor, including simple simple iteration over nodes to multithreaded or cluster-aware implementations. Details are given in chapter 4.

3.10 Clients

The components described so far implement the workflow engine, encapsulate functionality in plugins and provide methods to access the Conveyor system. They do not provide workflows yet, which is the task of the clients. A Conveyor client uses the processing engine to query information about available data and node types, creates a workflow, and finally uses Conveyor to execute it. The client itself is outside the scope of the Conveyor workflow engine; the engine is just used by the client to execute tasks. Several clients have been designed and implemented for Conveyor, from a simple commandline utility for executing predefined workflows, over a solution for offering workflow design and execution as a service to complex applications for data analysis.

CHAPTER 4

Implementation of Conveyor

The design presented in the previous chapter defines some requirements the implementation environment has to fulfill:

Object oriented type system At least one object oriented programming language has to be supported by the environment since Conveyor's design is based on features like interfaces and inheritance.

Generics support The programming language should also support generic classes to encourage node type reuse.

Full reflection support All information about libraries and their contained types has to be available at runtime, including information about generics and their type constraints.

Easy integration of third-party applications and libraries Third-party applications and libraries written in a different programming language should be integratable natively.

Of the available environments Microsoft's .NET runtime[1] and Oracle's Java[2] meets these requirements best. In the end .NET (resp. Mono, a free UNIX(tm) implementation of .NET) was chosen based on two criteria:

- Although Java supports reflection, the type parameter information of generics is discarded by the Java compiler. Thus it is not possible to retrieve type constraints on runtime.
- .NET has native support for a number of different programming languages like Python, Ruby or Java (via IKVM[3]). At the time of designing Conveyor, the Java runtime was limited

[1] http://www.microsoft.com/net/
[2] http://www.oracle.com/technetwork/java/index.html
[3] a open-source project for embedding a Java runtime environment into the .NET runtime allowing the execution of native Java code

to Java, Scala, and Closure.

The .NET framework comes with its own high-level language called *C#* which is used to implement Conveyor. This chapter describes the details of the various central Conveyor libraries. It also contains information about various optimizations for both performance and developing overhead, and gives an overview of the plugins implemented for bioinformatics data processing.

4.1 Conveyor.Core library

Being the base for all Conveyor plugins, the *Conveyor.Core* library implements all base node types and classes needed to define a Conveyor node or data type. It is the only single requirement a plugin needs.

4.1.1 Conveyor.Core.Data

The interface *Conveyor.Core.Data* is the marker interface described in chapter 3. All data types Conveyor is aware of are implementing this interface.

4.1.2 Conveyor.Core.Node

The classes *Conveyor.Core.Node*, *Conveyor.Core.NodeInput*, *Conveyor.Core.NodeOutput*, and *Conveyor.Core.NodeProcessing* form a hierarchy of abstract base classes as shown in figure 4.1. *Conveyor.Core.Node* contains the default implementation of the node life cycle as depicted in section 3.7.4. It also provides access to configuration items, the unique identifier of a node within a workflow, state management, and cleanup methods. A self-written node type must not derive directly from this class.

The three derived classes *Conveyor.Core.NodeInput*, *Conveyor.Core.NodeInput* and *Conveyor.Core.NodeProcessing* act as base classes for input, output and processing node types. Among differentiation into one of the three sections, they also contain default implementations of life cycle methods not provided by the base class. Especially the input and output node classes define additional methods which sub classes have to implement. These methods are either used to provide more information about the node in question, e.g. the type of output generated by an output node, or to simplify implementation of derived classes, as in the case of the input node's base class.

4.1.3 Conveyor.Core.LinkEndpoint

Endpoints are defined as fields in a node type class. The endpoint classes form the type hierarchy presented in figure 4.2. The abstract base class *Conveyor.Core.LinkEndpoint* contains functionality shared by all endpoints like state, identifier, containing node, etc. The hierarchy then splits

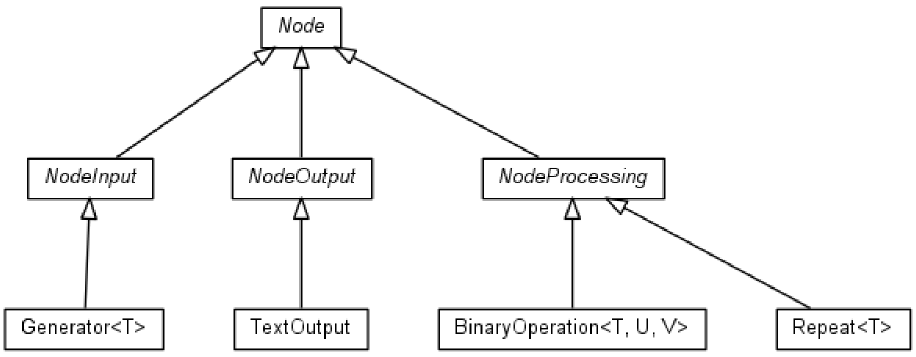


Figure 4.1: Class hierarchy of the Conveyor node base classes, including some examples for derived concrete and abstract classes in the *Conveyor.Core* component.

into two branches for input and output endpoints. Each branch contains an abstract non-generic class and a derived generic class. The later ones are the types actually used in a node type for defining endpoints. The non-generic base classes are helpers for reflection. They allow to retrieve all endpoints of a node type (fields derived from *LinkEndpoint*) or, more specifically, all input or output endpoints (fields derived from *InputEndpointBase* or *OutputEndpointBase* resp.). Endpoints are initialized in the constructor of the *Conveyor.Core.Node* class, using reflection for enumerating all endpoint fields in a derived class. It also collects them for later use in the life cycle methods. The single type parameter of the input and output endpoints define the data type passing an endpoint, and the methods for getting and sending data objects use this parameter in their signature. Using endpoint fields is thus type safe.

4.1.4 Conveyor.Core.ConfigBaseItem

As described in chapter 3, node instances in a workflow need to be customizable. This is made possible by defining fields of one of the classes derived from *Conveyor.Core.ConfigBaseItem*. Table 4.1 lists the concrete configuration classes and their associated data types.

Similar to endpoints, *Conveyor.Core.Node*'s constructor takes care for initializing the fields and sets default values if necessary. It also provides the methods for setting a configuration field's value. Node types defining configuration fields should also overwrite the *Verify()* method of the node life cycle to check the fields for correct values prior to using them.

4.2 Conveyor.Definition library

As pointed out in chapter 3, the *Conveyor.Definition* component acts as a connector between the processing engine and the plugins itself. It decouples both entities, and thus allows *Conveyor*

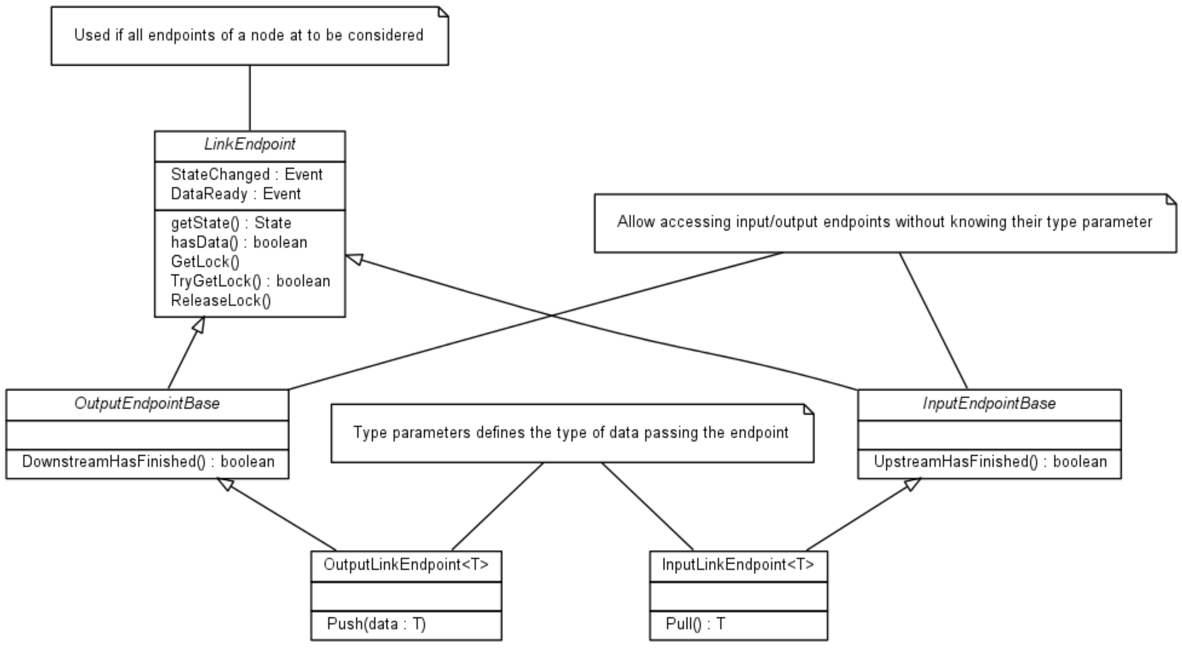

Used if all endpoints of a node at to be considered
LinkEndpoint
StateChanged : Event
DataReady : Event
getState() : State
hasData() : boolean
GetLock()
TryGetLock() : boolean
ReleaseLock()
Allow accessing input/output endpoints without knowing their type parameter
OutputEndpointBase
DownstreamHasFinished() : boolean
Type parameters defines the type of data passing the endpoint
InputEndpointBase
UpstreamHasFinished() : boolean
OutputLinkEndpoint<T>
Push(data : T)
InputLinkEndpoint<T>
Pull() : T

Figure 4.2: The hierarchy of endpoint classes define both generic and non generic classes to both type safe usage and easy retrieval via reflection.

Class	*Description*
ConfigBoolean	boolean value
ConfigByte	8 bit unsigned integer
ConfigSByte	8 bit signed integer
ConfigInteger	32 bit signed integer
ConfigULong	64 bit unsigned integer
ConfigLong	64 bit signed integer
ConfigDouble	IEEE double precision value
ConfigString	UTF-8 string
ConfigFile	a stream read as file
ConfigEnumeration<T>	one of the values of the enumeration type T
ConfigSelection<T,U<T>>	one of the values of type T, generated by type U

Table 4.1: Configuration field types available in Conveyor nodes. By convention the class name is composed of the prefix *Config* and the data type the class encapsulates, e.g. *Byte* or *String*. The enumeration type uses the values from the enumeration type given as type parameter, eliminating the need for comparing string values in the node code. The selection type provides a more generic way to define a selection of values. The first parameter is the item type, the second parameter a class generating these items. Selection types are used for example to provide a list of BLAST databases by scanning certain directories.

to be extended by new plugins.

The following subsections describes the implementation of the *Conveyor.Definition* component for retrieving information about available node types and creating instances of nodes.

4.2.1 Retrieving information

Plugins are implemented as .NET libraries and may contain any number of data type and node types. In the previous section 4.1 a data type has been defined to be any class that implements the *Conveyor.Core.Data* interface, and each node type has to be derived from the *Conveyor.Core.Node* class or one of its subclasses. The reflection mechanism built into .NET allows a simple retrieval of information about all types defined in a library[4], including their visibility, fields, properties, methods, etc.

The default mechanism for scanning assemblies only takes publically visible types into account. Each type is checked for its base classes and implemented interfaces and *IDataType* resp. *INodeType* instances are collected that represent the detected types. *IDataType* contains information about a data type, including its name and description. *INodeType* represents a node type usable in *Conveyor*. It exposes the node type's name, description, type parameters, endpoints, configuration items, etc.

Figure 4.3 schematically shows the scanning process and the resulting objects created by the *Conveyor.Definition* component. The *Plugin* objects are exported to the *Conveyor.Processing* component and thus available to a client using *Conveyor*.

4.2.2 Creating node instances

The second crucial task performed by the *Conveyor.Definition* component is the creation of node instances during workflow execution. A workflow definition refers to a node type by its name. Upon setting up a workflow for execution, each node contained in the workflow has to be instantiated by the processing component. It in turn uses the *INodeType* instance representing the node type to create a node instance. *INodeType* defines two methods for creating node instance:

Node CreateInstance(long id) Creates a node instance of a non-generic node type.

Node CreateInstance(long id, List<Type> typeParameter) Creates a node instance using the given list of type parameters.

The way instances of generic types are created at runtime impose a limitation on the way *Conveyor* works. All type parameters have to be known to create a node instance. The *Conveyor Designer* presented in section 4.8.1 takes care for resolving all type parameters for a node instance. Instantiation at runtime fails if the given type parameters does not match the type constraints or the number of parameters is wrong.

[4]referred to as *assembly* in .NET

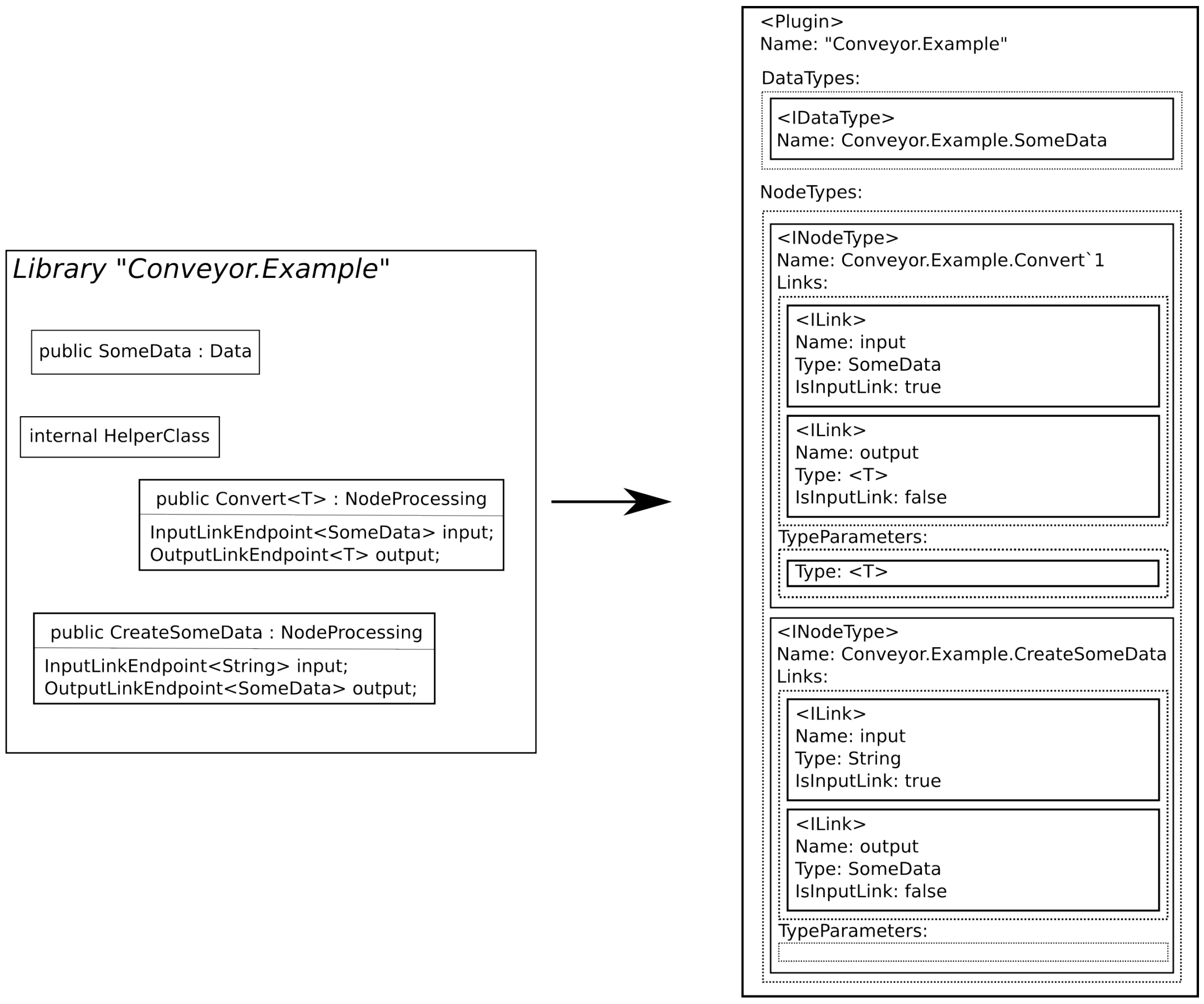


Figure 4.3: Default processing of a library by the *Conveyor.Definition* component. A *Plugin* instance is created for the library, refering to the library's name and its description. For each data type found in the library, an *IDataType* instance is created containing information about the data type. In a similar fashion an *INodeType* instance is setup for each node type, refering its fully qualified name, description attribute content, type parameter, link endpoint information, configuration item overview, etc.[a] [b]

[a] Some aspects of node type definitions are not included for the sake of clarity.

[b] Fully qualified type names contain an indicator for the number of type parameter. In the figure, *Converter<T>* in library *Conveyor.Example* has the fully qualified type name *Conveyor.Example.Converter‘1*

Although *INodeType* usually represents a concrete node type, *Conveyor.Definition* is designed and implemented with flexibility in mind. Section 4.6.2 describes another approach for defining node types and their instantiation at runtime.

4.3 Conveyor.Processing

The most important library for applications using Conveyor is *Conveyor.Processing* as it provides all necessary functionality for clients to actually use Conveyor. This includes initialization, loading of plugins, enumeration of plugins and contained node and data type, and finally the representation and processing of workflows. The *Conveyor.Processing.Graph* class represents a workflow by managing a list of node instances. It may either be built programmatically by a client or by using an instance of the *Conveyor.Processing.GraphParser* class that parses a workflow description from a XML file. Since Conveyor may be used in different environments with different resources, the actual processing of a graph is decoupled from the *Conveyor.Processing* library and abstracted by the *Conveyor.Processing.IProcessor* interface. It defines the necessary methods for validating a graph, asynchronous execution and monitoring, and result retrieval.

An overview of the methods is available in appendix A. The actual methods for execution is not specified, and each implementation may choose to execute a graph depending on available resources. It only has to adhere to the life cycle of nodes as depicted in section 3.7.4. A client may select the implementation by itself or use the default configured for a Conveyor installation. The following section describes the *Conveyor.Processing.IProcessor* implementations available in Conveyor.

4.3.1 IProcessor implementations

Since parts of the functionality available in *Conveyor.Processing.IProcessor* instances is the same between several instances, the abstract base class *Conveyor.Processing.ProcessorBase* provides a default implementation. Among instance construction, graph setup and graph information retrieval methods, it also contains a default implementation of the validation step, which simply calls the *Verify()* method for each node instance and handles exceptions in the correct way. All execution related methods are marked as abstract and have to be implemented by concrete implementations. The next sections describe the set of implementations already available in Conveyor.

SingleNodeProcessor

This is the most straight-forward implementation of a Conveyor processor. To prevent clients from becoming unresponsible during workflow execution, the processing logic is offloaded to a worker thread. The thread's code is shown in listing 4.1. Extra code (e.g. logging and handling of clients' abort requests) is removed for the sake of clarity. The processing is implemented by iterating over the elements of a queue of nodes until no more nodes are present in the queue. The queue contains those nodes of a graph that have not finished yet. Initially, the queue is filled with all nodes of the graph (line 1). Within each loop iteration, one node is taken from the queue (line 6) and is examined. If the node has finished, the next iteration of the loop is started without re-adding the node to the queue (line 7-11). Although not explicitly stated in the node's life cycle description, a node may also finish during its validation in the *Verify()* method. By checking for finished nodes prior to processing them, these nodes are removed from the queue at the next opportunity. The remainder of the loop handles node processing (line 13-18) and error handling (21-27). In case of an error during processing, all nodes of the graph are set to the error state with the current node marked where the exception occured. A client may later on examine the error state of the nodes and perform appropiate error-handling.

Processing is done by checking whether a node is able to process using the *Ready()* method (line 13), and invoking the *Process()* method on a positive response. If the node is indicating its ability to terminate the processing by returning a true value in the *CanFinish()* method, the node is shut down properly using the *Finish()* method, and the next iteration of the loop starts. Otherwise the node is put back into the queue for a new processing iteration. Taking the validation phase of the *Conveyor.Processing.ProcessorBase* base class into account, the complete life cycle of Conveyor nodes is implemented in this processing loop, while keeping the processor as simple as possible.

FastSingleNodeProcessor

While designed for simplicity instead of maximum performance, a small number of trivial modifications may be used to improve the overall performance of the *SingleNodeProcessor*, resulting in a new processor implementation called *FastSingleNodeProcessor*. One performance problem results from the fact that the queue contains all unfinished nodes of the graph at any moment although only a part of it may be ready for processing at all. In the worst case finding these nodes requires iterating through the whole queue. As a solution, an optimized implementation may ensure that only processable nodes are stored in the queue. Instead of adding the current node back to the queue in the loop, it is only added back if *Ready()* indicates that further processing is possible. In addition, all nodes connected to output endpoints of the current node need to be checked, too, since their readiness to process data might have been changed during processing of the current node (e.g. by sending data to the connected node). If the queue is initially filled with processable nodes only, the number of pending nodes in the queue is reduced. Nonetheless, it has to be ensured that a given node is added to the queue only once.

Another performance penalty occurs based on the amount of processing done in a single loop iteration. *Process()* is only called once, although more than one call might be possible. Using a *while*-loop instead of a single check in line 13 removes this bottleneck. Both optimizations are implemented in the *FastSingleNodeProcessor* class. Although the performance of this implementation is better than *SingleNodeProcessor* in most cases, some drawbacks of this attempt exist:

- *Ready()* is called more often. Complex implementations of *Ready()* may thus lead to a performance decrease.

```
Queue<Node> nodes = new Queue<Node> (Graph);

// iterate over the list, execute nodes if possible
while (nodes.Count != 0) {
        // get next node to process
        Node node = nodes.Dequeue ();
        if (node.IsFinished ()) {
                // we need to handle nodes that are already finished
                // during its Verify() method execution
                continue;
        }
        try {
                if (node.IsVerified () && node.Ready ())
                        node.Process ();
                if (node.CanFinish ()) {
                        node.Finish ();
                        continue;
                }
        }
        catch (Exception ex) {
                node.Error (ex);
                // mark all other nodes as aborted
                foreach (Node n in nodes) {
                        n.Error ((Exception)null);
                }
                SetState (GraphState.ERROR);
                return;
        }
        // this node is not finished yet, re-add it to the queue
        nodes.Enqueue (node);
}
SetState (GraphState.FINISHED);
```

Listing 4.1: Main processing loop of SingleNodeProcessor

- Some nodes may process large amounts of input data, and store large amounts of data passed to another nodes. This may result in increased memory requirements.
- Checking all nodes connected to a given node is tricky since this information is not exposed by the endpoints, and thus requires code that is hard to maintain.

A client may decide from case to case which processor is better suited for a workflow, or a Conveyor administrator may choose one of them as default for a Conveyor setup.

ThreadedProcessor

The *SingleNode* and *FastSingleNode* processor implementations described above are focussed on simplicity. However, with the advent of dual, quad and multi-core CPUs in the last decade, workflows should start to exploit these features and use multiple threads for execution. The *Conveyor.Processing.ThreadedProcessor* implementation of the *IProcessor* interface starts one thread per node, and takes care for the necessary synchronization between the threads. Synchronization is required if two or more threads interact. Although the nodes itself may work independently of each other, they have to exchange data with each other and have to react to state changes of connected nodes. The design of Conveyor as presented in chapter 3 does not explicitly take multithreading and related topics into account. Nonetheless, nodes are expected to behave correctly in a multithreaded environment, in particular if several instances of a node class exist and are executed in parallel. The *ThreadedProcessor* spawns one thread for each node, passing the node as thread start argument. In the most simple approach a thread would use the following loop, assuming the node is already validated using *Verify()*:

```
while(true) {
  // terminate loop if node is finished
  if (node.IsFinished())
    return;

  // process node if possible
  if (node.Ready())
    Process();

  // finish node if necessary
  if (node.CanFinish())
    Finish();
}
```

(Error handling and thread terminating are not shown for the sake of simplicity.)

Two problems may occur if this loop is used for multithreaded processing of a workflow:

- Each thread wastes CPU time if nodes are not able to process data at the moment. For large workflows more CPU time is wasted than actually used for processing due to a large number of threads running in parallel.
- The state of connected nodes may change between or during calling *Ready()* and *Process()*.

This will result in undefined behaviour, e.g. sending data to a finished node.

The following subsections address these problems and describe how the *ThreadedProcessor* resolves them.

Notification vs. polling

The technique of querying a state over and over in a loop is called *polling*. In a multithreaded environment polling should be avoided due to the consequences described above. Whether a node is able to process data or to finish solely depends on the state of its endpoints. Changes to their state should trigger further processing of the node, or allow it to terminate. This technique is called *event-based processing*; an external, asynchronous event is used as trigger. The thread handling a node may suspend itself until it is triggered.

Two features of the .NET runtime are used to implement an optimized version of *ThreadedProcessor*:

Event & Delegate A *delegate* is a definition of a method signature. Each method (either an instance method or a class method) may be used as a delegate if its signature matches. An *event* is a member of a class implementing a collection of delegates with the same signature. New delegates may be registered by other objects or removed if the object in question is not active anymore. Events may be raised by the enclosing class, resulting in each delegate being invoked. Both concepts allow a very loose coupling of classes by simply connecting delegates to the corresponding events.

WaitHandle A *WaitHandle* is used to synchronize to external events, either triggered by other threads or by the operating system itself. A thread waiting for a WaitHandle is suspended until the handle is triggered.

Using both techniques solves the problem of polling and changes the processing loop to event-based processing. The endpoint classes are extended by events signaling a state change and data availability (*DataReady* and *StateChanged* in figure 4.2). The processing loop is changed to the following pseudo code:

```
WaitHandle handle;
<register callback to trigger handle at endpoints' events>
while(true) {
  // terminate loop if node is finished
  if (node.IsFinished())
    return;

  // process node if possible
  if (node.Ready())
    Process();

  // finish node if necessary
  if (node.CanFinish())
    Finish();
  <suspend in wait for waithandle>
}
```

Enforcing atomicity

Another problem is the lack of *atomicity* of compound statements. During each single statement or sub statement the overall environment may change. An *atomic* operation in contrast is guaranteed to be executed without an external changes. Atomicity can be enforced for compound statements by the introduction of *lock*s. Every operation changing the environment has to acquire all necessary locks prior to execution, and has to release the locks afterwards. A complete introduction to locking is beyond the scope of this thesis. More Information can be found in e.g. in [Ste92].

In the case of Conveyor, a node depends on the states of its connected nodes. During calls to *Ready()*, *Process()*, *CanFinish()*, and *Finish()* the connected nodes may not alter their state. The processed node itself may change its state under these circumstances. The definition of nodes and endpoints do not expose the connected nodes to the *ThreadedProcessor* since the corresponding fields are not visible outside the core library. In addition an output endpoint may be connected to any number of nodes. Thus, instead of locking the nodes itself, the endpoint classes are extended by a method to return a lockable object. It is shared between an output endpoint and all of its connected input endpoints. To ensure atomicity, a processing thread has to acquire all locks of all its node's endpoints prior to using any of the node's methods listed above. The locking code (not shown) has to ensure that no deadlock will occur.

The overall processor loop is thus expanded to:

```
WaitHandle handle;
<register callback to trigger handle at endpoints' events>
while(true) {
  // terminate loop if node is finished
  if (node.IsFinished())
    return;

  <get locks>
  // process node if possible
  if (node.Ready())
    Process();

  // finish node if necessary
  if (node.CanFinish())
    Finish();

  <release locks>

  <suspend in wait for waithandle>
}
```

4.4 Plugins

The libraries presented so far only implement the workflow engine and all base definitions needed for data types and node types. The available node types in the core library only offer fundamental functionality.

This design makes the Conveyor core and processing components independent of the data domain. All specific functionality and specific data types are implemented using *Plugin*s. From the technical point of view a plugin is a .NET library that is loaded at runtime and contains classes implementing data and node types. The *Conveyor.Definition* component described above takes care of examining each plugin library and extracting all relevant classes.

Table 4.2 lists additional, non-domain-specific plugins that the author has developed for Conveyor. They provide wrapper for primitive data types, control flow operations, and generic collection types.

Name	**Description**
Conveyor.Comparison	generic results of ordered comparison (larger, equals, less)
Conveyor.Dictionary	key-value based generic dictionary type and processing nodes
Conveyor.List	generic list type
Conveyor.Logic	boolean processing and control flow nodes
Conveyor.Math	wrapper for all standard numeric types and operations on them
Conveyor.Pair	generic pair composed of two elements
Conveyor.Text	strings and string processing

Table 4.2: Default plugins available in Conveyor. These plugins implement a number of primitive data types, control flow operation and collection types.

4.5 Performance optimizations

The design and implementation of Conveyor focusses on functionality and type safety. The node classes and processor implementations are clearly designed and offer the flexibility to be used for almost any functionality. Performance has not been the primary goal goal of this implementation yet.

To evaluate the performance, a small workflow shown in figure 4.4 has been designed. It reads a number of lines from a text file and sorts the lines according to their lengths using a node type implementing the quicksort algorithm. It was chosen for benchmarking since all node instances of the workflow belong to the *Conveyor.Core* component.

The benchmark workflow was executed with a limit of 10,000 lines using both the *FastSingleNodeProcessor* and the *ThreadedProcessor*. Execution time was measured using the UNIX *time* command on a host with 4 Intel Xeon E7540 CPUs and 256 GB RAM, running Solaris 10 for x86

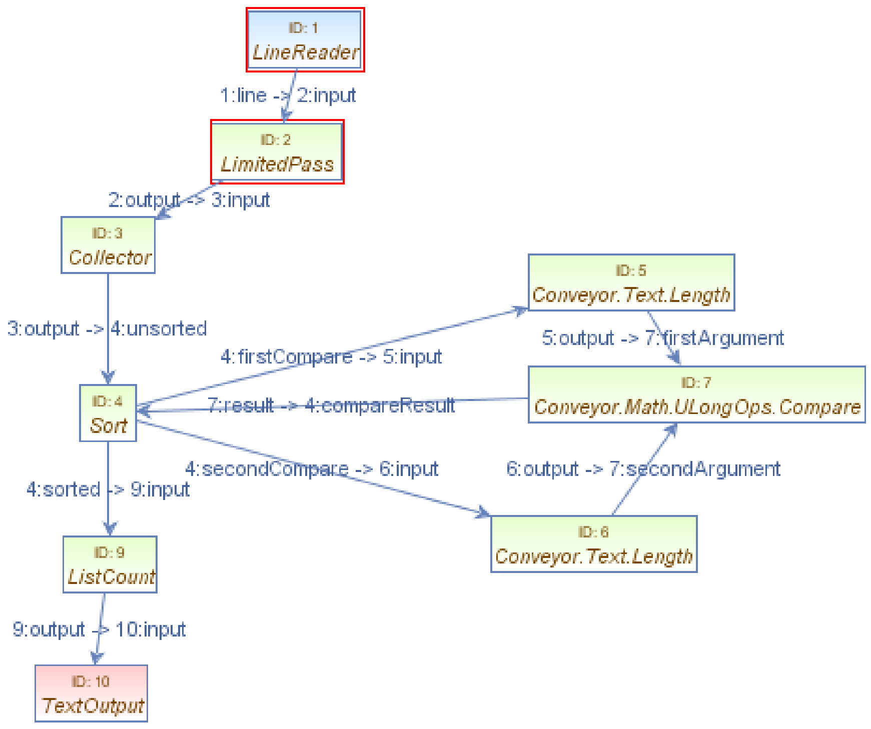


Figure 4.4: Sample workflow used for benchmarking Conveyor and its optimizations. A number of lines is read from a file and collected in a list of strings (nodes 1 - 3). Afterwards the list is sorted according to the lengths of the lines (nodes 4 - 7). The number of lines is finally printed as output of the workflow (node 9 + 10). The read boxes around nodes 1 and 2 indicate that some configuration values are missing for these nodes: the input file in the case of node 1, and the number of lines to be processed in case of node 2. These parameters depend on the benchmark setup and are set prior to execution.

and Mono 2.8.1. The results are available in table 4.3.

Processor	Number of lines	Execution time [s]		
		Real	User	System
FastSingleNodeProcessor	10,000	32.274	30.684	0.238
	20,000	121.561	120.096	0.270
ThreadedProcessor	10,000	505.445	864.605	152.091
	20,000	1968.219	3352.663	592.521

Table 4.3: Runtime of the benchmark workflow for various number of lines read from a multi entry GenBank file. The first column shows with processor implementation was used in a run, the second number contains the number of lines read in that benchmark run. The other three columns contain the results of the UNIX *time* command, transformed to seconds.

The performance of the *ThreadedProcessor* is seriously degraded compared to the *FastSingleNodeProcessor*. Especially the parallel pathes in the comparison loop of the *Sort* node should result in a faster execution time. The ratio of user time to real time indicates that in fact threads are running parallel. The high amount of processing time spend in system calls gives a first hint on the cause of the bottleneck.

4.5.1 Beyond time(1): Detection and quantification of bottlenecks

The *time* command gives useful information about the overall performance of an application. To actually identify the bottleneck resulting in degraded performance of the *ThreadedProcessor*, the information provided by *time* is not sufficient.

Recent versions of the Solaris operating system used during the development and benchmarking of Conveyor include a number of tools for monitoring processes and gathering statistics. *prstat* is able to collect information about each thread and the states it passed through. A excerpt of the output taken during running the sample workflow with the *ThreadedProcessor* is shown in figure 4.5.

Two of the threads (LWPID 1 + 2) are part of the mono runtime and the application executing the workflow. The remaining threads refer to the various node instances in the sample workflow. Since the output was taken some minutes after the process was started, the first nodes in the sample workflow are already terminated (node 1 - 3). LWPs 3, 7, 8 and 9 are the threads processing the *Sort* node and its comparison loop, LWPs 10 + 11 are the pending threads for the final output node instances (Node 9 + 10 in the sample workflow). These nodes are blocked while waiting for data to process.

Concerning the active threads, a fair amount of time is spent in the locked state (34% - 54%, see column *LCK* in the output), and less than half the time is used for processing and executing the processing loop (*USR* column). The threads are blocking each other while competing for the lock that ensures atomic processing as explained above.

```
  PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
13132 blinke    48  10 0.0 0.0 0.0  34 1.5 5.4 .1M  23 .5M  17 mono.bin/3
13132 blinke    47 9.5 0.0 0.0 0.0  36 1.6 5.3 .1M  21 .6M  18 mono.bin/7
13132 blinke    38 6.1 0.0 0.0 0.0  51 1.6 3.5 81K  21 .2M  19 mono.bin/8
13132 blinke    36 4.9 0.0 0.0 0.0  54 1.4 3.8 79K  18 .2M  19 mono.bin/9
13132 blinke   0.4 0.1 0.0 0.0 0.0  97 2.3 0.0  49   0 237  24 mono.bin/1
13132 blinke   0.0 0.0 0.0 0.0 0.0  98 2.0 0.0  25   0  81  25 mono.bin/2
13132 blinke   0.0 0.0 0.0 0.0 0.0  98 2.0 0.0  28   0 124  24 mono.bin/10
13132 blinke   0.0 0.0 0.0 0.0 0.0  98 2.0 0.0  29   0 127  24 mono.bin/11
```

Figure 4.5: Part of the output of the *prstat* command for running the sample workflow with the *ThreadedProcessor*. One line descrines one thread in the target process. The *USR* to *LAT* columns stand for the various states a thread may take on in the operation system. The numbers in that column indicate the percentage of time a thread has spent in the corresponding state during one sampling interval. A more detailed explanation can be found in the the *prstat* man page.

To quantify the lock contention occuring while using the *ThreadedProcessor*, the so called performance counters available in the Mono runtime are used. A performance counter provides information about various aspects of a running .NET/Mono process, including memory and thread management. Figure 4.6 shows a screen shot of the *mperfmon* tool, which is able to attach to a running Mono process and reads and visualizes performance counters. The screenshot shows the counter for the lock contentions per second.

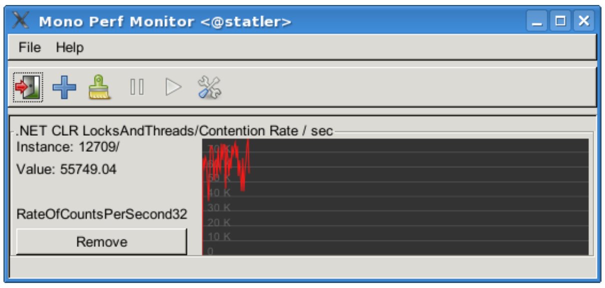

Mono Perf Monitor <@statler>
.NET CLR LocksAndThreads/Contention Rate / sec
Instance: 12709/
Value: 55749.04
RateOfCountsPerSecond32

Figure 4.6: The screenshot shows the user interface of the *mperfmon* tool while monitoring the execution of the sample workflow using the *ThreadedProcessor*. The active graph visualizes the lock contentions per second value as a performance counter.

The *GraphRun* tool (see section 4.8.2) has been extended to report performance counters upon completion. In the end, an average value for lock contention can be calculated using the available counter for the total number of contentions and the runtime of the process. Table 4.4 shows the lock contention values for the *ThreadedProcessor* for different input sizes.

Despite all advantages, and As explained in section 4.3.1 describing the *ThreadedProcessor*,

Lines processed	Processing time (real, seconds)	Total # of lock contentions	Contentions / second
1,000	10.6	456,519.4	43,067.9
2,000	34.3	1,617,366.6	47,153.5
5,000	159.7	9,920,537.8	62,119.8
10,000	572.4	36,326,829.0	63,464.1
20,000	1,918.8	127,884,977.4	66,647.6

Table 4.4: The table shows the average lock contention rate of the default *ThreadedProcessor* implementation. The figures are arithmetic averages for 5 independent runs.

locking is necessary to ensure the correct processing of node instances in a multithreaded environment. The following subsections present various optimization approaches to minimize the use of locks and lock contention, thus to improve the overall performance of Conveyor.

4.5.2 Immediate processing nodes

As explained in the previous subsection, the necessary locking used to synchronize threads processing node instances prevents Conveyor from scaling well with the number of nodes.

A first attempt to reduce the amount of lock contention has been made by simply to reduce the number of threads. For many nodes, the amount of time spent for actual processing is small compared to the overhead for locking and synchronization. An example is the *Conveyor.Logic.Not* node of the *Conveyor.Logic* plugin. It reads a boolean value from its input, negates it and writes the result to the output. Both input and output have to be locked during processing, the endpoints have to handle one pull and one push operation, etc. Negating a boolean value in contrast is a trivial operation.

Solution

The idea of *immediate processing* nodes is moving the processing step from a node to the input endpoint of a connected node. Figure 4.7 shows three nodes forming a simple pipeline with the middle node being an immediate processing node. The following figure 4.8 shows the same pipeline after transformation. The processing function of the former middle node has been merged to the input endpoint of the successor node, while maintaining type safety. The *DoProcessing()* method of the former middle node is called for every data object being passed to the input endpoint, processing and transforming it to the correct data type.

Immediate processing nodes are implemented by the abstract class *Conveyor.Core.ImmediateProcessingNode* of the *Conveyor.Core* component. A concrete implementation has to override the abstract processing method *DoProcessing*.

Candidates for immediate processing nodes have to fulfill a number of requirements:

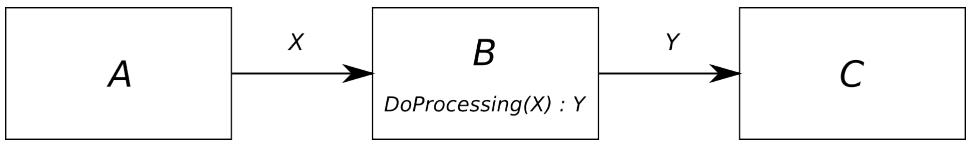


Figure 4.7: Example of an immediate processing node. Three nodes form a simple pipeline, with the middle node *B* being an immediate node.

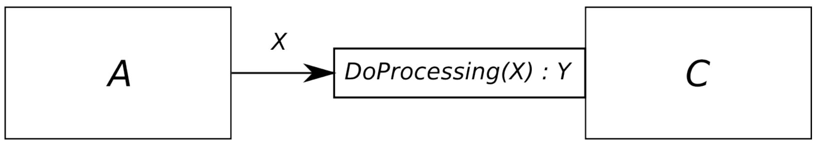


Figure 4.8: The transformation of immediate nodes eliminates node *B* completely, and its processing step implemented in the function *DoProcessing* becomes part of the node it was connected to (*C*).

- a single input endpoint and a single output endpoint
- very low processing requirements
- no life cycle related functionality

Since no thread or other controlling environment is used for an immediate processing node, the node is terminated using the *Finish()* method after its *DoProcessing()* method has been added to the destination endpoint and links have set up accordingly. Thus, no life cycle management is possible for immediate processing nodes, and the operation executed by the node may not use fields of the node class.

Immediate processing nodes have been used in the default Conveyor plugins where applicable.

Performance evaluation

The *Conveyor.Text.Length* nodes in the sample workflow are implemented as *immediate processing nodes*. Table 4.5 shows the average results for 5 runs of the sample workflow.

With respect to performance, the important nodes in the sample workflow are the sort node and the nodes belonging to sort's comparison loop. Since two of these 4 nodes are implemented as immediate processing nodes, enabling this optimized processing eliminates two of four threads. It cuts the processing time roughly in half, but contention remains on a high level due to the shared locks of the *Sort* and *Compare* nodes.

Since many low-level nodes in *Conveyor* are implemented as *immediate processing nodes*, the performance impact of this optimization depends on the number of immediate processing nodes used in a workflow instance.

Lines processed	Processing time (real, seconds)	Total # of lock contentions	Contentions / second
1,000	4.45	195,138.8	43,831.7
2,000	13.61	744,364.2	54,674.8
5,000	66.44	4,387,346.2	66,029.3
10,000	293.73	14,955,705.0	50,917.0
20,000	944.38	65,123,806.8	68,959.1

Table 4.5: The table shows the average lock contention rate of the default *ThreadedProcessor* implementation with immediate node processing enabled.

4.5.3 Lockless multithreading

Eliminating threads by using *immediate processing nodes* reduces lock contention, as shown in the previous section. But performance still leaves a lot to be desired, and again lock contention is the reason for it.

As explained in the section about the *ThreadedProcessor*, locking is needed to ensure that a node's methods are executied in a constant environment with respect to its connected nodes. This allows every node to act and terminate independently of other nodes. A good example for this is the *LimitedPass* node class in the *Conveyor.Core* component. It will pass data elements from its input to its output endpoint, but terminates after a configurable number of elements have passed. Figure 4.9 demonstrates the different termination impacts:

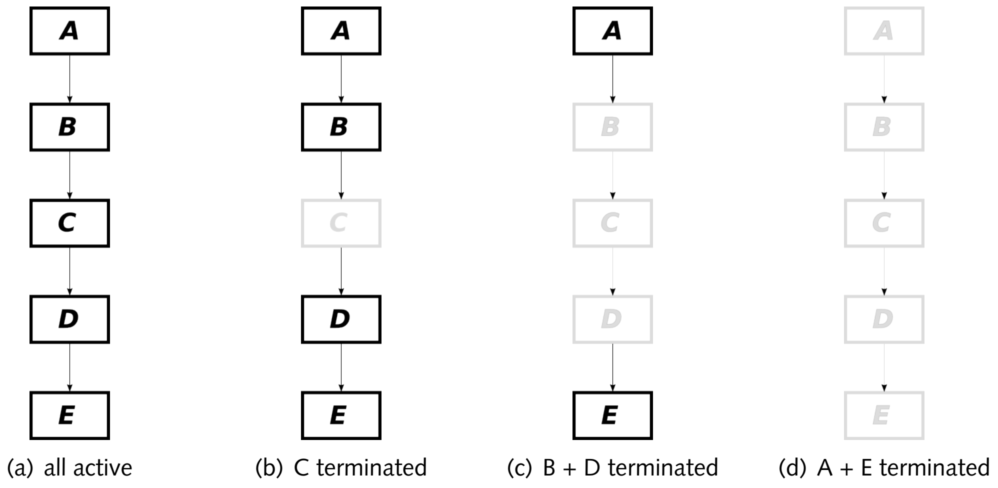


Figure 4.9: The figures show the different terminating orders, after node C has initiated termination. Nodes B and A are terminated *bottom-up*, while nodes D and E are terminated *top-down*, following the flow of data within the workflow.

top-down (nodes D + E) The nodes connected to input endpoints have terminated and no data is available in the endpoints' queues. The node thus should also terminate, triggering possible termination of the nodes connected to its output endpoints. As a result, the order of termination follows the logical flow of data between the nodes, starting at the input nodes and finishing with the output nodes.

bottom-up (nodes A + B) A node connected to an output endpoint has terminated. Since it will not accept data anymore on its input endpoint, the node has to terminate. In contrast to the previous mode, in this case termination goes against the logical flow of data, travelling from the initiating node to the input nodes.

Each node has to support both termination directions, and the default implementations of *Can-Finish()* already takes care of this. To ensure that especially the state of *downstream* nodes (nodes connected to output endpoints, thus following the flow of data) does not change during a processing iteration of the *ThreadedProcessor*, the processing loop has to use locking.

Solution

If a workflow does not include a node instance that triggers *bottom-up* termination, the termination order will follow the flow of data as explained above. Whether an input endpoint is able to provide data to the node depends on its internal buffer and on the state of the connected node. Changes to either trigger the next iteration of the processing loop as explained in the section about the *ThreadedProcessor*. Since the state of a connected node does not change after its termination, a node's processing loop does not require a fixed environment anymore. In the worst case a connected node changes its state during the checks in *CanFinish()*, which will trigger the next iteration of the processing loop. Locking is thus not required for these nodes.

The same assumption also holds for sub workflows. If all nodes *downstream* of a node may not terminate in a *bottom-up* manner, they do not require locking. This also means that as soon as a bottom-up terminating node is detected, all nodes upstream of it have to use locking.

Detection of bottom-up terminating nodes cannot be done automatically by code inspection or any other means. The developer has to mark these nodes, using the *OutOfOrderTermination* interface of the *Conveyor.Core* component.

Depending on the position of bottom-up terminating nodes in the workflow, a large part of the nodes may be operated in a lockless manner, reducing the overall lock contention and improving performance.

As a drawback of the lockless approach access to an input endpoint's buffer may happen asynchronously. All methods manipulating or querying the buffer have to ensure atomicity by using locks (again). Since the amount of code covered by these new locks is smaller than the endpoints lock's just eliminated, contention may only happen during a shorter period of time, and thus the performance impact is comparable lower.

Performance evaluation

Table 4.6 shows the performance values for the lockless multithreading approach combining with the immediate node processing approach. Although the amount of contention is still very high due to the locks introduced for accessing the endpoints' buffer, the overall processing time is significantly reduced.

Lines processed	Processing time (real, seconds)	Total # of lock contentions	Contentions / second
1,000	3.47	46,035.6	13,276.7
2,000	6.80	157,981.6	23,218.9
5,000	26.63	974,423.4	36,581.3
10,000	70.81	3,319,717.4	46,878.3
20,000	227.25	13,449,593.0	59,185.0

Table 4.6: The table shows the average lock contention rate of the default *ThreadedProcessor* implementation. Immediate node processing and lockless node handling threads are enabled.

4.5.4 Lockless queues

As mentioned in the previous section, access to an endpoint's buffer has to happen in a synchronized, locked way. Although the amount of code protected by the lock is rather small (and thus the time the lock is held), it affects the overall performance.

Solution

In the case of an endpoint's queue, there's exactly one reader and one writer to the queue. This scenario allows an easy implementation of a lockless queue, using atomic primitives offered by .NET (and based on atomic operations of modern CPUs).

Performance evaluation

With most of the locks eliminated, the overall processing time and lock contention is reduced dramatically as table 4.7 shows. The *LimitedPass* node still enforces locked operation of itself and the *LineReader* node but all other nodes operate in a lockless way. Since a large amount of processing time is spent in the *Sort* node and its processing loop, the average contention rate even drops with growing inputs.

Lines processed	Processing time (real, seconds)	Total # of lock contentions	Contentions / second
1,000	1.4	3,738.6	2,594.8
2,000	2.2	7,622.2	3,414.0
5,000	6.7	19,091.8	2,849.0
10,000	20.7	37,981.6	1,836.2
20,000	69.8	76,830.8	1,101.1

Table 4.7: The table shows the average lock contention rate of the default *ThreadedProcessor* implementation. Most locks are eliminated in this setup.

4.5.5 Asynchronous node types

Except for the *immediate processing node* optimization, all optimizations presented so far affect the engine itself. But among the single-input-single-output nodes necessary for *immediate processing nodes*, many other nodes also implement simple operations but require a different number of input or output endpoints. Most mathematical operation nodes for example have two input endpoints and one output endpoint. Similar to their *immediate processing node* cousins, a processing optimization for these nodes will also result in a noticeable speedup.

Solution

The abstract *AsynchronousNode* class in the *Conveyor.Core* components provides the base implementation for asynchronous node types. It registers a handler for the endpoints' data event, that implements the usual node processing (calling *Process()* if *Ready()* returns true). The node class is accompanied by a special processing class for *ThreadedProcessor* that implements the node life cycle logic without using a thread for node processing. If locking is required for a node instance handled by this class, it falls back to the default processing class based on a thread.

The processing overhead introduced by a processing thread and signaling between threads is eliminated by the *AsynchronousNode* class (if no locking is required). Since the processing class completely implements the node's life cycle control, a node class may use internal state in contrast to *immediate processing* nodes. Nonetheless, the processing method should consume only low CPU resources since it is executed on another node's processing thread.

Performance evaluation

The result of this final optimization step is presented in table 4.8. Compared to the previously run benchmark, the lock contention stays at the same level. Overall processing time is reduced by about 20%. The introduction of the *AsynchronousNode* class reduces the number of active threads during the sorting phase to one, since the comparison node is a prime candidate for an asynchronous node class.

Lines processed	Processing time (real, seconds)	Total # of lock contentions	Contentions / second
1,000	2.0	3,931.0	1,924.9
2,000	2.4	7,864.8	3,293.2
5,000	5.4	19,779.8	3,668.5
10,000	15.8	39,229.6	2,484.3
20,000	55.3	79,396.4	1,435.4

Table 4.8: The table shows the average lock contention rate of the default *ThreadedProcessor* implementation. All optimizations are enabled.

Conclusion

Comparing the *FastSingleNodeProcessor* results in table 4.3 with the results of the last optimization step, the overall processing time of the fully optimized *ThreadedProcessor* is reduced to less than the half. As mentioned in the previous sections, the effect of an optimization may vary with the workflow in question. Nonetheless, the sample workflow clearly demonstrates the effectiveness of each optimization.

Since *AsynchronousNode* is a recent addition to Conveyor, it has only been used for a limited number of node types in the core component yet. Especially more complex node types like *IfThenElse* may benefit from a conversion to asynchronous processing.

4.6 Development optimizations

While the previous section presents a number of performance optimizations for Conveyor, the following subsections focus on optimizing and simplifying the development process.

4.6.1 Special purpose base classes

A developer benefits from the object oriented nature of node types in Conveyor by using special purpose base classes. These classes usually implement a part of the node's life cycle or provide an easier interface for processing. As a result, the amount of code required for implementing a node type is reduced to an absolute minimum.

Input base classes

In addition the the default *NodeInput* class, some derived base classes exist in the *Conveyor.Core* component:

Generator<T> This simplification of the *NodeInput* class is intented for simple input nodes generating values of the same kind, e.g. the same string or a node for generating random numbers. The *InputDepleted()* method is overwritten to return false for any invocation.

SingleInput<T> Being intended for nodes providing a single value only, this base class implements the logic for ensuring that only one value is sent to the connected nodes. The value itself is generated by an abstract method that has to be implemented by derived classes.

SingleLinkInputNode<T> The design of input nodes requires them to wait until the buffer of at least one connected endpoint has run empty before the node may create the next data object. Otherwise, a node based on the *Generator* class described above will unconditionally flood its connected nodes with data objects.

To reduce the overhead involved in switching from the connected node(s) to the input node, the *SingleLinkInputNode* class offers a different processing mode. If only one node is connected to its single output endpoint, its input endpoint is modified to directly use the *InputDepleted()* and the new abstract *GetNext()* method of the *SingleLinkInputNode* class. Switching the processing task is thus eliminated.

If more than one node is connected to an instance of a node type derived from *SingleLinkInputNode*, the node uses the default behaviour without optimization. Developers are thus encouraged to derive all their input node types if applicable.

Output base classes

The default abstract output base class *NodeOutput* requires concrete classes to provide the output as a byte array. This allows Conveyor nodes to generate binary formatted results.

Many formats used in bioinformatics are text-based, and thus most output node types generate text files. The *TextNodeOutput* class is a specialized abstract output class that provides a *StreamWriter* property for writing strings to its output. It also manages the necessary life cycle actions like flushing the stream on termination and takes care of converting the written content to a byte array if the output is requested.

Processing base classes

ImmediateProcessingNode and *AsyncNode* have already been introduced in the previous sections when discussing performance optimization.

Another base class for processing node types is *BinaryOperation*. This class already defines two input endpoints, one output endpoint (with variable types due to type parameters), and the necessary implementation for *Process()*, *CanFinish()* and *Ready()*. If data is available on both endpoints, each is passed to the abstract method *Operation()*, and its return value is pushed to the output endpoint. Concrete classes thus only have to provide an implementation of *Operation()*, reducing the overall amount of code for a node type to one single method.

Most node types of the *Conveyor.Math* plugin are implemented as concrete sub classes of *Bi-*

naryOperation. Since *BinaryOperation* is in turn derived from *AsynchronousNode*, they directly benefit from the optimization described in section 4.5.5.

4.6.2 ITypeProvider and virtual nodes

Certain plugins like *Conveyor.EMBL* contain a number of node types that perform the same operation but differ in small aspects like the actual data type created or returned. For example, there are node types to return all *CDS* features of a sequence, or all *rRNA* features, etc.

Writing these node type classes is tedious and tends to introduce subtle errors. Since they only differ in the exact feature type used, the idea is to create the functionality once and pass information about all use cases to Conveyor.

The *ITypeProvider* interface allows plugins to implement the schema described above. During scanning a plugin, the *Conveyor.Definition* component detects all classes implementing this interface and uses them to retrieve information about additional node types not necessarily implemented as public classes[5].

In the *Conveyor.EMBL* plugins, the *FeatureScanner* class is an implementation of this interface. It scans the plugin for all feature types defined and creates virtual node types for getting all instances of a certain feature type from a sequence or creating a new instance. The functionality itself is implemented as two simple nested inner classes that are not publically visible in the plugin's API.

As a result, new node types for handling a certain feature type are automatically created for each implemented feature type. The amount of necessary code duplication and code maintenance is significantly reduced.

4.6.3 Automatic nodes

Another often used schema is calling a parameterless method defined in an interface (e.g. getting the length of a sequence) or accessing a property of an interface. To reduce the amount of necessary code, a *ITypeProvider* implementation is added to Conveyor that scans interfaces for matching methods and properties. If these methods/properties are marked with a certain attribute and have a suitable return type, a virtual node type is created. The implementation of this node type is based on *ImmediateProcessingNode* and simply invokes the method or the property's getter method. Thus providing a node type to wrap a method or a property defined in an interface is just a matter of adding an attribute (*AutomaticNode*) to the method's definition. An optional description attribute may be used to add a description to the virtual node, too.

[5]The actual plugin scanning for node classes is done by the default implementation of *ITypeProvider*.

4.7 Bioinformatics analysis plugins

Conveyor as presented so far is a generic workflow engine. Its open and extensible design allows using Conveyor with almost any kind of data.

Nonetheless, most plugins implemented for Conveyor so far are focussing on *bioinformatics* with an emphasis on sequence analysis. The following subsections briefly introduce the various plugins. A complete overview of all plugins is available in the appendix.

4.7.1 Conveyor.BioinformaticTypes

This plugin provides the base data types for all other bioinformatic plugins. It defines a number of interfaces and memory based implementations regarding DNA and protein sequences. Figure 4.10 gives an overview of the bioinformatic data types defined in this plugin. For the sake of completeness it also contain extensions and derived types from other plugins introduced in the following sections.

Data types are usually split into interface (e.g. *DNASequence*) and a simple memory based implementation (e.g. *SimpleDNASequence*). The interfaces are intended to be used for type constraints in node types processing corresponding data types.

Most bioinformatics node types are implemented as generic classes. The type parameters define the type of sequence to be used, usually restricted to *DNASequence* or *AASequence*. This allows result types to use the original sequence's type.

Among the data types, this plugin also contains a number of node types operating on DNA sequences, e.g. for creating the reverse complement of a sequence or for splicing out a part of a sequence.

4.7.2 Conveyor.QualitySequence

A *quality sequence* is a DNA sequence with associated quality values, e.g. a read from a sequencer or the result of an assembly software. This plugin provides the base interface for quality sequences (*IQualitySequence*) and an in-memory representation. It also contains a number of node types operating on quality sequences similar to the operations available for DNA sequences (reverse complement, splicing, etc.).

4.7.3 Conveyor.Annotation

Sequences are often associated with additional information that describe it. These *annotations* may vary depending on the kind of sequence, its source, processings done with the sequence and so on. A well-known source of annotation information are feature tables available in GenBank and EMBL formatted sequence entries.

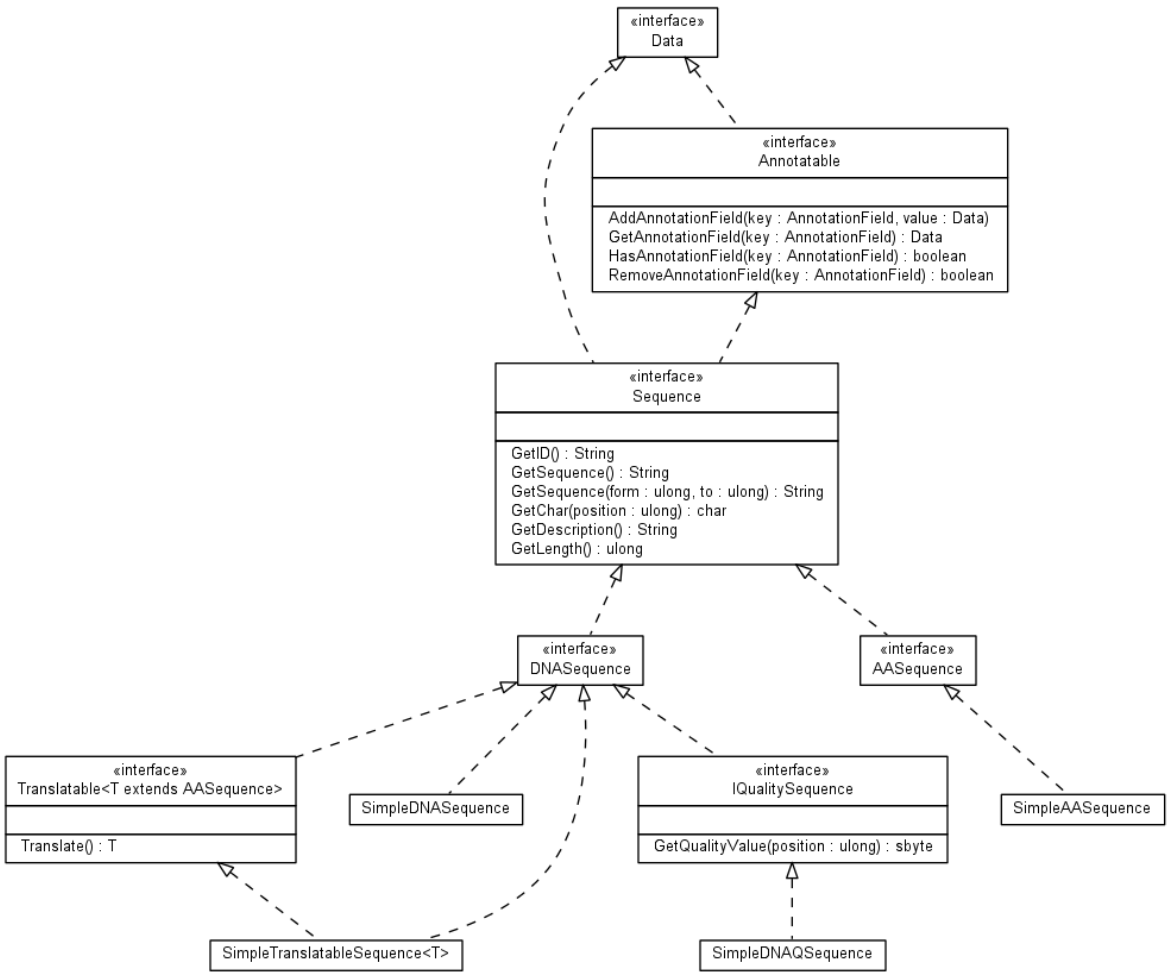

«interface»
Data
«interface»
Annotatable
AddAnnotationField(key : AnnotationField, value : Data)
GetAnnotationField(key : AnnotationField) : Data
HasAnnotationField(key : AnnotationField) : boolean
RemoveAnnotationField(key : AnnotationField) : boolean
«interface»
Sequence
GetID() : String
GetSequence() : String
GetSequence(form : ulong, to : ulong) : String
GetChar(position : ulong) : char
GetDescription() : String
GetLength() : ulong
«interface»
DNASequence
«interface»
AASequence
«interface»
Translatable<T extends AASequence>
Translate() : T
SimpleDNASequence
«interface»
IQualitySequence
GetQualityValue(position : ulong) : sbyte
SimpleAASequence
SimpleTranslatableSequence<T>
SimpleDNAQSequence

Figure 4.10: Overview of types defined in the bioinformatics plugins.

Instead of taking every possible annotation in account (including present and future ones), the design of Conveyor's sequence types is based on the *Conveyor.Annotation* plugin for storing annotations. It defines a hash-like interface based on the concept of an *AnnotationField*. An annotation field defines a possible annotation using a unique name, an optional description, and the type of data that is stored for that annotation[6]. A reflection based mechanism collects all annotation fields defined in the loaded plugins and makes them available to the node type working with annotations.

Which annotation fields are available thus only depends on the loaded plugins itself. This allows any plugin to define new annotation fields, resulting in an extensible mechanism for the annotation of sequences. A number of annotation fields is already defined in the *Conveyor.BioinformaticTypes* plugin.

4.7.4 Conveyor.BioIO

The *BioIO* plugins define node types for reading and writing bioinformatic file format. The *Conveyor.BioIO* plugins supports FASTA, FASTQ and Qual files, while *Conveyor.BioIO.Solexa* and *Conveoyr.BioIO.SFF* handle the formats generated by the Illumina and 454 sequencers.

4.7.5 Conveyor.EMBL

Complex sequence entries with subsequences of different types are made available by the *Conveyor.EMBL* plugins. Among parsing entries of GenBank/EMBL formatted files, it also allows creating corresponding entries and formats them as workflow output. A number of different *feature* types are available as subsequence types according to their specification for the GenBank/EMBL format. Annotation fields for the various feature qualifiers are also included in the plugin[7].

4.7.6 Conveyor.Taxonomy

Accompanied by the generic *Conveyor.Tree* plugin, the *Conveyor.Taxonomy* plugins contains data and node types for accessing the NCBI taxonomy and working with taxonomy entries.

4.7.7 Conveyor.ComputeCluster

Although not directly related to bioinformatics, this plugin is used in many other bioinformatics plugins. It provides a base class for all node types that have to execute an external application. The base class creates an abstraction layer to decouple the actual execution from the processing

[6]Technically, an *AnnotationField* is a public static object initialized at plugin loading time.

[7]Not all feature types contained in the specification are available yet and only a subset of the qualifiers are implemented as annotation fields.

of an application's result. Different implementations may be used for execution, e.g. running applications locally or distributing applications on a DRMAA([TRHD07]) compliant cluster.

4.7.8 Conveyor.Alignment

Alignments in Conveyor are classes implementing the generic *Alignment* interface provided by the *Conveyor.Alignment* plugin. It defines methods for enumeration of involved sequences and getting some basic information about the alignment. A memory-based implementation and node types for working with alignments are also available in the plugin.

The *Conveyor.Alignment.IO* plugin contains nodes for reading alignments from Phylip- and Stockholm-formatted files. Wrapper for a number of external applications to create multiple alignments like Muscle([Edg04]) or ClustalW ([THG94]) are available in the *Conveyor.Alignment.Applications* plugins. Simple pairwise alignments can be created by the node type provided by the *Conveyor.Alignment.Pairwise* plugin.

4.7.9 Conveyor.Blast

Running BLAST ([AMS+97]) is possible with the *Conveyor.Blast* plugin. The results are automatically parsed and converted to corresponding Conveyor data types for further processing.

The plugin also handles BLAST databases, including giving access to locally available preformatted databases and creating databases on the fly from a set of sequences.

4.7.10 Conveyor.HMMER3

Similar to the BLAST plugin described above, the *Conveyor.HMMER3* handles HMM based databases and a number of external applications from the HMMER3[8].

4.7.11 Conveyor.GenePrediction

The *Conveyor.GenePrediction* plugin wraps a number of applications for predicting genes on prokaryotic genomes, including Glimmer 3 ([DHK+99]), Critica ([BO99]), and Prodigal ([HCL+10]). Prediction results may later be used to create subsequences or features in GenBank/EMBL sequences.

4.7.12 Conveyor.Phylogeny

Alignment-based phylogeny applications are handled by the *Conveyor.Phylogeny* plugin. It defines the necessary data types for storing phylogenetic trees and distance matrices and contains

[8]http://hmmer.janelia.org/, no publication available yet

wrapper node types for most applications from the Phylip package ([F$^+$89])[9].

4.8 Conveyor clients

A number of different clients has been developed in conjunction with the Conveyor engine and they allow users to design workflows, execute workflows, and finally bundle workflows into executables of their own.

4.8.1 Web-Service + Designer

With the ability to create custom workflows, the Conveyor engine offers the opportunity to utilize available computing resources and provide users a simple way to create and execute their workflows.

The first clients developed for Conveyor clearly address this agenda. The *Conveyor web service* operates on top of the processing component, provides information about plugins and available node types, and controls the execution of workflows. The *Designer* is a Java Web Start application bundled with the web service and allows the creation of workflows with a graphical user interface.

The bundle enables sites to provide access to local compute resources using a *data processing as a service* approach. Users can download the designer, create a workflow, and use the offered compute resources to execute the workflow and retrieve the results after execution.

Figure 4.11 shows the workflow designer during the creation of a workflow. The GUI is splitted into two parts, the main workflow canvas and the node type list on the left. Node types can be added to a workflow by dragging them from the type list and drop them on an arbitary location on the canvas. Connections between node instances of a workflow are also created by dragging the mouse from the source to the target node. Tooltips provide additional information like the current setup of a node instance or whether some elements of a node instance are not valid.

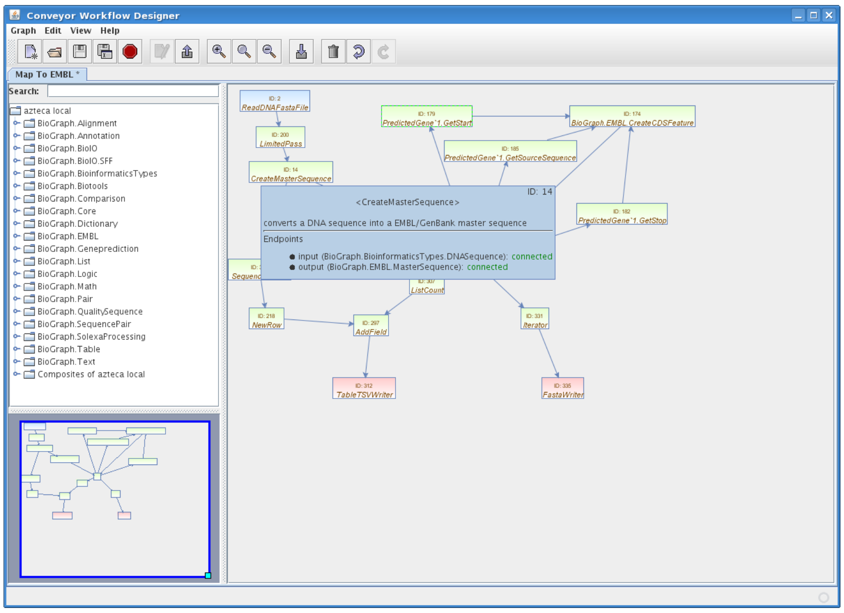
After the design of a workflow is finished, it can be sent to the service for execution. The Designer GUI also offers monitoring of workflows during execution as shown in figure 4.12. It presents an overview of the workflow, and uses different colours to indicate which part of the workflow has already finished. Detailed information about node instances and connection is again available as tooltip. The monitoring view also allows to save results after the processing of a workflow has finished.

The designer does not require an active connection to the service for designing workflows. Upon connection to a service, the current set of node and data types provides by the service is retrieved and stored locally.

The following sections describe the format used by the web server and the Designer to exchange

[9]Instead of using the original applications, the plugin is based on the EMBOSS ([RLB00]) package containing the Phylip tools.

information about plugins and node and data types, how the Designer works with the type system used by *Conveyor*, and the way the web server executes workflows.



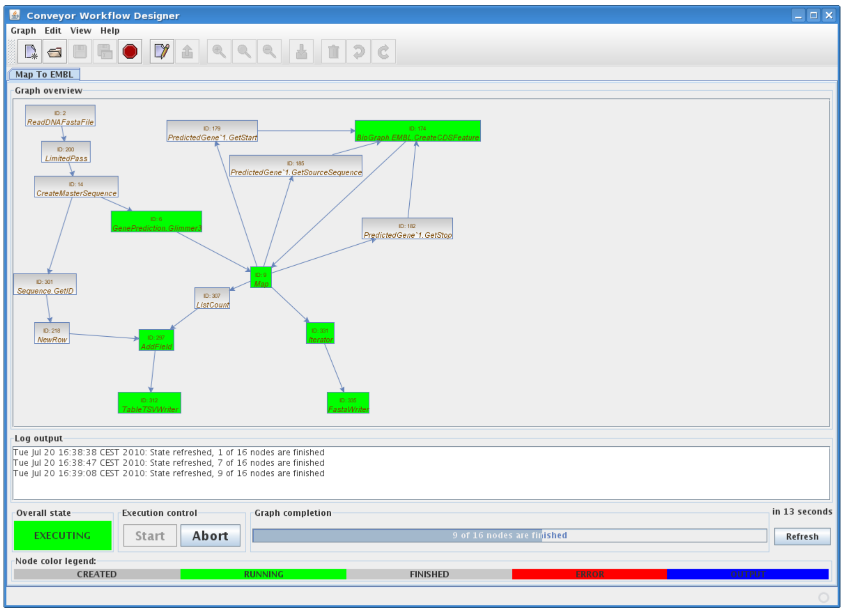
ID: 14
<CreateMasterSequence>
converts a DNA sequence into a EMBL/GenBank master sequence
Endpoints
- input (BioGraph.BioinformaticsTypes.DNASequence): connected
- output (BioGraph.EMBL.MasterSequence): connected

Figure 4.11: The screenshot shows the *Conveyor* designer GUI during the creation of a workflow. On the left a list of available node types grouped by plugin and an overview of the complete workflow is available. The main part on the right consists of the canvas used to draw the workflow. Dragging a node type's label from the list to the work canvas creates a new node instance. Furthermore dragging between two node instances creates a connection (if possible). The screenshot also shows a tooltip, which is available for a node instance. It includes information about the type, its connections, and their validation state.

Plugin information exchange format

One of the methods exposed by the *Conveyor* web service returns information about the available plugins and their data and node types. The Designer uses this information to build a list of available node types and to perform type checks (see the following section for information about the Designer type system implementation).

The plugin information format is based on XML. Foreach plugin available on the service, it contains a <*plugin*> element with a list of data types and a list of node types defined in the plugin. Types are identified by their full qualified name. Figure 4.13 shows an example of a data type definition in the plugin information format. The enclosing <*datatype*> element defines the unique name of the data type and contains sub elements for type parameter and super classes. The type parameter lists contains the position of the type parameter in the type parameter list and a sub



```
Tue Jul 20 16:38:38 CEST 2010: State refreshed, 1 of 16 nodes are finished
Tue Jul 20 16:38:47 CEST 2010: State refreshed, 7 of 16 nodes are finished
Tue Jul 20 16:39:08 CEST 2010: State refreshed, 9 of 16 nodes are finished
```

Figure 4.12: This screenshot shows the *Conveyor* designer GUI during monitoring the execution of a workflow. The main area of the window contains an overview of the workflow using different colors to indicate the state of the various node instances. Finished nodes are shown as gray boxes and green boxes indicate nodes that are currently able to process data.

element that defines the type constraints of the parameter. In this example, the type parameter's value is constrained to types implemented the *Conveyor.BioinformaticsTypes.Sequence* interface, the base interface for all sequence related types in *Conveyor*.

The super classes list contains elements for the base class and all implemented interfaces of a data type. If either the base class or an interface is a generic type, its type parameters are also defined as sub elements. In the example, the data type implements the generic *Conveyor.Alignment.Alignment'1* interface using its first type parameter. Super class definitions may be arbitary complex and may contain nested generic types or self references.

```
<datatype name="Conveyor.Alignment.SimpleAlignment`1">
  <typeParameters>
    <type position="0" name="T">
      <restrictedTo name="Conveyor.BioinformaticsTypes.Sequence"/>
    </type>
  </typeParameters>
  <superclasses>
    <type name="Conveyor.Core.Data"/>
    <type name="Conveyor.Alignment.Alignment`1">
      <typeParameters>
        <type position="0" name="T">
          <restrictedTo name="Conveyor.BioinformaticsTypes.Sequence"/>
        </type>
      </typeParameters>
    </type>
  </superclasses>
</datatype>
```

Figure 4.13: Definition of a data type in the plugin information format. The definition contains the name of the data type (*Conveyor.Alignment.SimpleAlignment'1*), the type parameters and all super classes and interfaces the data type implements. In this example the type is a simple generic type for storing alignments made of sequences. The type parameter restricts the possible values to those classes that implement the *Conveyor.BioinformaticsTypes.Sequence* interface. The same parameter is also referred to in the super class definition for *Conveyor.Alignment.Alignment'1*, which is the interface for all classes implementing alignments.

An example for a node type definition is shown in figure 4.14. The enclosing *<nodetype>* elements contains attributes for the unique class name of the node type, the optional user friendly display name and a description of the node type's function[10]. The *section* attribute defines whether the node type is an input, output, or processing node.

Type parameter, configuration items, links, and tags[11] are defined by sub elements. Type parameters of generic node type may contain restrictions similar to data types. In the example a boolean configuration items is defined for the node type, giving its name, a description, a de-

[10]Display name and description are usually taken from non mandatory attributes a developer added to a node type's class

[11]Optional classification of node types, by short, hierarchic description. Not used in *Conveyor* yet.

fault value, and a flag to indicate whether this item is optional. Two links are given, one input and one output link. The output link has a fixed type (*Conveyor.Math.Integer*) without any reference to one of the type parameters. The input link in turn uses the type parameter of the node class to restrict its possible data types. It is defined to be an implementation of the *Conveyor.Alignment.Alignment'1* interface, refering to the node type's first parameter for its type parameter. This allows the node type to be used with any alignment type (interface) that is based on DNA sequences (restriction of first parameter). More details on the type system and its implementation is presented in the next subsection.

Designer type system implementation

Since the Designer does not have access to the node types itself, it has to rebuild the type system used by *Conveyor* and implement the necessary logic for working with types, e.g. for checking whether two endpoint types are compatible.

The plugin information retrieved from a *Conveyor* web service is transferred to an internal representation of its content. Figure 4.15 shows an excerpt of the type system's API. It can roughly be splitted into two parts: the definition of data and node types (*IPlugin*, *INodeType*, *IDataType*), and node instances being part of a workflow (*INode*, *IConnector*, *ITypeParameter*, *IDataType*).

IDataType is the most important interface in the Designer type system. On the one hand it is used to represent the data types defined in the plugin information, the type parameters of node types (described by the *INodeType* interface), and their link types (as data type of *IConnectorType* instances). On the other hand concrete data types in a workflow are also instances of *IDataType*, e.g. type parameters of node types, or composed types like generic data types with bound type parameters. Instances of *INode* act as container for all information associated with a node using instances of *IConnector* and *ITypeParameter* to store endpoint setup and the current type parameter set. Both latter interfaces define entities that may change their type during workflow design. An endpoint's type may depend on other connected endpoints and a type parameter in turn depends on the endpoints that refer to this parameter in their definition. This aspect is handled by the *IVariableType* interface that allows the Designer to retrieve the orignal type definition (*base type*) and the current one.

Among storing information about available types and designed workflows, the type system is primarily used to validate a workflow during design. The Designer is able to check whether a connection between two endpoints is valid by invoking the *IsAssignableFrom* method of the target endpoint's current type with the source endpoint's current type. It refuses a user's attempt to create a link if the method returns a negative result. It is also used to determine which endpoint may be a target of a new connection if several unconnected input endpoints exists in a target node. Workflow validation also relies on this method.

The implementation of *IsAssignableFrom* depends on the actual *IDataType* implementation. The following ones are used in the *Conveyor* Designer for different purposes:

Concrete data type In the simpliest case the type parameter is a simple, concrete, non generic type, e.g. the *Conveyor.Math.Integer* type shown in the node type example in the previous

```
<nodetype classname="Conveyor.Alignment.DifferentSites`1" displayname="DifferentDNASites"
  description="calculates the number of different sites (S) in the alignment"
  section="processing">
  <typeParameters>
    <type position="0" name="T">
      <restrictedTo name="Conveyor.BioinformaticsTypes.DNASequence"/>
    </type>
  </typeParameters>
  <configurations>
    <config_item name="ignoreUnknown" type="ConfigBoolean" description="ignores columns
      with unknown bases" default="True" optional="0"/>
  </configurations>
  <links>
    <link name="alignment" description="alignment to process" is_input="1">
      <type name="Conveyor.Alignment.Alignment`1">
        <typeParameters>
          <type position="0" name="T">
            <restrictedTo name="Conveyor.BioinformaticsTypes.Sequence"/>
          </type>
        </typeParameters>
      </type>
    </link>
    <link name="sites" description="the number of different sites" is_input="0">
      <type name="Conveyor.Math.Integer"/>
    </link>
  </links>
  <tags/>
</nodetype>
```

Figure 4.14: Definition of a node type in the plugin information format. The type in question examines a DNA sequence alignment and returns the number of columns that contain different values. The type parameter is restricted to types implementing the *Conveyor.BioinformaticsTypes.DNASequence* interface. The node type has two endpoints, one input for the alignment, and one output for the number of sites. The aligment is defined by a nested generic type definition: the enclosing <*type*> element sets the type to the *Conveyor.Alignment.Alignment'1* interface, and the nested <*type*> elements restricts the type parameter of the alignment class to the type parameter of the node type itself. The output endpoint's type is a fixed type (*Conveyor.Math.Integer*) without any type parameter.

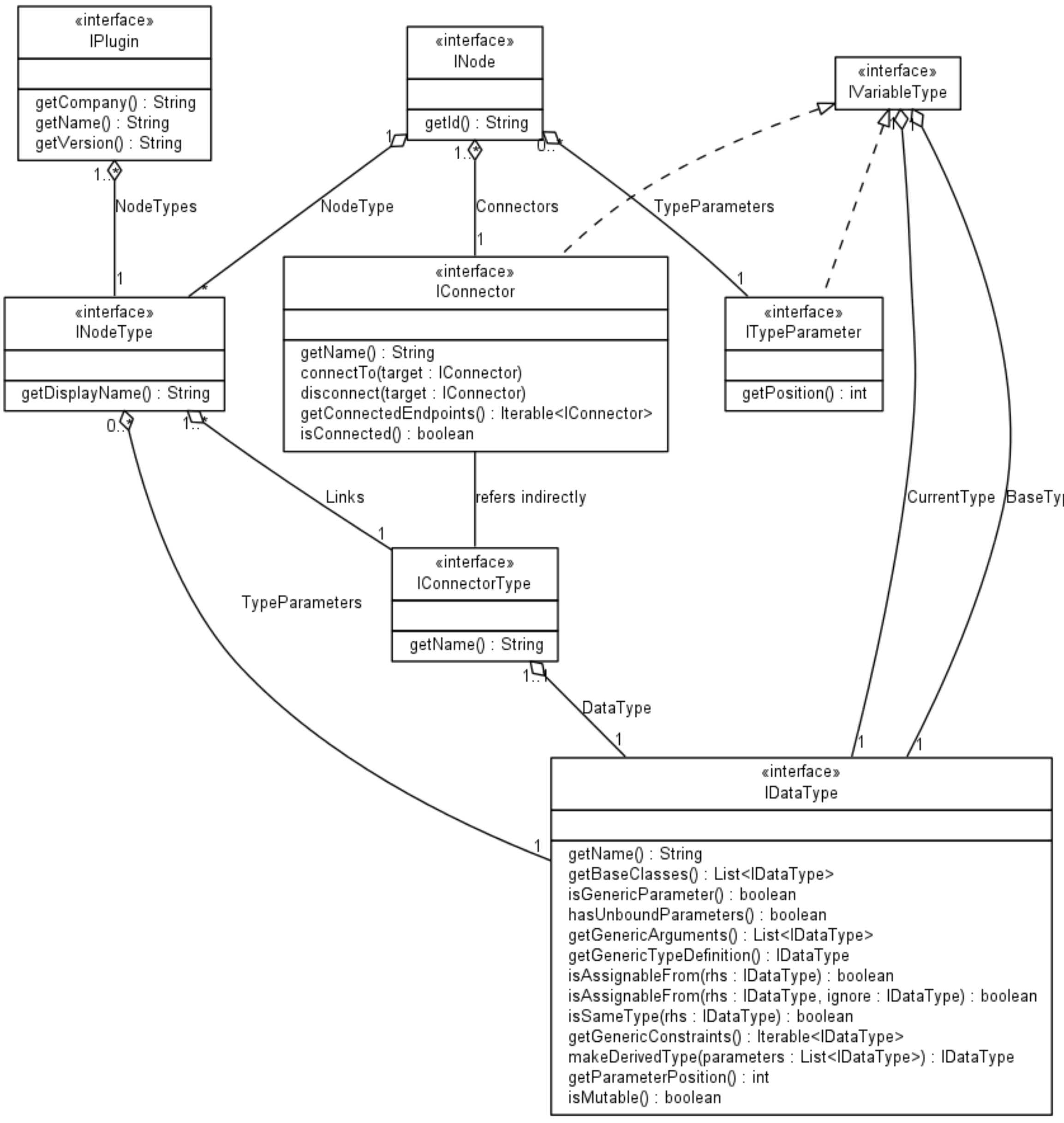

«interface»
IPlugin
getCompany() : String
getName() : String
getVersion() : String
«interface»
INode
getId() : String
«interface»
IVariableType
NodeTypes
NodeType
Connectors
TypeParameters
«interface»
INodeType
getDisplayName() : String
«interface»
IConnector
getName() : String
connectTo(target : IConnector)
disconnect(target : IConnector)
getConnectedEndpoints() : Iterable<IConnector>
isConnected() : boolean
«interface»
ITypeParameter
getPosition() : int
Links
refers indirectly
CurrentType
BaseType
TypeParameters
«interface»
IConnectorType
getName() : String
DataType
«interface»
IDataType
getName() : String
getBaseClasses() : List<IDataType>
isGenericParameter() : boolean
hasUnboundParameters() : boolean
getGenericArguments() : List<IDataType>
getGenericTypeDefinition() : IDataType
isAssignableFrom(rhs : IDataType) : boolean
isAssignableFrom(rhs : IDataType, ignore : IDataType) : boolean
isSameType(rhs : IDataType) : boolean
getGenericConstraints() : Iterable<IDataType>
makeDerivedType(parameters : List<IDataType>) : IDataType
getParameterPosition() : int
isMutable() : boolean

Figure 4.15: Excerpt of the type system implementation used by the *Conveyor* Designer. *IPlugin*, *INodeType*, *IConnectorType* and *IDataType* represent the data and node type information sent by a *Conveyor* web service. *INode*, *IConnector*, *ITypeParameter* and *IVariableType* are the APIs of classes implementing an actual node instance in a workflow during design. Certain details like configuration items, description, validation, etc. are not shown in this figure for the sake of clarity.

sub section. For concrete types, *IsAssignableFrom* returns true if the method's argument is the same type or it is one of the instance's base classes. In case of a type parameter as method argument, a true value is returned if the type parameter's constraints are met by the concrete type.

Type parameter According to the specifiction of type parameters, a given type may be used as type parameter if it mets all type constraints associated with the type parameter. The implementation of *IsAssignableFrom* thus checks whether the method's argument meets all type constraints by using the constraint's *IsAssignableFrom* method.

Generic data type In this case the method's implementation is more complex due to the nature of generic types. Inheritance has to be taken into account for the generic type itself but not for the type parameters.

If the method's argument is a type parameter, checking whether its constraints are fulfilled is sufficient. Otherwise the implementation has to check whether the generic type and the method's argument refer to the same unbound types (e.g. both are lists), falling back to delegating the check to the generic type's super classes. If both types are instances of the same generic type, their type parameters have to be checked. In case of a type parameter for one of the type, the check is a recursive call to *IsAssignableFrom*. Otherwise the method *IsSameType* is used, which can be implemented in a straight-forward way.

The Designer also has to keep track of type parameters and update them accordingly. Each type parameter is bound to all endpoints that refer to it. Upon connecting or disconnecting an endpoint, the type parameter update is triggered. It collects the current types of all collected endpoints and resolves a single type that is compatible with all endpoint's type. This type becomes the new value of the type parameter.

Changing a type parameter in turn updates the endpoints which delegate the type update to connected endpoints. Creating or deleting a single link between endpoints may thus trigger a flood of type changes in the whole workflow.

During saving a workflow to a file, the Designer stores all type parameter information in the workflow file. They are used either for recreating the type setup upon the next loading of the workflow, or for creating the correct instances of nodes in *Conveyor* if the workflow is executed.

Web service workflow execution

The *Conveyor* web service's main purpose is executing workflow, and thus offers a number of methods for handling execution:

CreateJob Creates a new job using a given workflow. A job is identified using a GUID[12] returned by this method.

CreateUploadFile Creates a new server-side file that can be used to transfer input data to the *Conveyor* server. All input files referred to in workflow nodes have to be transferred to

[12] globally unique identifier

the server. Access to local files on the host running the Designer or files stored on the *Conveyor* server itself are not accessible in the *Conveyor* web service.

UploadFileChunk Transfers a part of a input file to the server. Since the size of a web service request is limited, large file have to be transfered in several chunks.

ValidateGraph Instructs the job to validate the workflow and returns the validation result.

StartGraph Starts a validated job.

AbortGraph Aborts a running job.

GetJobState Returns the overall job state.

GetOutputType Returns the type of data an output node has created.

GetOutput Returns the data of an output node.

DeleteJob Removes a finished or aborted job.

A *job* acts as a wrapper for an *Conveyor.Processing.IProcessor* instance that was setup to exeucte the workflow given during job creation. The web service methods thus mostly resemble the IProcessor interface.

Jobs and the web service itself are separated as shown in figure 4.16. An intermediate *JobManager* is used for managing job instances and reporting the existing instances to the web service. It can act independently from the web service and can rerun on a different host. Depending on the job manager, a job may be executed in different ways, e.g. inside the job manager itself, as separated process, or using distributed computing like a grid or cloud based service. In a production setup, it is highly recommended to separate at least the web service and the job manager to compensate for disruption and outages, e.g. by running them on different hosts. Communication between all four entities shown in the figure is done by network-aware protocols available in .NET.

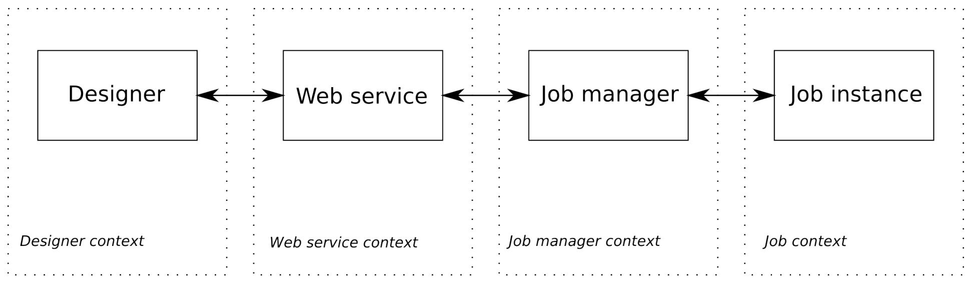


Figure 4.16: Different contexts (e.g. different hosts) can be used for the separation of jobs and web service during execution.

4.8.2 GraphRun commandline utility

While the service and designer applications presented in the previous subsection allow the creation of workflows, their execution is bound to the web service. This shortage is addressed by the *GraphRun* utility. It is part of the processing component and allows the execution of a workflow on the command line. Configuration parameters like input files or threshold values can be set as command line option, and output of nodes can be redirected straight into files. *GraphRun* also offers to monitor the execution of a workflow and provides statistical information about nodes and endpoints. It is primarly intented to be used for batch processing of workflows, using a locally available Conveyor installation.

4.9 Rapid Design and Deployment: Conveyor2Go

While the clients presented in the previous subsections focus on batch execution and offering workflows as a service, *Conveyor2Go* has a different goal. It implements the last step of the *Rapid design and deployment* process show in figure 4.9: a workflow is designed, tested, and finally deployed by the users.

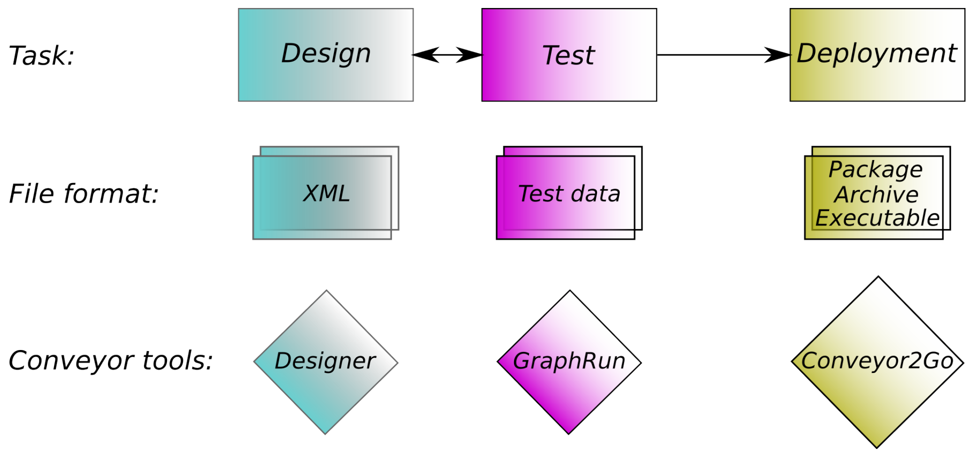
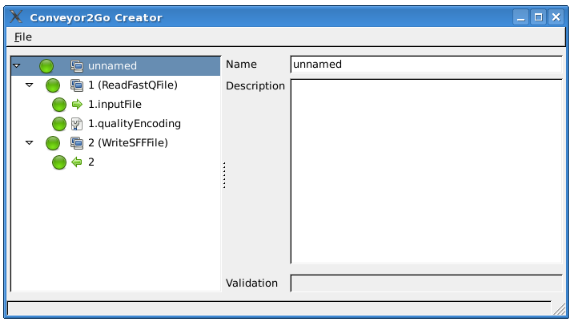


Figure 4.17: Software development process from design to deployment. The top row contains the steps involved in software development, the second row the kind of data processing the corresponding step in case of a workflow, and the last row finally shows the *Conveyor* tool involved in this step.

Conveyor2Go is able to convert a workflow definition to a standalone executable. It analyses the workflow and bundles Conveyor and all plugins used in the workflow in a standalone binary. The binary in turn can be installed on the local computer or sent to other users for installation.

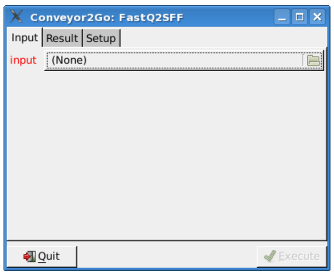
During the bundling process, the workflow developer can manipulate the configuration items of nodes in the workflow, assign fixed values to them, or rename them to better human readable names. The resulting binary provides both a commandline interface (and can thus be used for batch processing), and a Gtk+ based graphical user interface. Figure 4.18 shows a screenshot of the *Conveyor2Go Creator* for creating self-contained executables. Figure 4.19 present a screenshots of the GUI[13] generated by Conveyor2Go after changing the internal names of input files, configuration items and output files to more user friendly values. An overview of the corresponding command line usage is given in figure 4.20.



unnamed
1 (ReadFastQFile)
1.inputFile
1.qualityEncoding
2 (WriteSFFFile)
2

Name: unnamed
Description:
Validation:

Figure 4.18: The interface of the *Conveyor2Go Creator* user interface. The tree on the left contains all editable items, including the workflow description itself, input files, configuration items, and output file. The workflow in the example contains two nodes with one input file, one configuration setting, and one output node. Names and descriptions can be changed to be more user friendly and fixed values may be set for configuration items



input: (None)

Figure 4.19: Screenshot of the GUI generated for the *Conveyor2Go* application. It allows setting the input files, redirect output to file, and setup the configuration items for a workflow execution.

Currently Conveyor2Go creates self-contained .NET executables. For this reason the intended user requires a .NET environment. In case of UNIX, the *mkbundle* tool can be used to bundle the complete Mono runtime with the generated workflow executable, creating a single standalone, statically linked binary without a dependency to a .NET or Mono runtime. Additional

[13] preliminary version

```
FastQ2SFF   Converts FastQ files to SFF files
Usage: FastQ2SFF [options] input

File inputs:
  input        [primary] FastQ file to convert

Results:
  output       [primary] name of SFF file to generate

Configurable options:
  qualityEncodi[ENUMERATION] Encoding of quality values in FastQ quality line

Default options:
  help         show this help message
  batch        enforces command line only usage; all necessary options have to
                be given as command line arguments
  gui          starts the application with the graphical user interface
  debug        enables debug output in batch mode

Each option may be given on the command line using either '-', '/' or '--',
followed by the option's name, a delimiter (space, '=' or ':'), and the options
 value (if required).
Example: /foo:1 -bar test

The file input input may also be given as additional command line argument. If
'-' is used as file name in batch mode, the standard input is redirected for
this file input.
Instead of specifying an output file for output, not using the option at all
will redirect the result to standard output in batch mode.
Using the debug mode will disable the redirection of standard input and standard
output. Instead pressing enter during execution prints debug information about
the nodes in the workflow to standard error.
```

Figure 4.20: The usage message of Conveyor2Go executables. All information shown in the creator are stored in the resulting executable, and are used for naming command line options and giving descriptions to the user.

requirements like libraries or external applications have to be present in the target system.

4.10 MGX: Metagenome analysis enviroment

As a result of the widespread availability of next-gen sequencing technologies, the field of *Metagenomics* has been focussed by many researchers. Metagenomics does not address a single organism but allows to get information about a complete community with different organisms. Similar to whole genome shutgun, genomic DNA is sequenced. Instead of cultivating and cloning the same organism, DNA material is directly harvested from a metagenome sample. This approach allows researchers to sequence organisms that cannot be cultivated. It however also introduces a number of challenges, but which are beyond the scope of this thesis.

MGX is a new platform for management and processing of metagenome data sets, that is currently under development as part of a PhD project. It will use Conveyor for data processing engine, amd features predefined and user-specified workflows. A number of plugins is under development that contain metagenomic functionality and which will become part of the default Conveyor distribution.

CHAPTER 5

Use cases

The following sections present a selection of use cases of Conveyor from the bioinformatics domain. They have already been either used in production or have acted as templates for production usage.

Although these use cases put a focus on bioinformatics, Conveyor is not restricted to that scientific domain. Plugins may extend *Conveyor* to handle any type of data and perform automated analyses on them.

5.1 Reference based genome annotation

The workflow presented in this use case study implements a pipeline for annotating a genome using an already annotated reference genome. The new ultrafast sequencing technologies introduced in recent years, result (among other benefits) in a flood of new draft genomes. Furthermore instead of concentrating on a single organism or a single strain, a complete species including several subspecies can be sequenced in a single run. Due to the large amount of data generated in these cases, manually finishing or annotating these genomes is not feasible with respect to financial or man power constraints. The research objectives thus focus on the identification of differences among the draft genomes by mapping them onto a reference genome and by comparing the results for the different strains. In many cases, manual annotation is only done for the genes of interest - if done at all.

The overall workflow is built from three different parts: gene prediction on plain genomic DNA sequences, homology searches for the predicted genes versus an annotated reference genome, and handling of predicted genes without homologues. The result of the workflow is exported as an EMBL-formatted file containing the predicted genes and their functional annotations. The workflow in this use case includes a BLAST search to map genes to the reference genome. In addition, it uses another BLAST search against a general purpose database for genes without mapped reference genes. The workflow as shown in figure 5.1 consists of several parts:

a) The preparation of the reference genome. A BLAST database is built from all CDS features of the given EMBL or GenBank file.

b) Gene prediction for the novel genomes. Putative coding regions are predicted using the Glimmer3 ([DHK$^+$99]) prokaryotic gene prediction software with default parameters, and CDS regions are created.

c) Search for homologues in the reference genome. Using BLAST, all CDSs of the novel genome are compared to the CDSs of the reference genome. The results are filtered based on an e-value threshold and a minimal coverage cut-off, and put into a list.

d) Copy of annotation fields from matching CDSs; if a homologous CDS was found during the BLAST search, its annotation fields like gene product, gene name and function are transfered to the novel CDS.

e) Search for homologues in a public database: For novel CDSs without known similar genes in the reference genome, a public database like SwissProt as part of the UniProt database ([Con10]) or the GenBank database ([BKML$^+$10]) can be used with BLAST to search for homologues.

f) Copy of annotation fields from matching database hits: Similar to d) the content of annotation fields of the best hits against a public database are transferred to the novel CDSs.

g) Mark still unassigned CDS. All CDSs without assigned annotation during steps d) and f) are marked as being putative.

h) Export of genome: The complete genome including the predicted CDS is exported as an EMBL or GenBank file.

The main benefits of implementing the above workflow in the Conveyor system are flexibility and extensibility. The use case only shows an exemplary design of the workflow. Changing the reference organism or the gene prediction tool is only a matter of changing a node in the workflow. Thresholds e.g. for the e-value threshold of BLAST results or settings like the database to use in step e) are configuration fields directly accessible via the designer GUI.

The complete workflow is implemented without writing a single line of code or invoking an application except Conveyor and the designer GUI. Moreover, the command line execution utility that comes with Conveyor allows executing the workflow completely without user interaction. It also offers the ability to change the configuration for individual runs. The BLAST steps used in this use case are good examples for generic types. The BLAST result data type is a generic type using a type parameter for the BLAST query and another type parameter for the database sequence. Retrieving the query or the subject from a BLAST hit, thus results in the original feature from the EMBL or GenBank file with full access to all feature keys as annotation fields. This also allows for transfering keys usually not exported to BLAST databases like notes or EC numbers in a consistent way.

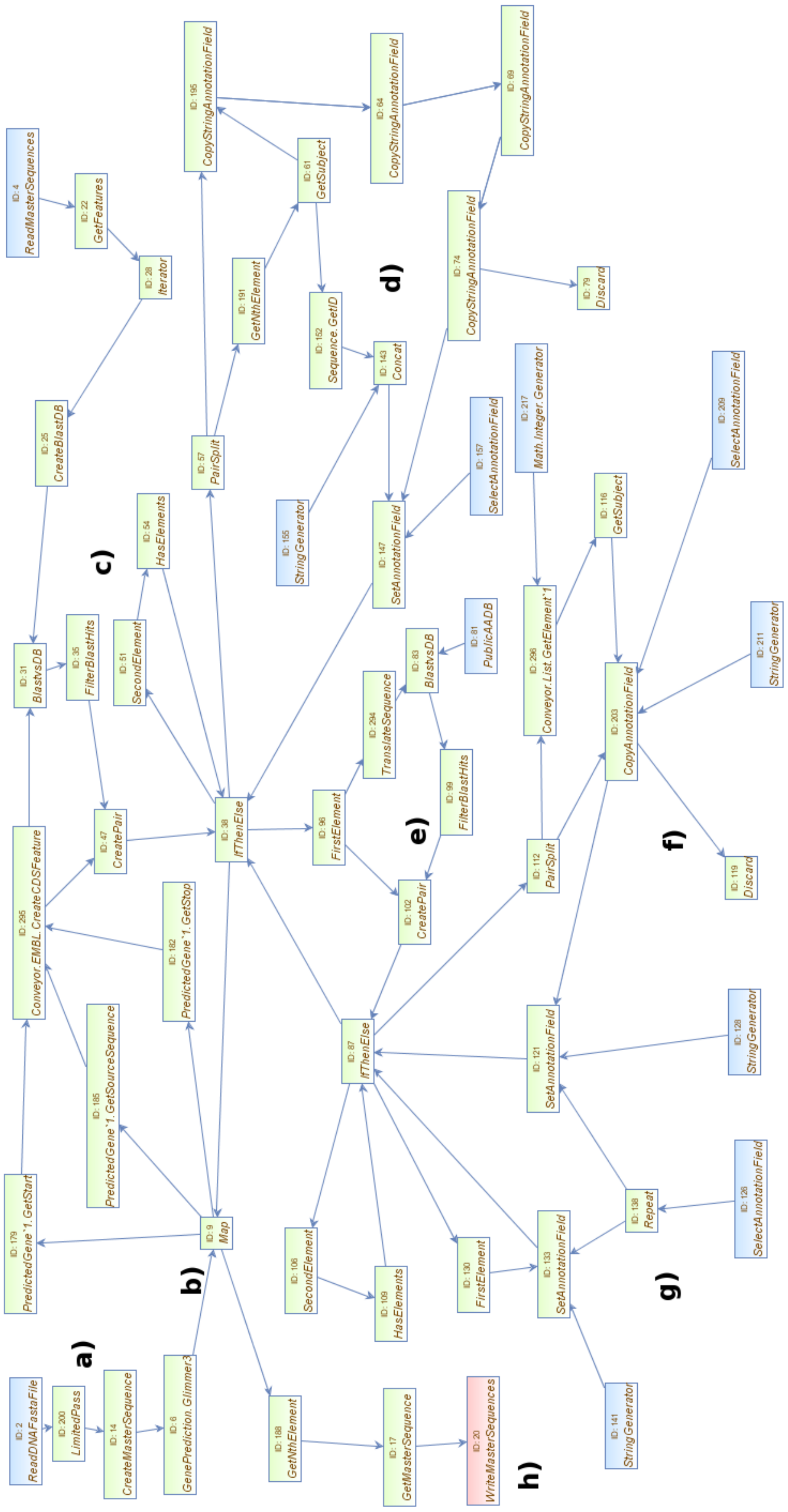


Figure 5.1: Example workflow for annotating a genome using a reference genome. The markers refer to certain functionalities explained in the text.

5.2 Comparative genomics

With several genomes of a selected species available, dividing them into groups according to their phenotype and determining the specific gene content of these groups is the next important step in comparative genomics. Conveyor provides all necessary functions for this kind of analysis. Figure 5.2 shows a workflow to calculate the core genome of a set of given genomes. The core genome consists of all genes present in all organisms, taking paralogs into account. Based on annotated genomes, the set of homologue genes for all genes of one organism are computed. If the list of homologues of a single gene only contains one homologue in each organisms, it is considered a member of the core genome.

The workflow thus consists of several parts:

a) Import of genomes including their annotation from EMBL or GenBank formatted files. Sequences shorter than a given length are filtered out, e.g. plasmids.

b) A list of all genome IDs is created.

c) A blastable database is created from the CDSs of all organisms.

d) The genome with the lowest number of coding sequences is selected.

e) A second BLAST-able database containing all coding sequences of the smallest genome is created.

f) Homologous sequences for all coding sequences of the smallest genome are determined using BLAST.

g) The list of BLAST hits is reduced to significant hits. Their bit scores have to exceed a certain ratio if compared to the bit score of the most significant hit (first hit in BLAST result).

h) The set of sequences referred to by the remaining BLAST hits has to contain a single gene of each organism to be considered a part of the core genome.

i) Each sequence detected as a possible homologue is compared against all coding sequences of the smallest genome using BLAST. This step filters out possible paralogs in the genome and ensures that all genes in all candidate sets are indeed homologues.

j) Alignments are created from the sequences of all candidate core gene sets using Muscle and written as result using the Phylip alignment format.

The workflow is only an example implementation of a core genome calculator. Other restrictions may also apply, e.g. filtering out genes related to mobile elements. The number of genomes processed by the workflow is not restricted. Many parameters like thresholds for BLAST results, the required score ratio in step g), or the required length of the genome sequence used in step a) can easily be adapted to the input sequences. The workflow may also be extended by additional processing steps based on the alignment of the core gene sets. Nodes for alignment processing and phylogenetic analyses are already available for Conveyor.

As a special feature this workflow demonstrates the list processing and branching flow features of Conveyor.

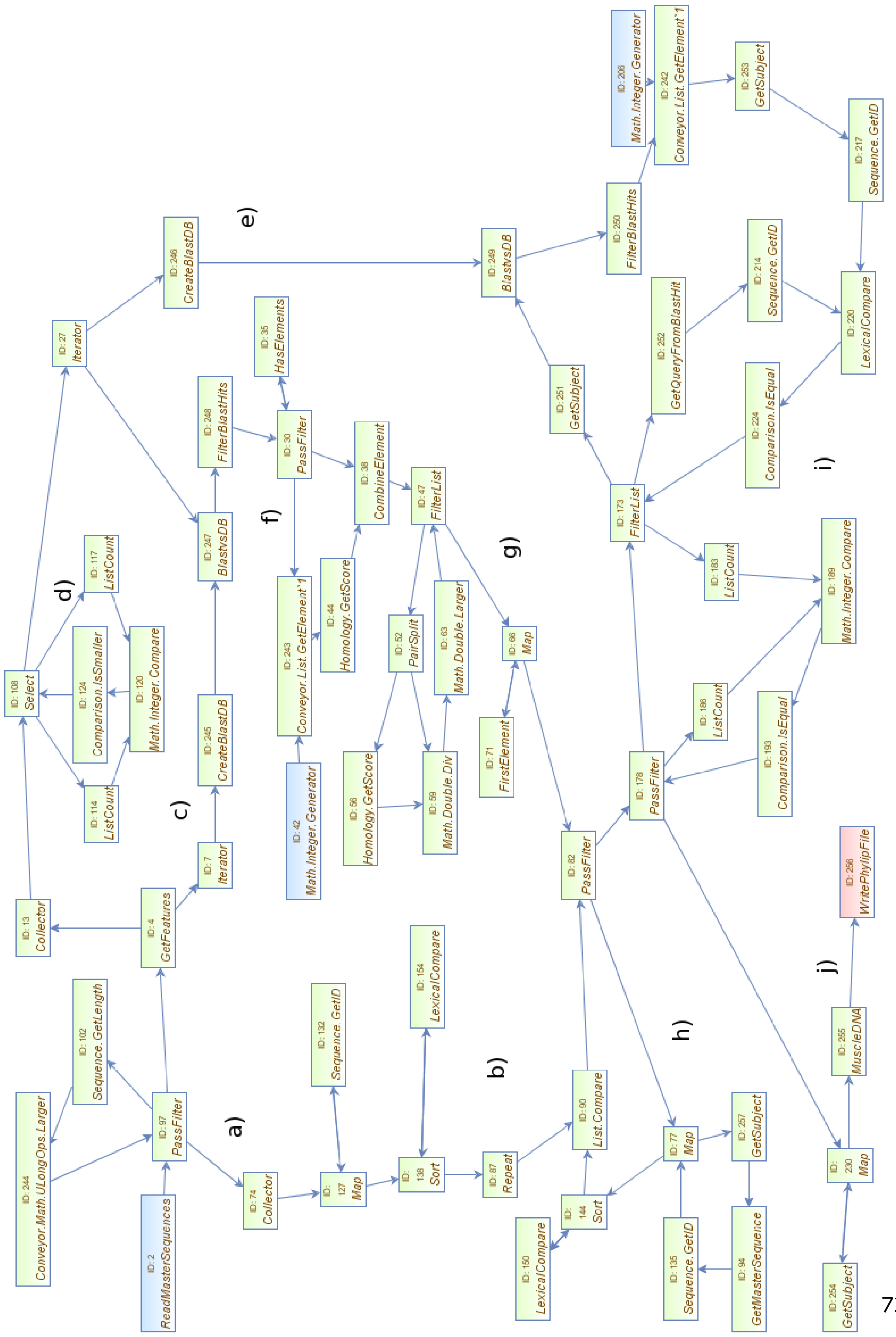


Figure 5.2: Example workflow for determining the core genome of a set of genomes. The markers refer to functionalities described in the text.

5.3 Next-Gen sequencing data processing

This use case is a real world application of Conveyor to sequence data produced by a Illumina Solexa GA IIx sequencer. This machine is able to generate a tremendous amount of sequencing reads but is only capable of producing short read lengths. At the time of writing this thesis, read lengths up to 150 base pairs are possible with 2 x 150 base pairs for paired end reads. Certainly, the assembly of genomic reads into a complete genomic sequence would greatly benefit from larger read lengths. To overcome the limitations of Solexa sequence reads, a non-standard laboratory protocol is used to generate longer sequence reads at the CeBiTec.

In this approach, shorter templates are used, containing a linker between two flanking parts of the template to be sequenced. The length of the template is shorter than the sum of the aimed length of both reads in the paired end read. The template is then sequenced from both sides as shown in figure 5.3. In an ideal case, both reads overlap and include the linker sequence in their 3′ region.

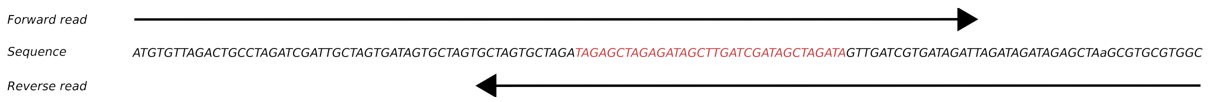

Forward read
Sequence
ATGTGTTAGACTGCCTAGATCGATTGCTAGTGATAGTGCTAGTGCTAGTGCTAGATAGAGCTAGAGATAGCTTGATCGATAGCTAGATAGTTGATCGTGATAGATTAGATAGATAGAGCTAaGCGTGCGTGGC
Reverse read

Figure 5.3: The schema shows the sequence template used for overlapping paired end reads. A known linker sequence (red) is inserted into the original template and both ends of the resulting template are sequenced with overlapping reads.

The resulting sequence pairs can be separated into three categories:

no linker: No linker sequence is found in any of the paired sequences.

single linker: The linker sequence is found in one the paired sequences.

both linker: Both paired sequences contain the linker.

Each category requires a different handling. Reads without linker sequence are considered as single reads and written to the same output file. In case of a single linker, any sequence downstream of the linker is discarded and a paired end read pair is written to the output file. If both sequences contain the linker, a consensus sequence of both reads is created and written to the output.

Figure 5.4 shows the workflow developed for processing the Illumina data. In this example, one lane is processed at a time and sequences are selected using an introduced sequence tag[1] Afterwards, the linker is searched using approximate matching, the number of overlapping bases is calculated, and the read pair is assigned to the corresponding category.

Output is written in *SFF* format to allow an easy processing of the read pairs with a number

[1] In this example, the seuqence run contains different organisms in the same lane using different sequence tags. A Genome Analyser sequencer generates a third read containing the sequence tag only. This read is used in the workflow to filter out non-relevant read pairs.

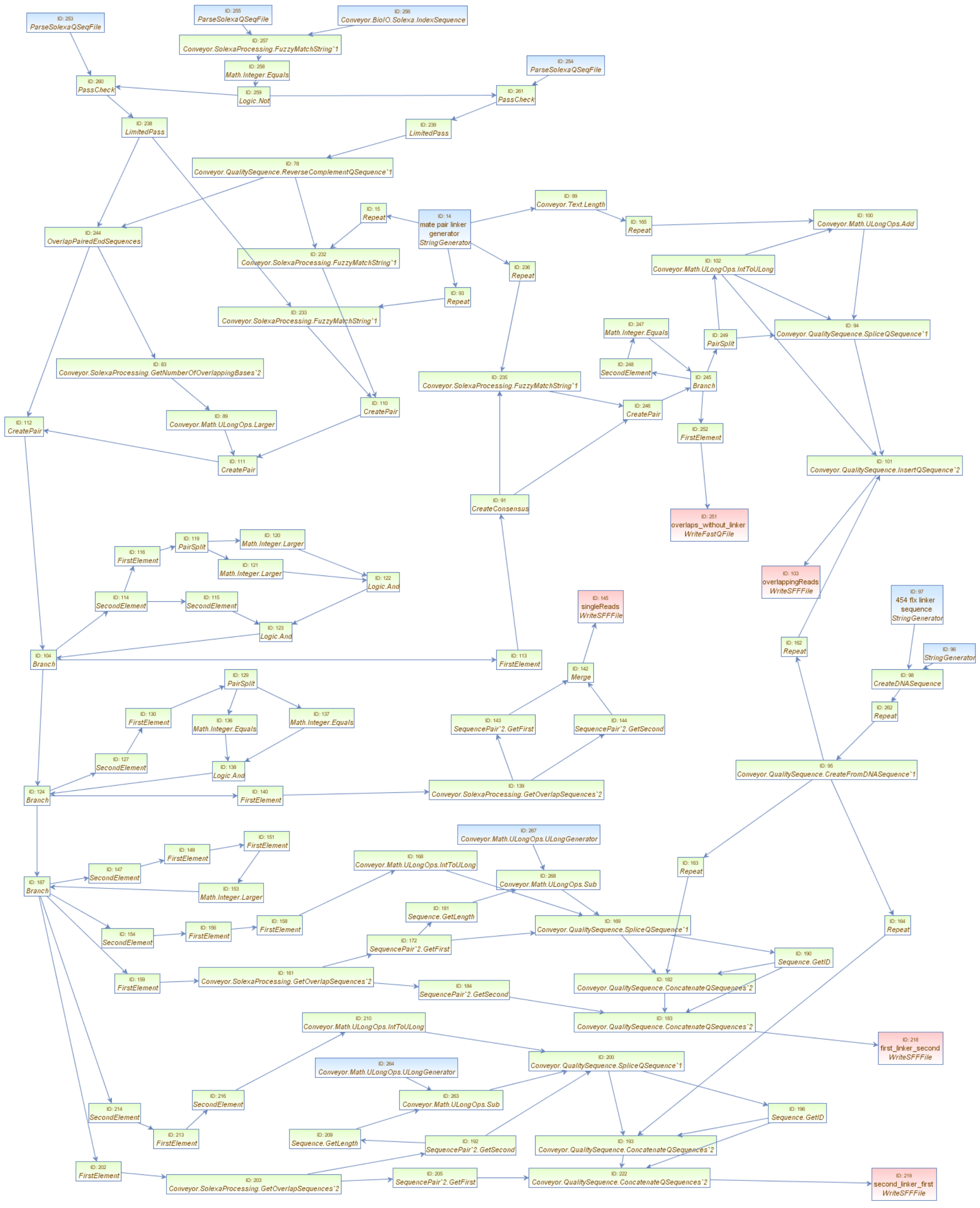

ID: 253 ParseSolexaQSeqFile
ID: 255 ParseSolexaQSeqFile
ID: 256 Conveyor.BioIO.Solexa.IndexSequence
ID: 257 Conveyor.SolexaProcessing.FuzzyMatchString`1
ID: 258 Math.Integer.Equals
ID: 254 ParseSolexaQSeqFile
ID: 260 PassCheck
ID: 259 Logic.Not
ID: 261 PassCheck
ID: 238 LimitedPass
ID: 239 LimitedPass
ID: 78 Conveyor.QualitySequence.ReverseComplementQSequence`1
ID: 99 Conveyor.Text.Length
ID: 15 Repeat
ID: 14 mate pair linker generator StringGenerator
ID: 165 Repeat
ID: 100 Conveyor.Math.ULongOps.Add
ID: 244 OverlapPairedEndSequences
ID: 232 Conveyor.SolexaProcessing.FuzzyMatchString`1
ID: 236 Repeat
ID: 102 Conveyor.Math.ULongOps.IntToULong
ID: 93 Repeat
ID: 233 Conveyor.SolexaProcessing.FuzzyMatchString`1
ID: 247 Math.Integer.Equals
ID: 249 PairSplit
ID: 94 Conveyor.QualitySequence.SpliceQSequence`1
ID: 83 Conveyor.SolexaProcessing.GetNumberOfOverlappingBases`2
ID: 248 SecondElement
ID: 235 Conveyor.SolexaProcessing.FuzzyMatchString`1
ID: 245 Branch
ID: 110 CreatePair
ID: 246 CreatePair
ID: 112 CreatePair
ID: 89 Conveyor.Math.ULongOps.Larger
ID: 252 FirstElement
ID: 111 CreatePair
ID: 101 Conveyor.QualitySequence.InsertQSequence`2
ID: 91 CreateConsensus
ID: 251 overlaps_without_linker WriteFastQFile
ID: 119 PairSplit
ID: 120 Math.Integer.Larger
ID: 116 FirstElement
ID: 121 Math.Integer.Larger
ID: 122 Logic.And
ID: 103 overlappingReads WriteSFFFile
ID: 97 454 flx linker sequence StringGenerator
ID: 114 SecondElement
ID: 115 SecondElement
ID: 145 singleReads WriteSFFFile
ID: 123 Logic.And
ID: 162 Repeat
ID: 96 StringGenerator
ID: 104 Branch
ID: 113 FirstElement
ID: 142 Merge
ID: 98 CreateDNASequence
ID: 129 PairSplit
ID: 130 FirstElement
ID: 136 Math.Integer.Equals
ID: 137 Math.Integer.Equals
ID: 143 SequencePair`2.GetFirst
ID: 144 SequencePair`2.GetSecond
ID: 262 Repeat
ID: 127 SecondElement
ID: 138 Logic.And
ID: 95 Conveyor.QualitySequence.CreateFromDNASequence`1
ID: 124 Branch
ID: 140 FirstElement
ID: 139 Conveyor.SolexaProcessing.GetOverlapSequences`2
ID: 267 Conveyor.Math.ULongOps.ULongGenerator
ID: 151 FirstElement
ID: 149 FirstElement
ID: 168 Conveyor.Math.ULongOps.IntToULong
ID: 163 Repeat
ID: 147 SecondElement
ID: 268 Conveyor.Math.ULongOps.Sub
ID: 187 Branch
ID: 153 Math.Integer.Larger
ID: 181 Sequence.GetLength
ID: 154 SecondElement
ID: 156 FirstElement
ID: 158 FirstElement
ID: 169 Conveyor.QualitySequence.SpliceQSequence`1
ID: 164 Repeat
ID: 172 SequencePair`2.GetFirst
ID: 190 Sequence.GetID
ID: 159 FirstElement
ID: 161 Conveyor.SolexaProcessing.GetOverlapSequences`2
ID: 184 SequencePair`2.GetSecond
ID: 182 Conveyor.QualitySequence.ConcatenateQSequences`2
ID: 183 Conveyor.QualitySequence.ConcatenateQSequences`2
ID: 210 Conveyor.Math.ULongOps.IntToULong
ID: 218 first_linker_second WriteSFFFile
ID: 200 Conveyor.QualitySequence.SpliceQSequence`1
ID: 264 Conveyor.Math.ULongOps.ULongGenerator
ID: 216 SecondElement
ID: 263 Conveyor.Math.ULongOps.Sub
ID: 196 Sequence.GetID
ID: 214 SecondElement
ID: 213 FirstElement
ID: 209 Sequence.GetLength
ID: 192 SequencePair`2.GetSecond
ID: 193 Conveyor.QualitySequence.ConcatenateQSequences`2
ID: 202 FirstElement
ID: 203 Conveyor.SolexaProcessing.GetOverlapSequences`2
ID: 205 SequencePair`2.GetFirst
ID: 222 Conveyor.QualitySequence.ConcatenateQSequences`2
ID: 219 second_linker_first WriteSFFFile

Figure 5.4: Workflow for Solexa data preprocessing. Read pairs are read from Solexa lane files, linker sequence position and read pair overlap are calculated, and the read pairs are processed and written to the output files depending on the presence of linker sequences and overlapping bases.

of established assembly tools. Since SFF is usually generated by Roche's *454* sequencer, the linker sequence in the read pairs is replaced with the sequence used by 454 to indicate read pair extends.

5.4 Results

This section presents the results obtained by the presented use cases. The first subsection introduces the test setups and the following subsections contain the results for the three use cases.

5.4.1 Test setup

During the use case tests, the workflows are executed on a host equipped with 256 GB RAM and 4 Intel Xeon E7540 CPUs running at 2.00 GHz. Each CPU provides 6 cores and hyperthreading per core, resulting in 24 real cores and 48 virtual cores presented to the operating system[2]. Runtime of workflows are measured using the standard UNIX time(1) command which lists the overall processing time (*real*), the time spent in the application itself (*user*), and the time spent in system calls (*sys*). Unless mentioned otherwise, the workflows were executed 5 times, and the average times over those runs are used in the discussion.

Two different setups are used for benchmarking the workflows:

DRMAA cluster All nodes utilizing external tools are submitting their tool invocations to the CeBiTec compute cluster. Table 5.1 lists the available hosts of the compute clsuter at the time of writing this thesis. These 236 hosts contain 472 CPUs with a total of 1170 cores, resulting in about 800 slots in the appropriate cluster queue. The cluster is managed by Oracle Grid Engine (OGE), which provides both a command line and DRMAA([TRHD07]) compliant API for submitting and monitoring jobs. Tools are set up to read their input from standard input and to write results to standard output. Intermediate files are stored on a NFS-based, network-wide available volume.

For accurate discussion of timing benchmark results, several factors of this setup have to be taken into account. First of all, due to the nature of the shared filesystem, files created on a host are not immediately visible to other hosts in the network. Depending on the setup of the NFS caches, it may take some time until a NFS client refreshes its caches. In the current CeBiTec setup the refresh interval is set to 30 seconds. Conveyor (resp. the library taking care for running compute jobs) has to take this into account. A node thus has to sleep for 30 seconds between creating job input files and submitting the actual job to the scheduler, and again has to wait for up to 30 seconds after a job has terminated to be sure that the output file exist[3].

[2]Conveyor is not able to differentiate between real cores and virtual cores. Scheduling Conveyor threads to certain cores is operating system dependent and beyond the scope of Conveyor.

[3]30 second timeouts are used at the CeBiTec cluster, the value itself is adjustable for other setups.

Quantity	Machine type	No. of CPUs	Cores/CPU	CPU Freq.	RAM
50	SUN V20z	2	1	$1.8GHz$	$4GB$
50	SUN V20z	2	1	$1.8GHz$	$8GB$
7	SUN V20z	2	1	$1.8GHz$	$16GB$
22	SUN X2200	2	2	$2.6GHz$	$16GB$
73	SUN X2250	2	4	$2.5GHz$	$32GB$
34	FUJ RX200	2	4	$2.6GHz$	$48GB$

Table 5.1: Configuration of the CeBiTec compute cluster in August 2011. Each rows lists the machine type name, its number of CPU, cores, CPU speed, and RAM setup, and the number of machine of this type available in the cluster
.

Since the cluster is a shared resource between several groups at the CeBiTec, exclusive access to all slots of a queue cannot be guaranteed, and other computations may interfere with a benchmark run. It should also be noted that scheduling jobs itself does not happen directly at submission time but within certain intervals defined by the cluster setup[4].

These waiting times have to be taken into account when discussing overall execution time of a workflow. In addition, file I/O (input and output files of compute jobs) happens on a network volume, resulting in an extra overhead compared to local disk I/O.

Despite of this, the *user* time reported in the cluster-based benchmark runs gives a good approximation of the time actually used for the processing of a workflow itself including Mono startup overhead, since it does not include execution time of external tools or file I/O (which is part of the *sys* time).

Local execution To rule out cluster-based effects and to demonstrate the scalability of *Conveyor* with respect to external tools, a second setup uses only local resources. External tools are started as processes on the local host and input and output are streamed directly to resp. from the process. No intermediate files are used.

Depending on the number and kind of available CPU cores and the resources needed by each instance of an external tool, the number of instances may scale well up to the number of available cores. To rule out swapping effects, the hosts used for these tests provide a large amount of RAM, suitable for almost any application available in Conveyor.

It also offers up to 48 virtual cores but since half of them are hyperthreading cores, the performance may not scale as expected, e.g. due to shared floating point logic.

Benchmark runs are done with a different number of processes running in parallel, ranging from a low number of 2 or 4 parallel instances upto the number of (virtual) cores reported by the operating system.

[4]Scheduling interval was set to 35 seconds, resulting in a lag of up to 35 seconds between submission of jobs and transfer of jobs to compute hosts.

5.4.2 Reference based annotation

The workflow was used to annotate 58 *Escherichia coli* replicons (GenBank ([BKML$^+$10]) having accession numbers NC_011602, NC_011603, NC_008253, NC_011748, NC_008563, NC_009837, NC_009838, NC_012947, NC_012759, NC_012967, NC_004431, NC_010468, NC_009786, NC_009787, NC_009788, NC_009789, NC_009790, NC_009791, NC_009801, NC_011745, NC_009800, NC_011741, NC_011750, NC_010473, NC_000913, AC_000091, NC_002127, NC_002128, NC_002695, NC_002655, NC_007414, NC_011350, NC_011351, NC_011353, NC_013008, NC_013010, NC_011742, NC_011747, NC_011407, NC_011408, NC_011411, NC_011413, NC_011415, NC_011416, NC_011419, NC_010485, NC_010486, NC_010487, NC_010488, NC_010498, NC_011739, NC_011749, NC_011751, NC_007941, NC_007946, NC_011740, NC_011743). The official GenBank annotation of *E.coli K-12 DH10B* (NC_010473) was used as the reference genome. For the second BLAST run, the SwissProt subset of the NCBI non-redundant protein database (*nr*) was used, dated of Jun. 23, 2011.

Table 5.2 contains the benchmark results of a varying number of local cores and the values for running the benchmark using the CeBiTec compute cluster. Each row contains the number of processes spawned in parallel and the result of the UNIX `time` command. Figure 5.5 shows the results as a bar chart, and figure 5.6 presents the results as relative values using the case of 4 local cores as reference.

No. of cores	Real time [s.]	User time [s.]	System time [s.]
4	23672.3	88514.8	4855.3
8	11493.9	85415.7	4775.0
16	6878.1	100060.0	6141.8
32	3847.1	108067.0	6587.0
48	3239.9	130318.0	7711.3
Cluster	2379.4	365.5	116.6

Table 5.2: Results of benchmarking the reference annotation use case with a different number of local cores and using the CeBiTec compute cluster. Each row contains the number of processes, the mean values of the results of the UNIX `time` command for overall execution time (*real*), time spent in the user process (*user*), and time spent in system calls (*system*).

As expected, in the case of using the compute cluster almost no time is spent for the actual processing of the workflow compared to the local execution cases, so the values for user and system time of the cluster benchmark run may act as indicators for the time used for executing the workflow itself (nonetheless, as explained above, a small overhead also exists in the cluster case). The overall execution time varies due to different cluster allocation patterns during the benchmark.

For the local execution benchmarks, the case for eight processes shows the expected results. The overall time is almost cut in halve while user and system time differ only slightly. For a higher

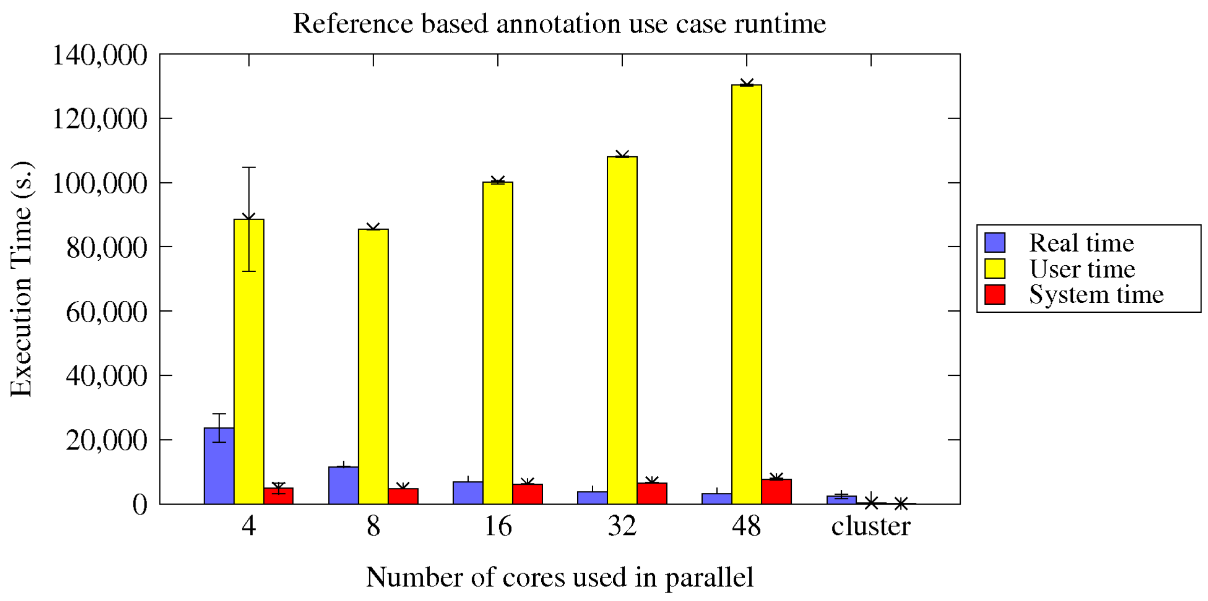


Figure 5.5: Absolute runtime for the reference based annotation use case. Each data set shows the mean real time, user time, and system time. Error bars indicate the minimum and maximum time for all 5 iterations.

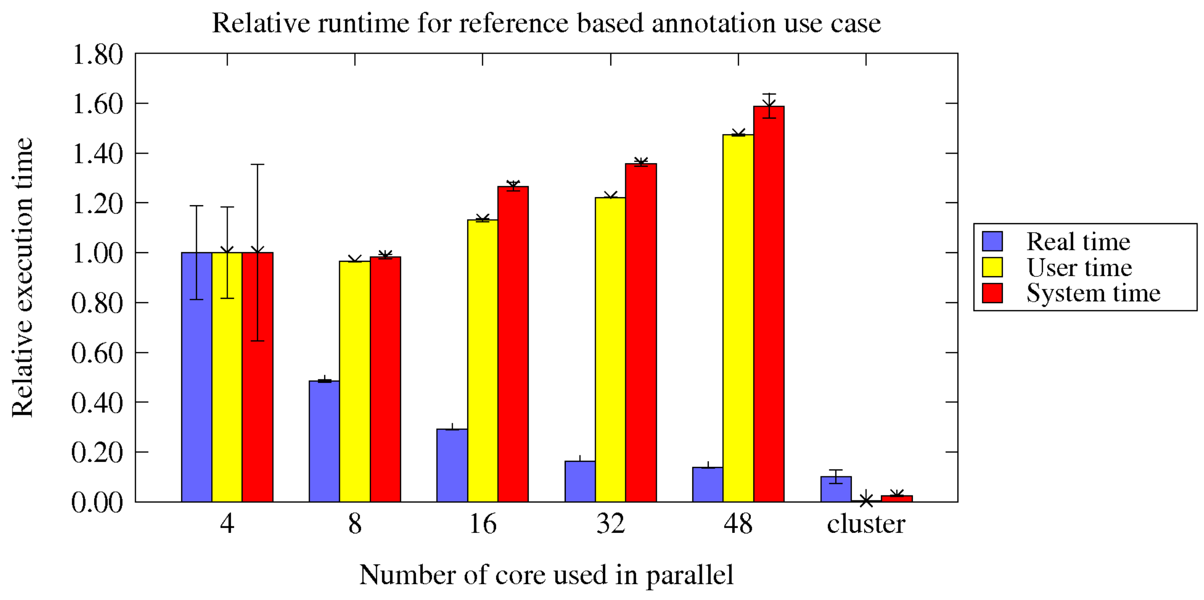


Figure 5.6: Relative runtime of the reference based annotation use case using the benchmark run with 4 cores as reference values.

number of processes, the overall time is still significantly reduced but higher amounts of relative user and system time results in a reduction factor of less than two. The exact reason is unknown but it is probably related to the high number of parallel execution threads and necessary context switches. To eliminate the need to disc I/O, two threads are used for each started process to handle input and output streams. With 16 processes running in parallel, 1*16 (processes itself) + 2*16 (input + output) threads are used which clearly exceeds the number of available real cores (24 real, 48 virtual) on the benchmark host. The additional context switches, I/O handling, and thread management probably results in the effects visible in the relative runtime chart.

As a conclusion of this use case, *Conveyor*'s runtime obviously scales well with the number of parallel processes as long as enough resources are present for handling the additional threads.

5.4.3 Comparative genomics

Using the same set of genomes as described in the section before, five benchmark runs of the comparative genomics workflow were carried out for a different number of local cores and the compute cluster. The average run times are show in table 5.3 and figures 5.7 and 5.8 present the results as barcharts.

No. cores	Real time [s.]	User time [s.]	System time [s.]
2	4807.7	8526.9	701.9
4	2540.1	8600.1	720.7
8	1402.3	8860.5	778.2
12	1022.0	9019.1	625.2
16	887.1	9235.7	858.7
24	707.6	10178.4	695.0
32	729.1	10668.9	927.0
48	642.6	12586.5	760.7
cluster	686.5	202.5	30.4

Table 5.3: Results of benchmarking the comparative genomics workflow using different number of local cores and the CeBiTec compute cluster. Each row contains the number of cores, the mean values of the results of the UNIX `time` command for overall execution time (*real*), time spent in the user process (*user*), and time spent in system calls (*system*).

The results in case of cluster usage are similar to the previous use case. The relative amount of local processing time compared to overall time is larger since less tools and faster running tools (e.g. BLAST versus smaller databases) are used in this workflow. Again, the error bar of the overall run time indicates that the cluster processing time varies due to different cluster allocations during the benchmark runs.

For the local processing cases, the relative numbers again show a slight increase of the relative user and system time, if more processes are run in parallel. The overall processing time again scales well with the number of processes. As indicated by the error bars, the system times

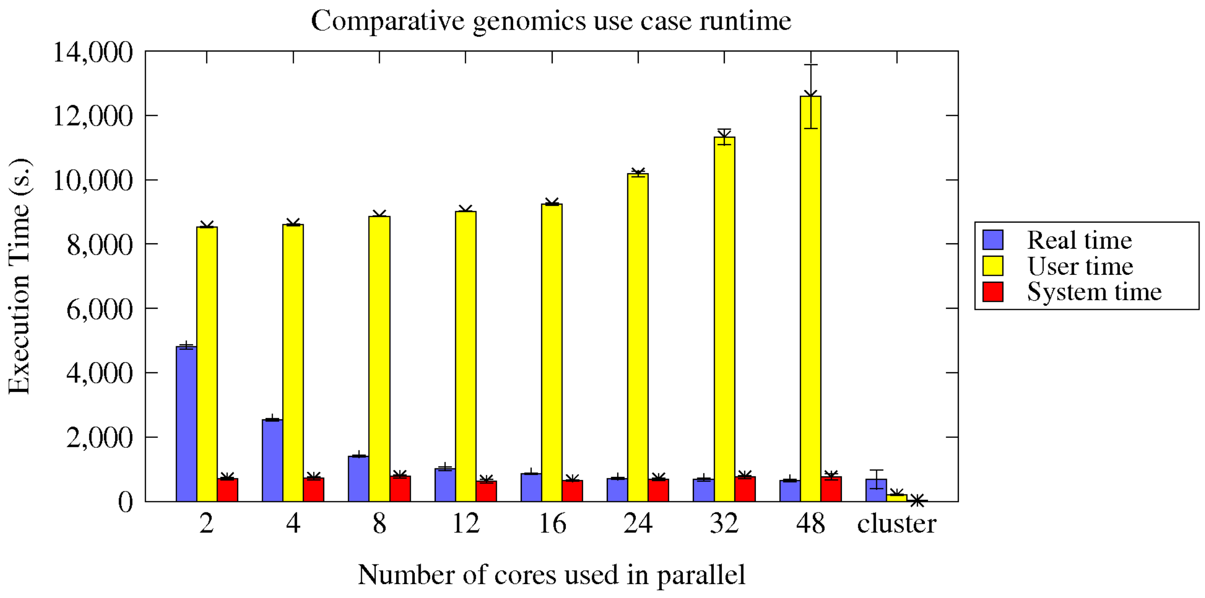


Figure 5.7: Absolute runtime of the comparative genomics use case. Each data set shows the mean real time, user time and system time. Error bars indicate the minimum and maximum time for all five iterations.

fluctuate in higher degree compared to the previous use case. A possible explanation is the different nature of the processes spawned in this use case. While the reference based annotation use case involves execution time of tens of seconds or several minutes for external tools, the BLAST and *Muscle* processes terminate earlier, often within tens of a second. This puts additional pressure to the operating system.

After all, again *Conveyor* scales quite well, until the host is saturated with processes and threads.

5.4.4 NGS data processing

In contrast to the other use cases, this use case has been executed in a different setup. The amount of data in the corresponding lane exceeds 17 GB, the time needed for transfering the data over the network would make up a substantial part of the processing time.

Instead, the workflow has been executed on the host storing the data itself (8x Quad-core AMD 8356 running at 2.3 GHz, 256 GB RAM). Since no external tools are involved in the workflow, its processing time only depends on file IO and the local resources.

Table 5.4 shows the processing time of the NGS data processing workflow. While the overall processing time was 435m 46s (26.146 s), the accumulated processing time (*user* + *sys*) of all threads was 1944m 28s (116.668 s). Multithreading thus leads to a reduction of the overall processing time by a factor of at most 4.45 (the real speed-up factor may be lower, since

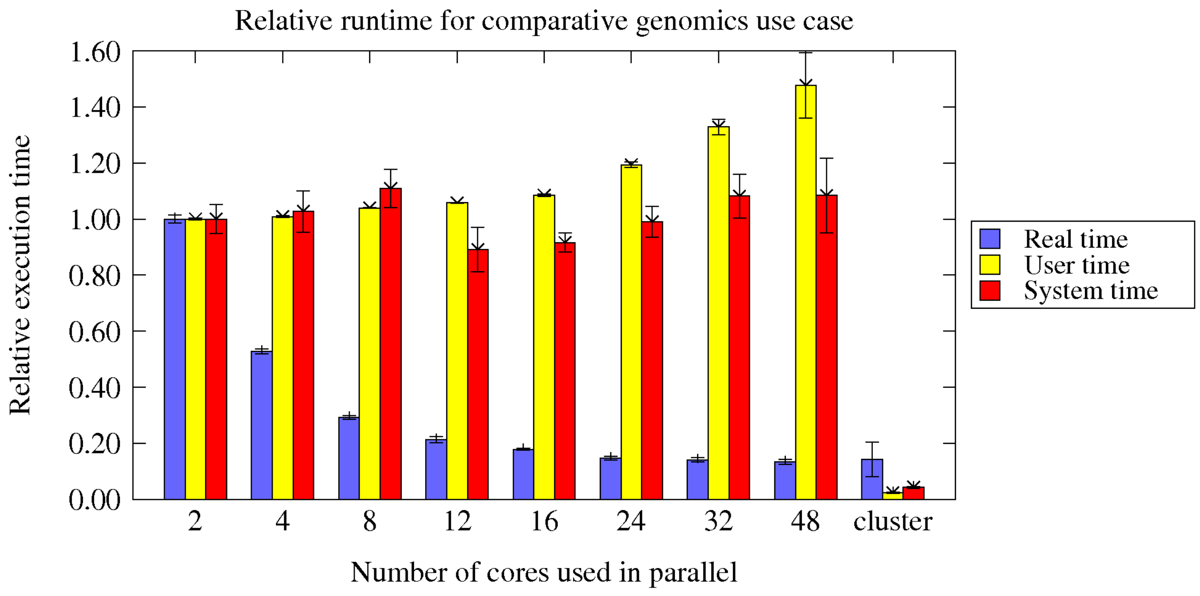


Figure 5.8: Relative runtime of the comparative genomics use case, using the benchmark run with 2 cores as reference values.

`time` results		
real	*user*	*sys*
435m 46s	1583m 53s	360m 35s

Table 5.4: Runtime of the NGS data processing workflow, as measured with the UNIX `time` command.

multithreading introduces system overhead for synchronization).

To determine whether the overall execution time may be reduced even further, one workflow run was closer examined. The *GraphRun* command line utility offers an option that prints statistics for each node and its endpoints, including the number of elements passed and pending. This reveals that almost no elements are pending in the input endpoints' buffers. As a consequence, the performance-crucial nodes in this workflow are the input nodes that parse the reads from the Illumina sequence files. Since these nodes are mostly I/O-bound, no further attempt was made to speed up the workflow execution.

CHAPTER 6

Discussion

The *Conveyor* workflow library presented in the previous chapter is a novel approach to workflow processing, and adds a viable alternative to the workflow systems presented in chapter 2. Similar to Kepler and Pegasus, *Conveyor* implements a generic workflow engine that is not bound to a certain application domain. Although the use cases and example shown in this thesis are all related to bioinformatics, the core, processing and definition components that make up the engine are suitable for any kind of data. In contrast to almost all workflow systems mentioned in chapter 2, *Conveyor* offers to operate on a finer granularity, and allows manipulation of data on the level of single integers or boolean values. Kepler is the only noticable exception of the presented systems, since it is also suitable to work on a similar level. Higher-order functionality in *Conveyor* is available by the use of *compound node types*, that refer to complete workflows of their own instead of a single processing steps. Many, but not all other workflow engines offer a similar concept, e.g. nested workflows in Taverna.

The truely unique feature of *Conveyor* is the type system for data and node types, which allows working with generic types and greatly improves the reusability of plugins. No other workflow engine examined during the course of this thesis offers a comparable setup. All requirements listed in chapter 2 for a modern workflow system are fulfilled by *Conveyor*:

Data-driven processing is implemented by passing objects between node instances, thus eliminating the need to use file-based transfers. The type system used for the object-oriented approach is also the key for the *extensibility* of Conveyor. Interfaces, inheritance and the use of generic types allow developers to decouple their implementations, and bundle them into plugins for the easy extension of Conveyor. The programming language may be chosen from a large selection of languages by the use of the .NET environment for Conveyor.

External *application*s, like many analysis tools for bioinformatics, can be *integrated* via wrapper node types into Conveyor. Results of these tools can be parsed and provided for further processing in an object-oriented way. An extensible layer controls how external tools are executed (not shown in this work), and implementations for local execution or distribution on a DRMAA compliant cluster exist. A system administrator may select an instance of this layer for each external tool used. *Security* and *discretion* of data thus only depend on the setup and the tools used. In the most secure setup, no data has to leave a local computer, e.g. by calling a web-service.

Finally, *Conveyor* is designed for *integration* into other applications due to its component based architecture. A number of clients have been presented in chapter 4.

The use cases described in chapter 5 demonstrate the suitability of *Conveyor* for processing bioinformatics data. Legacy applications can easily be integrated, and execution of external applications scales well with the number of processors available. Large data volumes, like a lane of Illumina GA IIx sequence reads (as in the case of the NGS use case with a size of 18 GB of input data), can easily be processed by *Conveyor*. Its design does not impose a limit on the amount of data a workflow can process[1].

A number of performance optimizations have been shown in the implementation chapter. Bottlenecks have been identified, and solutions have been proposed and implemented. For each optimization, its impact on the presented example workflow has been demonstrated.

The presented development optimizations aim at the plugin developer and simplify the task of implementing new node types. With concepts like *immediate processing nodes* or simplifications for input node types, a developer only spends a minimal effort into implementing new node types.

Summarizing the previous chapters, the desired features and requirements are implemented in the *Conveyor* library and its pratical use for solving bioinformatics problems has been illustrated.

[1]Certain limits for RAM size of a .NET/Mono process exists, e.g. the amount of physical and virtual RAM available and the amount of RAM the integrated garbage collector can handle.

CHAPTER 7

Summary & Outlook

The *Conveyor* workflow library is a mature and highly efficient library for the management and execution of scientific workflows. It offers unique features like an object-oriented type system, extensibility by plugins, and easy integration into higher-level applications.

Two different purposes are intended for *Conveyor*:

De-facto engine for execution of workflows in new CeBiTec applications A number of new bioinformatics applications are currently under development at the CeBiTec like the already introduced MGX metagenomics data management tool. To reduce the redundant development of central features, *Conveyor* is the designated workflow engine for these applications. Application developers thus only have to take care for data import, data export, and specific processing. Existing plugins can be shared between similar applications.

Replacement for Perl-based scripts A significant amount of time has been spent into implementing ad-hoc Perl scripts or similar software for one-time applications. With the available bioinformatics plugins, at least parts of these scripts can be replaced by workflows. This allows a more efficient design and execution of one-time tasks.

At the time of writing this thesis, the *Conveyor* system includes 25 plugins with over 300 node types. A complete list is available in the appendix.

With the *Conveyor* library implementation being finished by now, more time can be spent for implementing new plugins or improving other aspects of *Conveyor*. The following list gives an (incomplete) overview of planned features:

Netbeans Platform based workflow designer The current GUI workflow designer is implemented as a Java Web Start application and a number of limitations have inspired its redesign as a Netbeans Platform application. This more modular approach will improve the extensibility of the designer dramatically and allow for an easier distribution and maintance of its components. The design for this new application is already finished, and some core components like the definition of node types are already implemented.

Improved user interfaces for Conveyor2Go The GUI screenshots of Conveyor2Go in section 4.9 show the prelimiminary version of a prove-of-concept implementation. The interfaces will be redone prior to publishing Conveyor2Go for production usage.

Support for read mapping software used for RNA-Seq and resequencing A lot of sequence data is generated for RNA sequencing, and sequencing of species closely related to a species with a known genomic sequence. Both cases require software that is able to map a sequence read to a given reference genome. Initial support for software has been developed during a student programming course during the winter term 2010/2011 and it will become an official *Conveyor* module in the near future.

Improved handling of primitive data types Primitive types like boolean values, integers, or floating point values are currently implemented by wrapper data types. The code base for node types operating on these wrappers contains a large degree of duplicated code and conversions between types are not always implemented yet. A new library for handling these primitive types can certainly further improve the overall code quality of *Conveyor*.

EMBOSS[RLB00] software support *EMBOSS* is a collection of various bioinformatics tools covering many different aspects. A unique feature of EMBOSS is a description language (*AJAX/ACD*) that is used to explain the parameters and input and output types of these tools. Based on these description, a *Conveyor* plugin is planned that parses the description, detects which tools are available, and creates virtual node types integrating these tool into a workflow.

Conveyor2Go web resources Using the technology described in section 4.9, a web-based platform is currently developed to allow users to search for existing workflows, upload their own solutions to bioinformatics tasks, and to execute one of the available workflows on the CeBiTec compute cluster.

Conveyor is also suitable for student programming courses as mentioned above, and two of these courses have already been held at the CeBiTec. They emphasized on both aspects of *Conveyor*: plugin development and workflow design.

As a conclusion, despite being intended as a tool for end-users in the beginning, *Conveyor* has become a powerful tool for bioinformatics developers and will definitely change the way software is created at the CeBiTec (and hopefully at other places, too).

APPENDIX A

Conveyor.Processing.IProcessor interface

```
namespace Conveyor.Processing
{
    /// <summary>
    /// Defines the common interface of all processor classes that
    /// may be used within BioGraph. A processor’s main purpose is
    /// processing a biograph instance, either a complete graph or
    /// a part of a biograph forest.
    ///
    /// Since the processor instance may be located on a remote host,
    /// the interface is restricted to classes that are serializable,
    /// e.g. no direct passing of Xml elements.
    /// </summary>
    public interface IProcessor : IDisposable
    {
        /// <summary>
        /// The graph processed by this processor instance.
        ///
        /// An attempt to set a new graph while another graph is currently
        /// processed should result in an InvalidOperationException.
        /// </summary>
        Graph Graph {get; set;}

        /// <summary>
        /// Returns the number of nodes added to this processor.
        /// </summary>
        /// <returns></returns>
        int NumberOfNodes();

        /// <summary>
        /// Processes the biograph instance until it is either completely
/// finished
/// </summary>
void StartProcessGraph();

/// <summary>
/// Validates the BioGraph instance set with SetGraph(). If validation
/// fails this methods throws an exception.
/// </summary>
void ValidateGraph();

/// <summary>
/// Returns the current state of the Graph and the processor.
/// </summary>
/// <returns></returns>
GraphState GetState();

/// <summary>
/// Returns the last exception that occured in the node with the given
/// ID or null if no exception has occured yet.
/// </summary>
/// <returns></returns>
Exception GetError(long nodeId);

/// <summary>
/// Returns the output type of the given node.
/// </summary>
/// <param name="nodeId"></param>
/// <returns></returns>
string GetOutputType(long nodeId);

/// <summary>
/// Returns the output of the given node.
/// </summary>
/// <param name="nodeId"></param>
/// <returns></returns>
/// <exception cref="InvalidNodeIDException">thrown if the given node
/// exists and is not an output node</exception>
byte[] GetOutput(long nodeId);

/// <summary>
/// Sets the stream the output node with the given ID should use for
/// storing its data.
/// </summary>
/// <param name="nodeId"></param>
/// <param name="stream"></param>
void SetOutputStream(long nodeId, Stream stream);

/// <summary>
/// Returns a list the all output nodes' ids.
/// </summary>
/// <returns></returns>
List<long> GetOutputNodes();
```

```
        /// <summary>
        /// Shutdown the processor.
        /// </summary>
        void Terminate();

        /// <summary>
        /// Returns an XML document describing the status of all nodes.
        /// </summary>
        /// <returns></returns>
        IEnumerable<NodeReport> GetGraphStatus();

        /// <summary>
        /// Returns the given node if is exists in this processor.
        /// </summary>
        /// <param name="nodeId">unique identifier of node to search
        /// for</param>
        /// <returns></returns>
        Node GetNode(long nodeId);

        /// <summary>
        /// Waits until a processor terminates.
        /// </summary>
        void WaitForTermination();

        /// <summary>
        /// Returns an iterator over all nodes in the graph.
        /// </summary>
        /// <returns></returns>
        IEnumerable<Node> GetNodes();
    }
}
```

APPENDIX B

Implemented plugins

The following sections lists all plugins implemented for *Conveyor* at the time of writing this thesis. The information has been generated by converting the plugin information file descripted in section 4.8.1 to LATEXusing an XSL-T processor.

B.1 Conveyor.Alignment

Nodes and data types for generic alignments, mostly produced by external tools

Data types

Conveyor.Alignment.Alignment<T> - no description given -

Conveyor.Alignment.AnnotatableAlignment<T> - no description given -

Conveyor.Alignment.EmptyAlignment<T> - no description given -

Conveyor.Alignment.SimpleAlignment<T> - no description given -

Node types

Conveyor.Alignment.AlignmentMembers<T>

converts an alignment to a list of its member sequences

Endpoints:

← **input (Conveyor.Alignment.Alignment<T>)** - no description given -
→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.Alignment.ConcatAAAlignment<T,U,V,W>

concatenates the sequences of two alignments and returns the resulting alignment

Endpoints:

← **first (T)** first alignment
← **second (V)** second alignment
→ **result (Conveyor.Alignment.SimpleAlignment<Conveyor.BioinformaticsTypes.SimpleAASequence>)**
resulting alignment

Configuration items:

linker (literal string) linker sequence to put into the alignment

Conveyor.Alignment.ConcatDNAAlignment<T,U,V,W>

concatenates the sequences of two alignments and returns the resulting alignment

Endpoints:

← **first (T)** first alignment
← **second (V)** second alignment
→ **result (Conveyor.Alignment.SimpleAlignment<Conveyor.BioinformaticsTypes.SimpleDNASequence>)**
resulting alignment

Configuration items:

linker (literal string) linker sequence to put into the alignment

Conveyor.Alignment.CreateAlignment<T>

converts a list of sequences to an alignment

Endpoints:

← **input (Conveyor.List.List<T>)** - no description given -
→ **output (Conveyor.Alignment.SimpleAlignment<T>)** - no description given -

Conveyor.Alignment.CreateEmptyAAAlignment

creates an empty amino acid alignment

Endpoints:

→ **output (Conveyor.Alignment.EmptyAlignment<Conveyor.BioinformaticsTypes.SimpleAASequence>)** - no description given -

Conveyor.Alignment.CreateEmptyDNAAlignment

creates an empty nucleotide alignment

Endpoints:

→ **output (Conveyor.Alignment.EmptyAlignment<Conveyor.BioinformaticsTypes.SimpleDNASequence>)** - no description given -

Conveyor.Alignment.DifferentSites<T>

calculates the number of different sites (S) in the alignment

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** alignment to process
→ **sites (Conveyor.Math.Integer)** the number of different sites

Configuration items:

ignoreUnknown (boolean) ignores columns with unknown bases

Conveyor.Alignment.GetNthMember<T>

extracts the Nth member of an alignments

Endpoints:

← **input (Conveyor.Alignment.Alignment<T>)** - no description given -
→ **output (T)** - no description given -

Configuration items:

memberIndex (32 bit signed integer) index of the member to extract, e.g. 0 for first sequence, 1 for second

Conveyor.Alignment.MeanPairWiseDifferenSites<T>

calculates the average number of different sites between to two sequences (Pi)

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** alignment to process
→ **pi (Conveyor.Math.Double)** the average number of different sites

Configuration items:

ignoreUnknown (boolean) ignore columns with unknown bases

Conveyor.Alignment.OrderAlignment<T>

reorders the sequences of an alignment to have a certain sequences as first

Endpoints:

← **alignmentIn (Conveyor.Alignment.Alignment<T>)** alignment to process
← **sequenceName (Conveyor.Text.String)** name of sequence to put as first in the result
→ **alignmentOut (Conveyor.Alignment.Alignment<T>)** resulting alignment

Configuration items:

checkBlastNames (boolean) also checks additionally for blast-compliant names (lcl|XYZ and XYZ)

Alignment<T>

returns the number of columns of the alignment

Endpoints:

← **input (Conveyor.Alignment.Alignment<T>)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Alignment<T>

returns the number of sequences in the alignment

Endpoints:

← **input (Conveyor.Alignment.Alignment<T>)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

B.2 Conveyor.Alignment.Applications

Wrappers for external application for generating sequence alignments

Node types

Conveyor.Alignment.Applications.ClustalWAA<T>

creates a DNA alignment with ClustalW

Endpoints:

← **seqList (Conveyor.List.List<T>)** sequences to align
→ **alignment (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaAASequence>)** the aligned sequences

Conveyor.Alignment.Applications.ClustalWDNA<T>

creates a DNA using ClustalW

Endpoints:

← **seqList (Conveyor.List.List<T>)** sequences to align
→ **alignment (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaDNASequence>)** the aligned sequences

Conveyor.Alignment.Applications.GBlocksAA<T>

processes AA alignments with Gblocks

Endpoints:

← **alignmentIn (Conveyor.Alignment.Alignment<T>)** alignment to process
→ **alignmentOut (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaAASequence>)** alignment with masked positions

Conveyor.Alignment.Applications.GBlocksDNA<T>

processes DNA alignments with Gblocks

Endpoints:

← **alignmentIn (Conveyor.Alignment.Alignment<T>)** alignment to process
→ **alignmentOut (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaDNASequence>)** alignment with masked positions

Conveyor.Alignment.Applications.MuscleAA<T>

calculates a AA alignment based on muscle

Endpoints:

← **seqList (Conveyor.List.List<T>)** sequences to align
→ **alignment (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaAASequence>)** the generated alignment

Configuration items:

findDiags (boolean) Optimize speed for similar sequences by finding diagonals
maxIters (32 bit signed integer) Maximum number of iterations

Conveyor.Alignment.Applications.MuscleDNA<T>

calculates a DNA alignment using muscle

Endpoints:

← **seqList (Conveyor.List.List<T>)** sequences to align
→ **alignment (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaDNASequence>)** the generated alignment

Configuration items:

findDiags (boolean) Optimize speed for similar sequences by finding diagonals
maxIters (32 bit signed integer) Maximum number of iterations

Conveyor.Alignment.Applications.RevTrans<T,U>

converts a protein alignment to a codon-based DNA alignment

Endpoints:

← **dnaSequences (Conveyor.List.List<T>)** the dna sequences used to generate the protein alignments
← **proteinAlignment (Conveyor.Alignment.Alignment<U>)** the protein alignment
→ **dnaAlignment (Conveyor.Alignment.SimpleAlignment<Conveyor.BioIO.FastaDNASequence>)** the resulting codon-based DNA alignment

Conveyor.Alignment.Applications.RevTransDB<T,U>

converts a protein alignment to a codon-based DNA alignment

Endpoints:

← **dnaDB (Conveyor.Blast.BlastDB<U>)** the dna sequence database
← **proteinAlignment (Conveyor.Alignment.Alignment<T>)** the protein alignment
→ **dnaAlignment (Conveyor.Alignment.SimpleAlignment<U>)** the resulting codon-based DNA alignment

B.3 Conveyor.Alignment.IO

IO-related classes for alignment file formats

Node types

Conveyor.Alignment.IO.ReadAAPhylipFile

reads AA sequence alignments in phylip format from a file

Endpoints:

→ **alignmentOut (Conveyor.Alignment.Alignment<Conveyor.BioinformaticsTypes.AASequence>)** the read alignment

Configuration items:

inputFile (input file) the file to parse

Conveyor.Alignment.IO.ReadDNAPhylipFile

reads DNA sequence alignments in phylip format from a file

Endpoints:

→ **alignmentOut (Conveyor.Alignment.Alignment<Conveyor.BioinformaticsTypes.DNASequence>)** the read alignment

Configuration items:

inputFile (input file) the file to parse

Conveyor.Alignment.IO.WritePhylipFile<T>

creates a phylip formatted file for storing (multiple) alignments

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** the alignments to write to the file

Conveyor.Alignment.IO.ReadAAStockholmFile

Reads amino acid alignments from a Stockholm formatted file

Endpoints:

→ **alignmentOut (Conveyor.Alignment.AnnotatableAlignment<Conveyor.BioinformaticsTypes.SimpleAASequence>)** the read alignment

Configuration items:

inputFile (input file) the file to parse

B.4 Conveyor.Alignment.Pairwise

Method and data classes for generating pairwise alignments within Conveyor

Data types

Conveyor.Alignment.Pairwise.AlignedAASequence<T> - no description given -

Conveyor.Alignment.Pairwise.AlignedDNASequence<T> - no description given -

Conveyor.Alignment.Pairwise.AlignedSequence<T> - no description given -

Conveyor.Alignment.Pairwise.AlignmentScoring<T> - no description given -

Conveyor.Alignment.Pairwise.PairwiseAAAlignment<S,T> - no description given -

Conveyor.Alignment.Pairwise.PairwiseAlignment<S,T,U> An alignment containing two sequences

Conveyor.Alignment.Pairwise.PairwiseDNAAlignment<S,T> - no description given -

Conveyor.Alignment.Pairwise.UniformScoring<T> An alignment scoring scheme with one uniform score for matching and mismatching

Node types

Conveyor.Alignment.Pairwise.BlastDNAScoringSchema

BLAST-like scoring schema for DNA sequence: match 1, mismatch -3, gap open -5, gap extend -2

Endpoints:

→ **output (Conveyor.Alignment.Pairwise.UniformScoring<Conveyor.BioinformaticsTypes.DNASequence>)** - no description given -

Conveyor.Alignment.Pairwise.GetMatchingRange<T,U,V,W>

returns the maximum range of matches of a pairwise alignment

Endpoints:

← **alignment (V)** alignment to process;
→ **start (Conveyor.Math.ULong)** start position (1-based) of the matching part
→ **stop (Conveyor.Math.ULong)** stop position (1-based) of the matching part

Conveyor.Alignment.Pairwise.GlobalDNAAlignment<S,T,V>

Creates a global alignment of two sequence

Endpoints:

← **first (S)** first sequence
← **scoring (V)** scoring schema for alignment
← **second (T)** second sequence
→ **alignment (Conveyor.Alignment.Pairwise.PairwiseDNAAlignment<S,T>)** alignment of both sequences

Configuration items:

freeEndGaps (boolean) allows overhanging bases at the ends of the sequence

Conveyor.Alignment.Pairwise.MatchingPositions<T>

calculates the number of matching positions in an alignment

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** alignments to process
→ **matches (Conveyor.Math.Integer)** number of matches

Conveyor.Alignment.Pairwise.NumberOfMatches<S,T,U>

calculates the number of matching positions in the alignment

Endpoints:

← **alignment (Conveyor.Alignment.Pairwise.PairwiseAlignment<S,T,U>)** alignment to process
→ **matchingPositions (Conveyor.Math.ULong)** number of matching positions

Conveyor.Alignment.Pairwise.UniformDNAScoringSchema

DNA scoring scheme using uniform scores for matches/mismatches

Endpoints:

→ **output (Conveyor.Alignment.Pairwise.UniformScoring<Conveyor.BioinformaticsTypes.DNASequence>)**
- no description given -

Configuration items:

gapExtendScore (IEEE double value) score for extending a gap
gapOpenScore (IEEE double value) score for opening a gap
matchScore (IEEE double value) score for match
mismatchScore (IEEE double value) score for mismatch

B.5 Conveyor.Annotation

Definition of annotatable objects

Data types

Conveyor.Annotation.Annotatable - no description given -

Conveyor.Annotation.AnnotationField a definition of a field of an annotation

Node types

Conveyor.Annotation.CopyAnnotationField<T,U,V>

copies the content of a given annotation field from a template to the input sequence, or the default value if the field is not set in the template

Endpoints:

← **defaultValue (V)** the default value if the annotation field is not set in the template
← **field (Conveyor.Annotation.AnnotationField)** the annotation field to copy
← **input (T)** the sequence to add the annotation to
← **template (U)** the template sequence to get the annotation value from
→ **output (T)** the sequence after adding the annotation

Conveyor.Annotation.DeleteAnnotationField<T>

removes the given annotation field if it exists

Endpoints:

← **firstArgument (T)** the first argument of the operation
← **secondArgument (Conveyor.Annotation.AnnotationField)** the second argument of the operation
→ **result (T)** the result of the operation

Conveyor.Annotation.GetAnnotationField<T>

returns the current value of an annotation field or results in an error if no such field exists

Endpoints:

← **annotatable (Conveyor.Annotation.Annotatable)** annotatable object to examine
← **fieldName (Conveyor.Annotation.AnnotationField)** field to retrieve
→ **value (T)** the value of the field

Conveyor.Annotation.HasAnnotationField

Returns true if the annotatable object contains an annotation with the given name

Endpoints:

← **annotation (Conveyor.Annotation.Annotatable)** annotatable object to examine
← **fieldName (Conveyor.Annotation.AnnotationField)** field to check
→ **hasAnnotation (Conveyor.Logic.Boolean)** flag to indicate whether the annotation contains the field

Conveyor.Annotation.SelectAnnotationField

selects one of the available annotation fields

Endpoints:

→ **output (Conveyor.Annotation.AnnotationField)** - no description given -

Configuration items:

field (- unknown type -) the field to use

Conveyor.Annotation.SetAnnotationField<T,U>

sets an annotation field in a annotatable object. Old values are discarded if existing.

Endpoints:

← **fieldName (Conveyor.Annotation.AnnotationField)** the name of the field to set
← **input (T)** the annotatable object to add a value to
← **value (U)** the value to set
→ **output (T)** the annotatable object after adding the value

B.6 Conveyor.BioIO

IO-related classes for bioinformatics file formats

Data types

Conveyor.BioIO.FastaAASequence - no description given -

Conveyor.BioIO.FastaDNASequence - no description given -

Conveyor.BioIO.FastaSequenceBase<T> - no description given -

Conveyor.BioIO.ParseableSequence - no description given -

Node types

Conveyor.BioIO.FastaWriter

dumps sequences in fasta format

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -

Conveyor.BioIO.Fasta.ReadAAFastaFile

reads AA sequences from a given fasta file

Endpoints:
→ **sequenceOut (Conveyor.BioIO.FastaAASequence)** the sequences

Configuration items:
inputFile (input file) the file to parse

Conveyor.BioIO.Fasta.ReadDNAFastaFile

reads DNA sequences from the given fasta file

Endpoints:
→ **sequenceOut (Conveyor.BioIO.FastaDNASequence)** the sequences

Configuration items:
inputFile (input file) the file to parse

Conveyor.BioIO.ReadFastaQualFiles

Reads the fasta and quality file for DNA sequences

Endpoints:
→ **seq (Conveyor.QualitySequence.SimpleDNAQSequence)** the read sequences

Configuration items:
qualFile (input file) file containing the quality values in fasta format
seqFile (input file) file containing the sequence in fasta format

Conveyor.BioIO.WriteQualFile<T>

exports the quality information of quality sequences to an external file

Endpoints:
← **seq (T)** Sequence to export

Conveyor.BioIO.FastQ.ReadFastQFile

read the given FastQ file and produces sequences with quality information

Endpoints:
→ **sequence (Conveyor.QualitySequence.SimpleDNAQSequence)** a read sequence with quality information

Configuration items:
inputFile (input file) file to read as input
qualityEncoding (- unknown type -) Encoding of quality values in FastQ quality line

Conveyor.BioIO.FastQ.WriteFastQFile<T>

writes sequences with quality information as FastQ files

Endpoints:
← **sequence (T)** the sequences to be written

Configuration items:
encoding (- unknown type -) the encoding to use for the quality values

B.7 Conveyor.BioIO.SFF

Conveyor plugin for reading and writing SFF files

Data types

Conveyor.BioIO.SFF.ClippedSFFSequence - no description given -

Conveyor.BioIO.SFF.FlowgramSequence - no description given -

Conveyor.BioIO.SFF.SFFSequence - no description given -

Node types

Conveyor.BioIO.SFF.ClipSFFSequence

clips adapter sequences and parts with low quality from an SFF sequence (based on information in sequence header

Endpoints:
← **input (Conveyor.BioIO.SFF.SFFSequence)** - no description given -
→ **output (Conveyor.BioIO.SFF.ClippedSFFSequence)** - no description given -

Conveyor.BioIO.SFF.ReadSFFFile

parses the given SFF file and iterates over the sequence in that file

Endpoints:
→ **sequence (Conveyor.BioIO.SFF.SFFSequence)** a sequence read from the SFF file

Configuration items:
sffFile (input file) The SFF file to parse

Conveyor.BioIO.SFF.WriteSFFFile

creates a SFF file from sequences

Endpoints:
← **sequence (Conveyor.QualitySequence.IQualitySequence)** sequence to write to SFF file

B.8 Conveyor.BioIO.Solexa

Data types

Conveyor.BioIO.Solexa.SolexaRead - no description given -

Node types

Conveyor.BioIO.Solexa.GetClipPosition

returns the position of the first 'B' in the quality scores (indicating low quality until end of read

Endpoints:

← **input (Conveyor.BioIO.Solexa.SolexaRead)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

Conveyor.BioIO.Solexa.GetQualityCheckFlag

returns the quality check flag parsed from the Solexa read

Endpoints:

← **input (Conveyor.BioIO.Solexa.SolexaRead)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.BioIO.Solexa.IndexSequence

Generates string of Solexa default index sequences used for multiplex sequencing

Endpoints:

→ **output (Conveyor.Text.String)** - no description given -

Configuration items:

index (- unknown type -) Index sequence

Conveyor.BioIO.Solexa.IsUsableRead

checks whether the read is usable (does not contain a stretch of B in the quality values of certain length

Endpoints:

← **input (Conveyor.BioIO.Solexa.SolexaRead)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

maxBStretchLength (64 bit unsigned integer) maximum length of B stretch at end of quality values to accept a read

Conveyor.BioIO.Solexa.PairedEndLinkerSequence

Generates string of Solexa default paired end linker sequence (5'-3')

Endpoints:

→ **output (Conveyor.Text.String)** - no description given -

Configuration items:

useLinker1 (boolean) Return linker sequence 1

Conveyor.BioIO.Solexa.ParseSolexaQSeqFile

Parses the files for all single read of a given lane in a given directory

Endpoints:

→ **read (Conveyor.BioIO.Solexa.SolexaRead)** A read from the given lane

Configuration items:

basecallDirectory (literal string) Directory containing the Solexa basecalls
lane (32 bit signed integer) Lane (1-8) to read
readIndex (32 bit signed integer) Index of sub read to parse (e.q. 1-3 for indexed paired end reads)

B.9 Conveyor.BioinformaticsTypes

Base type definitions for bioinformatic data objects like sequences etc.

Data types

Conveyor.BioinformaticsTypes.AASequence - no description given -

Conveyor.BioinformaticsTypes.ConcatenatedSequence<T,U> A sequence composed of two sub sequence

Conveyor.BioinformaticsTypes.DNASequence - no description given -

Conveyor.BioinformaticsTypes.Homology - no description given -

Conveyor.BioinformaticsTypes.InsertedSequence<T,U> A DNA sequence created by inserting one sequence into another at a given position

Conveyor.BioinformaticsTypes.ReadOnlyDerivedSequence<T> - no description given -

Conveyor.BioinformaticsTypes.ReverseComplementedSequence<T> Reverse complement of a DNA sequence

Conveyor.BioinformaticsTypes.ReversedSequence<T> Reversed sequence of a given DNA sequence

Conveyor.BioinformaticsTypes.Sequence - no description given -

Conveyor.BioinformaticsTypes.SequenceBase<T> - no description given -

Conveyor.BioinformaticsTypes.SimpleAASequence - no description given -

Conveyor.BioinformaticsTypes.SimpleDNASequence - no description given -

Conveyor.BioinformaticsTypes.SimpleTranslatableSequence<T> - no description given -

Conveyor.BioinformaticsTypes.SplicedSequence<T> DNA sequence with a part spliced out

Conveyor.BioinformaticsTypes.Translatable<T> - no description given -

Conveyor.BioinformaticsTypes.ISubSequenceable<T> - no description given -

Node types

Conveyor.BioinformaticsTypes.ConcatenateSequences<T,U>

creates a new sequence by concatenating two given ones

Endpoints:

← **first (T)** first sequence
← **name (Conveyor.Text.String)** name of the concatenated sequence
← **second (U)** second sequence
→ **concatenated (Conveyor.BioinformaticsTypes.ConcatenatedSequence<T,U>)** result of concatenation

Conveyor.BioinformaticsTypes.CreateAASequence

creates a AA sequence data object from a name and a sequence

Endpoints:

← **name (Conveyor.Text.String)** - no description given -
← **seq (Conveyor.Text.String)** - no description given -
→ **sequence (Conveyor.BioinformaticsTypes.SimpleAASequence)** - no description given -

Conveyor.BioinformaticsTypes.CreateDNASequence

creates a DNA sequence data object from a name and a sequence

Endpoints:

← **name (Conveyor.Text.String)** - no description given -
← **seq (Conveyor.Text.String)** - no description given -
→ **sequence (Conveyor.BioinformaticsTypes.SimpleDNASequence)** - no description given -

Conveyor.BioinformaticsTypes.CreateTranslatableSequence

creates a translatable DNA sequence data object from a name and a sequence

Endpoints:

← **name (Conveyor.Text.String)** - no description given -
← **seq (Conveyor.Text.String)** - no description given -
→ **sequence (Conveyor.BioinformaticsTypes.SimpleTranslatableSequence<Conveyor.BioinformaticsTypes.SimpleDNAS**
- no description given -

Conveyor.BioinformaticsTypes.ExtractSequence

returns the plain textual sequence of a sequence data object

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Conveyor.BioinformaticsTypes.GetSubSequence<T,U>

returns a part of the given sequence

Endpoints:

← **from (Conveyor.Math.ULong)** - no description given -
← **sequence (T)** - no description given -
← **to (Conveyor.Math.ULong)** - no description given -
→ **subSequence (U)** - no description given -

Conveyor.BioinformaticsTypes.InsertSequence<T,U>

Inserts a sequence into another sequence at a given position

Endpoints:

← **insert (U)** sequence to be inserted
← **position (Conveyor.Math.ULong)** position to insert sequence at
← **sequence (T)** sequence to insert into
→ **result (Conveyor.BioinformaticsTypes.InsertedSequence<T,U>)** result of the insert

Conveyor.BioinformaticsTypes.MaskStopCodons

replaces stop codon in the sequence by three 'N's

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Conveyor.BioinformaticsTypes.ReverseComplementSequence<T>

Creates a new DNA sequence using the reversed complemented input sequence

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.BioinformaticsTypes.ReverseComplementedSequence<T>)** - no description given -

Conveyor.BioinformaticsTypes.ReverseSequence<T>

Creates a new DNA sequence using the reversed input sequence

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.BioinformaticsTypes.ReversedSequence<T>)** - no description given -

Conveyor.BioinformaticsTypes.SpliceSequence<T>

Returns a DNA sequence with a given parts spliced out

Endpoints:

← **from (Conveyor.Math.ULong)** start of spliced range
← **sequence (T)** sequence to process
← **to (Conveyor.Math.ULong)** end of spliced range
→ **splicedSequence (Conveyor.BioinformaticsTypes.SplicedSequence<T>)** sequence after splicing

Conveyor.BioinformaticsTypes.TranslateSequence<T>

translates a DNA sequence to amino acids

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Translatable<T>)** - no description given -
→ **output (T)** - no description given -

Homology.GetEValue

returns the expectation value of the homology or NaN if no evalue is available

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Homology)** - no description given -
→ **output (Conveyor.Math.Double)** - no description given -

Homology.GetQueryRange

returns the start and stop position the homologue query part, e.g. of the alignment for blast HSPs

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Homology)** - no description given -
→ **output (Conveyor.Pair.Pair<Conveyor.Math.Integer,Conveyor.Math.Integer>)** - no description given -

Homology.GetSubjectRange

returns the start and stop position the homologue subject part, e.g. of the alignment for blast HSPs

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Homology)** - no description given -
→ **output (Conveyor.Pair.Pair<Conveyor.Math.Integer,Conveyor.Math.Integer>)** - no description given -

Homology.GetIdentical

returns the number of identical residues, e.g. identical bases or amino acids in blast alignments

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Homology)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

Homology.GetScore

returns the native score of the homology, e.g. the bit score of blast HSPs

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Homology)** - no description given -
→ **output (Conveyor.Math.Double)** - no description given -

Sequence.GetID

returns the identifier of the sequence

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Sequence.GetSequence

returns the text representation of the complete sequence

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Sequence.GetDescription

returns the description of the sequence (if any)

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Sequence.GetLength

returns the length of the sequence in bases or amino acids

Endpoints:

← **input (Conveyor.BioinformaticsTypes.Sequence)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

B.10 Conveyor.Biotools.Misc

Wrapper for misc binformatics tools

Node types

Conveyor.Biotools.Misc.GeneConv<T,U>

Endpoints:

← **inputAlignment (Conveyor.Alignment.Alignment<T>)** the alignment that is screened for gene conversion events
→ **alignmentList (Conveyor.List.List<Conveyor.Alignment.Alignment‘1>)** a list of sub alignments corresponding to global inner fragments, or a list of the complete alignment if no inner fragment was found
→ **pValue (Conveyor.Math.Double)** pValue result of GeneConv run

Configuration items:

minimalFragmentLength (32 bit signed integer) minimal length global inner fragments to creating sub alignments
pValueCutOff (IEEE double value) cut off for gene conversion probability (p-value cut off for inner fragments)

Conveyor.Biotools.Misc.PAML<T>

executes PAML with a phylogeny tree and an alignment

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** the alignment to process
← **tree (Conveyor.Phylogeny.PhylogenyNode)** the tree corresponding to the alignment
→ **logValue (Conveyor.Math.Double)** the log-likelihood of recombination

Configuration items:

hypoType (literal string) hypothesis to test, either M1A, M2A, M3, M7, M8 or BSA

Conveyor.Biotools.Misc.PhiPack<T>

uses PhiPack to check an alignment for recombination

Endpoints:

← **input (Conveyor.Alignment.Alignment<T>)** the alignment to use as input
→ **chiSquareMax (Conveyor.Math.Double)** chi square maximum
→ **nss (Conveyor.Math.Double)** nss result
→ **phi (Conveyor.Math.Double)** p value of recombination

Configuration items:

numberOfPermutations (32 bit signed integer) number of permutations
windowSize (32 bit signed integer) window size

Conveyor.Biotools.Misc.Reticulate<T>

executes reticulate to predict positive selection

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** DNA alignment to examine
→ **pValue (Conveyor.Math.Double)** p-value from reticulate

Conveyor.Biotools.Misc.Selecton<T>

executes selecton for a codon-based alignment and returns the likelihood for positive selection

Endpoints:

← **alignment (Conveyor.Alignment.Alignment<T>)** the alignment to use as input
→ **likelihood (Conveyor.Math.Double)** maximum likelihood of positive selection

Configuration items:

model (literal string) model to use, valid values are 'M8','M8a' and 'M7'

B.11 Conveyor.Blast

Conveyor data and node types related to NCBI Blast

Data types

Conveyor.Blast.BlastHit<T,U> - no description given -

Conveyor.Blast.BlastResult<T,U> - no description given -

Conveyor.Blast.BlastDB<T> - no description given -

Conveyor.Blast.BlastDBFile<T,U> - no description given -

Conveyor.Blast.GenericSequenceDatabase<T> - no description given -

Node types

Conveyor.Blast.BlastvsDB<T,U>

executes blast vs. a given database

Endpoints:

← **database (Conveyor.Blast.BlastDB<U>)** database to blast against
← **query (T)** query sequence to blast vs. the database
→ **result (Conveyor.Blast.BlastResult<T,U>)** blast result

Configuration items:

blastType (- unknown type -) blast sub type
chunkSize (32 bit signed integer) number of sequences to process in a single blast instance; use 1 for large databases, and larger values for small databases
iterations (32 bit signed integer) maximum number of iterations (for psiblast)
threads (32 bit signed integer) number of thread per blast instance
useComplexityFilter (boolean) activate the complexity filter (default yes)
wordSize (32 bit signed integer) word size of initial seed kmers (blastp: 3, blastn: 11, megablast: 28)

Conveyor.Blast.FilterBlastHits<T,U>

returns a list of all blast hits of a blast result with an evalue better than a given threshold

Endpoints:

← **input (Conveyor.Blast.BlastResult<T,U>)** - no description given -
→ **output (Conveyor.List.List<Conveyor.Blast.BlastHit‘2>)** - no description given -

Configuration items:

evalueCutoff (IEEE double value) evalue cut off
identityCutoff (IEEE double value) identity cut off, in relation to the length of the complete query
lengthCutoff (IEEE double value) lengths cut off, in relation to the length of the complete query

Conveyor.Blast.GetQueryFromBlastHit<T,U>

extracts the query sequence from a blast hit

Endpoints:

← **input (Conveyor.Blast.BlastHit<T,U>)** - no description given -
→ **output (T)** - no description given -

Conveyor.Blast.GetQueryFromBlastResult<T,U>

extracts the query sequence from a blast result

Endpoints:

← **input (Conveyor.Blast.BlastResult<T,U>)** - no description given -
→ **output (T)** - no description given -

Conveyor.Blast.GetSubject<T,U>

returns the database entry the blast hit refers to

Endpoints:

← **input (Conveyor.Blast.BlastHit<T,U>)** blast hit to retrieve subject from
→ **output (U)** subject sequence

Configuration items:

hitChunkSize (32 bit signed integer) number of hits to put into a single retrieve operation (for faster retrieval of subjects from ASN1 databases

Conveyor.Blast.CreateBlastDB<T>

collects sequences and creates a temporary blastable sequence database

Endpoints:

← **sequence (T)** sequence to put into the database
→ **db (Conveyor.Blast.BlastDB<T>)** collection of all input sequences as blastable database

Conveyor.Blast.GetDBEntry<T>

returns a sequence from a database identified by its name

Endpoints:

← **blastDatabase (Conveyor.Blast.BlastDB<T>)** blast database to extract sequence from
← **seqID (Conveyor.Text.String)** name or ID of sequence to be retrieved
→ **sequence (T)** sequence from database entry

Conveyor.Blast.PublicAADB

Loads public blastable protein sequence databases from configured locations

Endpoints:

→ **db (Conveyor.Blast.BlastDBFile<Conveyor.BioIO.FastaAASequence,Conveyor.BioinformaticsTypes.SimpleAASequen**
the selected database

Configuration items:

database (- unknown type -) nucleotide sequence databases available

Conveyor.Blast.PublicDNADB

Loads public blastable DNA sequence databases from configured locations

Endpoints:

→ **db (Conveyor.Blast.BlastDBFile<Conveyor.BioIO.FastaDNASequence,Conveyor.BioinformaticsTypes.SimpleDNASequenc**
the selected database

Configuration items:

database (- unknown type -) nucleotide sequence databases available

B.12 Conveyor.Comparison

Generic purpose comparison type for Conveyor

Data types

Conveyor.Comparison.ComparisonResult - no description given -

Node types

Conveyor.Comparison.IsEqual

returns true if the comparison result indicates that both elements are equal

Endpoints:

← **input (Conveyor.Comparison.ComparisonResult)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Comparison.IsLarger

returns true if the comparison result indicates that the first element was larger

Endpoints:

← **input (Conveyor.Comparison.ComparisonResult)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Comparison.IsSmaller

returns true if the comparison result indicates that the first element was larger

Endpoints:

← **input (Conveyor.Comparison.ComparisonResult)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Comparison.Reverse

reverses the comparison result

Endpoints:

← **input (Conveyor.Comparison.ComparisonResult)** - no description given -
→ **output (Conveyor.Comparison.ComparisonResult)** - no description given -

B.13 Conveyor.Core

Central library of Conveyor, containing the data and node type definitions.

Data types

Conveyor.Core.Data - no description given -

Node types

Conveyor.Core.TextOutput

simple output note that prints data objects

Endpoints:

← **input (Conveyor.Core.Data)** - no description given -

Conveyor.Core.Abort

aborts a BioGraph run if a data object reaches the input by throwing an exception that contain the data as string

Endpoints:

← **input (Conveyor.Core.Data)** input that triggers aborting the graph

Conveyor.Core.CastData<T,U>

casts the input to the output

Endpoints:

← **input (T)** - no description given -
→ **output (U)** - no description given -

Conveyor.Core.ConcatStreams<T>

Concatenates the data elements from the first and the second stream

Endpoints:

← **first (T)** first stream to use data from
← **second (T)** second stream to use data from
→ **output (T)** output of the merged streams

Conveyor.Core.Conditional<T>

processes data based on a configurable switch

Endpoints:

← **input (T)** data input
← **procIn (T)** input of the processing loop
→ **output (T)** data output, either directly passed from input or after processing on the loop
→ **procOut (T)** output to the processing loop

Configuration items:

doProcessing (boolean) enables passing of data to the processing loop

Conveyor.Core.Discard

discards all data send in

Endpoints:

← **input (Conveyor.Core.Data)** data to discard

Conveyor.Core.Duplicate<T>

Duplicates data read from input

Endpoints:

← **input (T)** - no description given -
→ **output1 (T)** - no description given -
→ **output2 (T)** - no description given -

Conveyor.Core.FeedbackProcessing<T,U>

processes all input elements and passes result back into processing

Endpoints:

← **input (T)** data to be processed
← **resultIn (U)** processing result that is either passed as input to the next iteration or passed to the output link as end result
← **startValue (U)** single input used as start element
→ **firstOut (T)** loop output value taken from input link
→ **output (U)** result after processing all inputs
→ **secondOut (U)** feedback output value, either taken from start value link or from result of the the former run

Conveyor.Core.LimitedPass<T>

limits the number of passed data objects to a given count

Endpoints:

← **input (T)** the to be passed input
→ **output (T)** the passed output

Configuration items:

limit (32 bit signed integer) the number of elements to pass

Conveyor.Core.Merge<T>

merges two streams of data without predefined order

Endpoints:

← **input1 (T)** first stream of data to get elements from
← **input2 (T)** second stream of data to get elements from
→ **output (T)** merged stream

Conveyor.Core.ParallelLoop<T,U>

Parallization of processing

Endpoints:

← **input (T)** main input of node
← **loopIn (U)** input from loop to parallize
→ **loopOut (T)** output to loop to parallize
→ **output (U)** main output of node

Configuration items:

numberOfInstances (32 bit signed integer) number of instance to create from loop; 1 for default processing, 0 for CPU saturation

Conveyor.Core.Repeat<T>

reads one element at the input link and repeats it at the output link

Endpoints:

← **input (T)** the element to repeat
→ **output (T)** repeated copies of the element read on the input link

Conveyor.Core.Replace<T,U>

replaces a data object by another object, e.g. to maintain loops

Endpoints:

← **discard (T)** dummy input to be replaced and discarded
← **replacement (U)** new data to be used instead
→ **output (U)** replaced data

Conveyor.Core.UnzipStream<T>

unzips a stream of data by passing all odd elements to the first and all even elements to the second output

Endpoints:

← **input (T)** stream to split
→ **even (T)** even elements from the stream (2, 4., 6., ...)
→ **odd (T)** odd elements from the stream (1., 3., 5., ...)

Conveyor.Core.ZipStreams<T>

Zips elements from both streams, alternating selecting elements from the input streams (starting with first stream)

Endpoints:

← **first (T)** first stream of elements
← **second (T)** second stream of elements
→ **zipped (T)** stream after zipping

Conveyor.Core.Skip<T>

skips a configurable number of elements

Endpoints:

← **input (T)** - no description given -
→ **output (T)** - no description given -

Configuration items:

numberOfElements (32 bit signed integer) number of elements to skip

B.14 Conveyor.Dictionary

A dictionary data type for Conveyor

Data types

Conveyor.Dictionary.Dictionary<T,U> A simple dictionary for storing key-value pairs

Node types

Conveyor.Dictionary.AddKeyValue<T,U>

adds the given value to the dictionary using the given key or overwrites an existing value, and passes the dictionary to the output

Endpoints:

← **dictIn (Conveyor.Dictionary.Dictionary<T,U>)** dictionary to add key and value to
← **key (T)** key to use for adding or overwriting
← **value (U)** value to add to the dictionary
→ **dictOut (Conveyor.Dictionary.Dictionary<T,U>)** dictionary after adding/overwriting

Conveyor.Dictionary.ContainsKey<T,U>

returns true if the dictionary contains the given key

Endpoints:

← **firstArgument (Conveyor.Dictionary.Dictionary<T,U>)** the first argument of the operation
← **secondArgument (T)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.Dictionary.CountObjects<T>

Counts the occurence of objects send into this node and creates a dictionary using the objects as key and their number of occurences as value

Endpoints:

← **obj (T)** objects to count
→ **counts (Conveyor.Dictionary.Dictionary<T,Conveyor.Math.ULong>)** dictionary containing objects and their number of occurences

Conveyor.Dictionary.CreateDictionary<T,U>

creates a dictionary from the key value pairs read on the input endpoints

Endpoints:

← **key (T)** key of a pair
← **value (U)** value of a pair
→ **dictionary (Conveyor.Dictionary.Dictionary<T,U>)** the dictionary after reading all available key value pair

Conveyor.Dictionary.CreateEmptyDictionary<T,U>

creates empty dictionary instances

Endpoints:

→ **output (Conveyor.Dictionary.Dictionary<T,U>)** - no description given -

Conveyor.Dictionary.DeleteKey<T,U>

removes the key and its associated value from the dictionary

Endpoints:

← **firstArgument (Conveyor.Dictionary.Dictionary<T,U>)** the first argument of the operation
← **secondArgument (T)** the second argument of the operation
→ **result (Conveyor.Dictionary.Dictionary<T,U>)** the result of the operation

Conveyor.Dictionary.GetKeys<T,U>

Returns a list of all keys in a dictionary

Endpoints:

← **input (Conveyor.Dictionary.Dictionary<T,U>)** - no description given -
→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.Dictionary.GetValue<T,U>

returns the value associated with the given key, or throws an exception if the key does not exists

Endpoints:

← **firstArgument (Conveyor.Dictionary.Dictionary<T,U>)** the first argument of the operation
← **secondArgument (T)** the second argument of the operation
→ **result (U)** the result of the operation

Conveyor.Dictionary.GetValues<T,U>

Returns a list of all values in a dictionary

Endpoints:

← **input (Conveyor.Dictionary.Dictionary<T,U>)** - no description given -
→ **output (Conveyor.List.List<U>)** - no description given -

Conveyor.Dictionary.MatchElements<T,U,V>

Creates pairs of matching elements by synchronising the keys of elements. First input is consumed and stored until match to second input is found.

Endpoints:

← **firstElement (U)** first element
← **firstKey (T)** key of first element
← **secondElement (V)** second element
← **secondKey (T)** key of second element
→ **match (Conveyor.Pair.Pair<U,V>)** pair of matching elements

B.15 Conveyor.EMBL

EMBL/GenBank file access for Conveyor

Data types

Conveyor.EMBL.Feature - no description given -

Conveyor.EMBL.TranslatedCDS the translation of a CDS feature

Conveyor.EMBL.Features.CDS - no description given -

Conveyor.EMBL.Features.Gene - no description given -

Conveyor.EMBL.Features.rRNA - no description given -

Conveyor.EMBL.Features.tRNA - no description given -

Conveyor.EMBL.CreatedMasterSequence - no description given -

Conveyor.EMBL.MasterSequence - no description given -

Conveyor.EMBL.ParsedMasterSequence - no description given -

Node types

Conveyor.EMBL.CreateMasterSequence

converts a DNA sequence into a EMBL/GenBank master sequence

Endpoints:

← **input (Conveyor.BioinformaticsTypes.DNASequence)** sequence for creatinga master sequence of
→ **output (Conveyor.EMBL.CreatedMasterSequence)** generated master sequence

Conveyor.EMBL.GetFeatures

extracts all features of the given type from the master sequence

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the master sequence to extract features from
→ **features (Conveyor.List.List<Conveyor.EMBL.Feature>)** the features of the master sequence

Configuration items:

featureType (- unknown type -) type of features to extract, all features if undefined

Conveyor.EMBL.GetMasterSequence

returns the master sequence a feature is part of

Endpoints:

← **input (Conveyor.EMBL.Feature)** - no description given -
→ **output (Conveyor.EMBL.MasterSequence)** - no description given -

Conveyor.EMBL.ReadMasterSequences

reads entries from EMBL or GenBank formatted flat files

Endpoints:

→ **entry (Conveyor.EMBL.ParsedMasterSequence)** an entry read from the input file

Configuration items:

fileType (- unknown type -) the input file type
inputFile (input file) the file to read from

Conveyor.EMBL.WriteMasterSequences

creates an EMBL or GenBank formatted file from the master sequences

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the sequences to write

Configuration items:

fileType (- unknown type -) format of the output file

Conveyor.EMBL.CreateCDSFeature

returns features of class CDS

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the sequence to create the feature on
← **start (Conveyor.Math.ULong)** 0-based start position of the feature
← **stop (Conveyor.Math.ULong)** 0-based stop position of the feature
→ **feature (Conveyor.EMBL.Features.CDS)** the created feature

Conveyor.EMBL.GetCDSFeatures

returns features of class CDS

Endpoints:

← **input (Conveyor.EMBL.MasterSequence)** master sequence to get features from
→ **output (Conveyor.List.List<Conveyor.EMBL.Features.CDS>)** list of features

Conveyor.EMBL.CreateGeneFeature

returns features of class Gene

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the sequence to create the feature on
← **start (Conveyor.Math.ULong)** 0-based start position of the feature
← **stop (Conveyor.Math.ULong)** 0-based stop position of the feature
→ **feature (Conveyor.EMBL.Features.Gene)** the created feature

Conveyor.EMBL.GetGeneFeatures

returns features of class Gene

Endpoints:

← **input (Conveyor.EMBL.MasterSequence)** master sequence to get features from
→ **output (Conveyor.List.List<Conveyor.EMBL.Features.Gene>)** list of features

Conveyor.EMBL.CreaterRNAFeature

returns features of class rRNA

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the sequence to create the feature on
← **start (Conveyor.Math.ULong)** 0-based start position of the feature
← **stop (Conveyor.Math.ULong)** 0-based stop position of the feature
→ **feature (Conveyor.EMBL.Features.rRNA)** the created feature

Conveyor.EMBL.GetrRNAFeatures

returns features of class rRNA

Endpoints:

← **input (Conveyor.EMBL.MasterSequence)** master sequence to get features from
→ **output (Conveyor.List.List<Conveyor.EMBL.Features.rRNA>)** list of features

Conveyor.EMBL.CreatetRNAFeature

returns features of class tRNA

Endpoints:

← **sequence (Conveyor.EMBL.MasterSequence)** the sequence to create the feature on
← **start (Conveyor.Math.ULong)** 0-based start position of the feature
← **stop (Conveyor.Math.ULong)** 0-based stop position of the feature
→ **feature (Conveyor.EMBL.Features.tRNA)** the created feature

Conveyor.EMBL.GettRNAFeatures

returns features of class tRNA

Endpoints:

← **input (Conveyor.EMBL.MasterSequence)** master sequence to get features from
→ **output (Conveyor.List.List<Conveyor.EMBL.Features.tRNA>)** list of features

B.16 Conveyor.Geneprediction

Gene prediction and related node and data classes for conveyor

Data types

Conveyor.Geneprediction.PredictedGene<T> - no description given -

Conveyor.Geneprediction.SimplePredictedGene<T> - no description given -

Conveyor.Geneprediction.SimplePredictedrRNA<T> - no description given -

Node types

Conveyor.Geneprediction.Critica<T>

executes the critica gene prediction software and returns a list of predicted genes

Endpoints:

← **contig (T)** input sequence to predict genes on
→ **genes (Conveyor.List.List<Conveyor.Geneprediction.SimplePredictedGene‘1>)** list of genes predicted by Critica

Configuration items:

geneticCode (32 bit signed integer) genetic code to use for gene prediction, default to 11 (bacterial code)

Conveyor.Geneprediction.Glimmer3<T>

executes the CeBiTec Glimmer 3 wrapper to predict genes on the given contig

Endpoints:

← **contig (T)** input sequence to predict genes on
→ **genes (Conveyor.List.List<Conveyor.Geneprediction.SimplePredictedGene‘1>)** list of genes predicted by Glimmer3

Configuration items:

scoreThreshold (IEEE double value) score threshold value to discard gene predictions, defaults to 4.0

Conveyor.Geneprediction.RNAMMER<T>

executes the rnammer to predict ribosomal RNA genes

Endpoints:

← **contig (T)** input sequence to predict genes on
→ **rrnas (Conveyor.List.List<Conveyor.Geneprediction.SimplePredictedrRNA‘1>)** list of rRNAs predicted by rnammer

Configuration items:

domain (- unknown type -) domain model to use

Conveyor.Geneprediction.Prodigal<T>

executes Prodigal to predict genes on the given contig

Endpoints:

← **contig (T)** input sequence to predict genes on
→ **genes (Conveyor.List.List<Conveyor.Geneprediction.SimplePredictedGene‘1>)** list of genes predicted by Prodigal

PredictedGene<T>

Returns the sequence this gene was predicted for

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (T)** - no description given -

PredictedGene<T>

Returns the zero-based start position of this predicted gene

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

PredictedGene<T>

Returns the zero-based stop position of this predicted gene (first base outside the gene)

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

PredictedGene<T>

Returns the length of the predicted gene (stop - start)

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

PredictedGene<T>

Returns the plain DNA sequence string of the predicted gene

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

PredictedGene<T>

Translates the predicted gene using the default genetic code and returns the plain AA string

Endpoints:

← **input (Conveyor.Geneprediction.PredictedGene<T>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

B.17 Conveyor.HMMER3

Conveyor plugin for HMMER3

Data types

Conveyor.HMMER3.HMMER3DomainHit A hit vs. a HMM domain

Conveyor.HMMER3.HMMER3Result<T,U,V,W> The result of a hmmscan run with the given query sequence

Conveyor.HMMER3.HMMER3ScanDomainHit<T,U,V,W> A hit vs. a HMM domain

Conveyor.HMMER3.HMMv3 a HMM used with the HMMER3 application suite

Conveyor.HMMER3.HMMv3DB A collection of HMMs

Conveyor.HMMER3.IHMM<T,U> Interface for classes representing HMMs to be used with HMMER3

Conveyor.HMMER3.IHMMDB<T,U,V> - no description given -

Node types

Conveyor.HMMER3.GetHMMFromHMMERHit<T,U,V,W>

Gets the HMM from a HMMER3 hit

Endpoints:

← **input (Conveyor.HMMER3.HMMER3ScanDomainHit<T,U,V,W>)** - no description given -
→ **output (W)** - no description given -

Conveyor.HMMER3.GetQueryFromHMMERHit<T,U,V,W>

Gets the original query sequence from a HMMER3 hit

Endpoints:

← **input (Conveyor.HMMER3.HMMER3ScanDomainHit<T,U,V,W>)** - no description given -
→ **output (T)** - no description given -

Conveyor.HMMER3.GetQueryFromHMMERResult<T,U,V,W>

Gets the original query sequence from a HMMER3 result

Endpoints:

← **input (Conveyor.HMMER3.HMMER3Result<T,U,V,W>)** - no description given -
→ **output (T)** - no description given -

Conveyor.HMMER3.HMMDatabase

Provides access to a HMM database on the system

Endpoints:

→ **db (Conveyor.HMMER3.HMMv3DB)** the selected database

Configuration items:

database (- unknown type -) HMM databases available

Conveyor.HMMER3.HMMScanv3<T,U,V,W>

Searches for (protein) domains absed on a HMM database using HMMER3

Endpoints:

← **database (Conveyor.HMMER3.IHMMDB<U,V,W>)** HMM database for search domains in
← **seq (T)** Sequence for search domains in
→ **result (Conveyor.HMMER3.HMMER3Result<T,U,V,W>)** hmmscan result

Configuration items:

chunkSize (32 bit signed integer) number of sequences to process in a single hmmscan instance; use 1 for large databases, and larger values for small databases

Conveyor.HMMER3.HMMSeedAlignment<T,U,V>

Converts a HMM to an alignment via accession

Endpoints:

← **hmm (V)** - no description given -
→ **output (U)** - no description given -

IHMM<T,U>

no description available

Endpoints:

← **input (Conveyor.HMMER3.IHMM<T,U>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

IHMM<T,U>

no description available

Endpoints:

← **input (Conveyor.HMMER3.IHMM<T,U>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

IHMM<T,U>

no description available

Endpoints:

← **input (Conveyor.HMMER3.IHMM<T,U>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

B.18 Conveyor.List

Generic list for Conveyor

Data types

Conveyor.List.List<T> - no description given -

Node types

Conveyor.List.AppendElement<T>

adds an element to the end of a list

Endpoints:

← **firstArgument (Conveyor.List.List<T>)** the first argument of the operation
← **secondArgument (T)** the second argument of the operation
→ **result (Conveyor.List.List<T>)** the result of the operation

Conveyor.List.CastList<T,U>

casts a list of elements to a list of derived elements

Endpoints:

← **input (Conveyor.List.List<T>)** - no description given -
→ **output (Conveyor.List.List<U>)** - no description given -

Conveyor.List.CollectN<T>

creates lists with a defined number of elements

Endpoints:

← **input (T)** elements to bundle to lists
→ **list (Conveyor.List.List<T>)** list with the requested number of elements

Configuration items:

numberOfElements (32 bit signed integer) number of elements to put into a list

Conveyor.List.Collector<T>

collects data objects and returns them as list

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.List.CombineElement<T,U>

creates a list of pairs by using all elements of a given list as the first element and the given element as the second one

Endpoints:

← **element (U)** the element to combine the list with
← **inputList (Conveyor.List.List<T>)** the list to process
→ **outputList (Conveyor.List.List<Conveyor.Pair.Pair'2>)** the resulting list of pairs

Conveyor.List.Compare<T>

compares two lists by pairwise checking their elements from equality

Endpoints:

← **firstArgument (Conveyor.List.List<T>)** the first argument of the operation
← **secondArgument (Conveyor.List.List<T>)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.List.FilterList<T>

removes all elements from a list not matching a criteria

Endpoints:

← **checkIn (Conveyor.Logic.Boolean)** the result of a check
← **listIn (Conveyor.List.List<T>)** the list to process
→ **checkOut (T)** a single element of the list to check
→ **listOut (Conveyor.List.List<T>)** the list with non matching elements removed

Conveyor.List.GetElement<T>

returns the element from the list at the given position, or terminates processing if the index is out of range

Endpoints:

← **firstArgument (Conveyor.List.List<T>)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (T)** the result of the operation

Conveyor.List.GetEmptyList<T>

generates empty lists

Endpoints:

→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.List.HasElements<T>

returns true if the given list contains elements

Endpoints:

← **input (Conveyor.List.List<T>)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.List.Iterator<T>

reads list from the input and returns their elements one by one

Endpoints:

← **input (Conveyor.List.List<T>)** list to iterate over
→ **output (T)** elements from the list, one by one

Conveyor.List.ListCount<T>

gets the number of elements in a list

Endpoints:

← **input (Conveyor.List.List<T>)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

Conveyor.List.Map<T,U>

converts a list of elements to another list of elements

Endpoints:

← **elementIn (U)** the element after processing
← **listIn (Conveyor.List.List<T>)** a list to process
→ **elementOut (T)** an element from the current list to be processed
→ **listOut (Conveyor.List.List<U>)** the complete processed list

Conveyor.List.NewList<T>

creates a new list using the input elements as first and single list element

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.List.PrependElement<T>

adds an element in front of a list

Endpoints:

← **firstArgument (Conveyor.List.List<T>)** the first argument of the operation
← **secondArgument (T)** the second argument of the operation
→ **result (Conveyor.List.List<T>)** the result of the operation

Conveyor.List.Select<T>

selects an element from list by comparing the elements

Endpoints:

← **listIn (Conveyor.List.List<T>)** list to process
← **result (Conveyor.Logic.Boolean)** result of comparison, true selects first element, false selects second element
→ **firstOut (T)** first element of comparison
→ **secondOut (T)** second element of comparison
→ **selected (T)** selected element

Conveyor.List.Sort<T>

sorts the elements of a list

Endpoints:

← **compareResult (Conveyor.Comparison.ComparisonResult)** result of comparison, expressing the order of the first and the second element in the pair
← **unsorted (Conveyor.List.List<T>)** list to sort
→ **firstCompare (T)** first element to compare
→ **secondCompare (T)** second element to compare
→ **sorted (Conveyor.List.List<T>)** sorted list

Conveyor.List.SROL<T,U>

structural recursion on lists

Endpoints:

← **input (Conveyor.List.List<T>)** list to be processed
← **resultIn (U)** processing result that is either passed as input to the next iteration or passed to the output link as end result
← **startValue (U)** start element for each list iteration
→ **firstOut (T)** loop value taken from input list
→ **output (U)** result after processing all inputs
→ **secondOut (U)** feedback output value, either taken from start value link or from result of the the former run

B.19 Conveyor.Logic

Definition of a logic (boolean) data type and nodes operating on it.

Data types

Conveyor.Logic.Boolean - no description given -

Node types

Conveyor.Logic.And

logical and of two boolean values

Endpoints:

← **firstArgument (Conveyor.Logic.Boolean)** the first argument of the operation
← **secondArgument (Conveyor.Logic.Boolean)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.Logic.BoolGenerator

generator that produces boolean truee values

Endpoints:

→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

value (boolean) boolean value to generate

Conveyor.Logic.Branch<T>

Branches the control flow depending on an external condition

Endpoints:

← **conditionIn (Conveyor.Logic.Boolean)** result of the condition loop
← **input (T)** data elements to branch
→ **conditionOut (T)** start of the condition loop that defines which branch a data element has to take
→ **falseOut (T)** output of elements with a condition result of false
→ **trueOut (T)** output of elements with a condition result of true

Conveyor.Logic.IfThenElse<T,U>

commonly known two branched flow control node

Endpoints:

← **conditionResult (Conveyor.Logic.Boolean)** result of condition check
← **elseIn (U)** result of else branch processing
← **input (T)** data to pass to then or else branch
← **thenIn (U)** result of then branch processing
→ **conditionOut (T)** data passed to condition check
→ **elseOut (T)** data passed to else branch
→ **output (U)** returned data of then or else branch
→ **thenOut (T)** data passed to then branch

Conveyor.Logic.Not

logical inverse of the input value

Endpoints:

← **input (Conveyor.Logic.Boolean)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Logic.Or

logical or of two boolean values

Endpoints:

← **firstArgument (Conveyor.Logic.Boolean)** the first argument of the operation
← **secondArgument (Conveyor.Logic.Boolean)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.Logic.PassCheck<T>

passed stream elements depending on boolean input

Endpoints:

← **input (T)** - no description given -
← **pass (Conveyor.Logic.Boolean)** - no description given -
→ **output (T)** - no description given -

Conveyor.Logic.PassFilter<T>

forwards objects that passes a check loop

Endpoints:

← **check_in (Conveyor.Logic.Boolean)** - no description given -
← **input (T)** - no description given -
→ **check_out (T)** - no description given -
→ **output (T)** - no description given -

Conveyor.Logic.Xor

logical xor of two boolean values

Endpoints:

← **firstArgument (Conveyor.Logic.Boolean)** the first argument of the operation
← **secondArgument (Conveyor.Logic.Boolean)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

B.20 Conveyor.Math

Definition of numeric data types within Conveyor

Data types

Conveyor.Math.Byte unsigned 8 bit integer

Conveyor.Math.Double - no description given -

Conveyor.Math.Integer - no description given -

Conveyor.Math.Long signed 64 bit integer

Conveyor.Math.SByte signed 8 bit integer

Conveyor.Math.ULong unsigned 64 bit integer

Node types

Conveyor.Math.ByteOps.Add

returns the sum of both byte values

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Between

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.Byte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (- unknown type -) lower bound
upperBound (- unknown type -) upper bound

Conveyor.Math.LongOps.ByteGenerator

generator that produces long value

Endpoints:

→ **output (Conveyor.Math.Byte)** - no description given -

Configuration items:

value (- unknown type -) integer value to generate

Conveyor.Math.ByteOps.Compare

compares two byte values

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.ByteOps.Div

returns the integral part of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.Byte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

compareValue (- unknown type -) value to compare to

Conveyor.Math.ByteOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.Byte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (- unknown type -) value to compare to

Conveyor.Math.ByteOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.Byte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (- unknown type -) value to compare to

Conveyor.Math.ByteOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Mod

returns the integral remainder of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Mul

returns the product of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.ByteOps.Sub

returns the difference of the two byte values

Endpoints:

← **firstArgument (Conveyor.Math.Byte)** the first argument of the operation
← **secondArgument (Conveyor.Math.Byte)** the second argument of the operation
→ **result (Conveyor.Math.Byte)** the result of the operation

Conveyor.Math.DoubleOps.Add

returns the sum of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.DoubleOps.Between

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (IEEE double value) lower bound
upperBound (IEEE double value) upper bound

Conveyor.Math.DoubleOps.Compare

compares two double values

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.DoubleOps.Div

returns the quotient of the both operands

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.DoubleOps.Generator

generator that produces floating point values

Endpoints:

→ **output (Conveyor.Math.Double)** - no description given -

Configuration items:

value (IEEE double value) double value to generate

Conveyor.Math.DoubleOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (IEEE double value) value to compare to

Conveyor.Math.DoubleOps.IsInf

returns true if the double value is positive or negative infinity

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Math.DoubleOps.IsNaN

returns true if the value is NaN (not a number)

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Math.DoubleOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (IEEE double value) value to compare to

Conveyor.Math.DoubleOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.Double)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (IEEE double value) value to compare to

Conveyor.Math.DoubleOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.DoubleOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.DoubleOps.Mul

returns the product of the two factors

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.DoubleOps.Random

generates random dboule numbers within a given range

Endpoints:

→ **output (Conveyor.Math.Double)** - no description given -

Configuration items:

lowerBound (IEEE double value) lower bound of the range
upperBound (IEEE double value) upper bound of the range

Conveyor.Math.DoubleOps.Sub

returns the difference of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Double)** the first argument of the operation
← **secondArgument (Conveyor.Math.Double)** the second argument of the operation
→ **result (Conveyor.Math.Double)** the result of the operation

Conveyor.Math.IntegerOps.Absolute

returns the absolute value of the integer

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

Conveyor.Math.IntegerOps.Add

returns the sum of both integer values

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.BetweenInt

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (32 bit signed integer) lower bound
upperBound (32 bit signed integer) upper bound

Conveyor.Math.IntegerOps.Compare

compares two integer values

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.IntegerOps.Div

returns the integral part of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

compareValue (32 bit signed integer) value to compare to

Conveyor.Math.IntegerOps.IntegerGenerator

generator that produces integer value

Endpoints:

→ **output (Conveyor.Math.Integer)** - no description given -

Configuration items:

value (32 bit signed integer) integer value to generate

Conveyor.Math.IntegerOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (32 bit signed integer) value to compare to

Conveyor.Math.IntegerOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (32 bit signed integer) value to compare to

Conveyor.Math.IntegerOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.Mod

returns the integral remainder of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.Mul

returns the product of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.IntegerOps.Neg

negates the given value

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Math.Integer)** - no description given -

Conveyor.Math.IntegerOps.Random

generates random integer numbers within a given range

Endpoints:

→ **output (Conveyor.Math.Integer)** - no description given -

Configuration items:

lowerBound (32 bit signed integer) lower bound of the range
upperBound (32 bit signed integer) upper bound of the range

Conveyor.Math.IntegerOps.Sub

returns the difference of the two integer values

Endpoints:

← **firstArgument (Conveyor.Math.Integer)** the first argument of the operation
← **secondArgument (Conveyor.Math.Integer)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.Math.LongOps.Absolute

returns the absolute value of the integer

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Math.Long)** - no description given -

Conveyor.Math.LongOps.Add

returns the sum of both integer values

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.BetweenInt

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (64 bit signed integer) lower bound
upperBound (64 bit signed integer) upper bound

Conveyor.Math.LongOps.Compare

compares two long values

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.LongOps.Div

returns the integral part of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

compareValue (64 bit signed integer) value to compare to

Conveyor.Math.LongOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (64 bit signed integer) value to compare to

Conveyor.Math.LongOps.LongGenerator

generator that produces long value

Endpoints:

→ **output (Conveyor.Math.Long)** - no description given -

Configuration items:

value (64 bit signed integer) integer value to generate

Conveyor.Math.LongOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (64 bit signed integer) value to compare to

Conveyor.Math.LongOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.Mod

returns the integral remainder of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.Mul

returns the product of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.LongOps.Neg

negates the given value

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Math.Long)** - no description given -

Conveyor.Math.LongOps.Sub

returns the difference of the two long values

Endpoints:

← **firstArgument (Conveyor.Math.Long)** the first argument of the operation
← **secondArgument (Conveyor.Math.Long)** the second argument of the operation
→ **result (Conveyor.Math.Long)** the result of the operation

Conveyor.Math.SByteOps.Absolute

returns the absolute value of the signed byte

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Math.SByte)** - no description given -

Conveyor.Math.SByteOps.Add

returns the sum of both integer values

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Between

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (8 bit signed integer) lower bound
upperBound (8 bit signed integer) upper bound

Conveyor.Math.SByteOps.Compare

compares two signed byte values

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.SByteOps.Div

returns the integral part of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

compareValue (8 bit signed integer) value to compare to

Conveyor.Math.SByteOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (8 bit signed integer) value to compare to

Conveyor.Math.SByteOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (8 bit signed integer) value to compare to

Conveyor.Math.SByteOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Mod

returns the integral remainder of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Mul

returns the product of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.SByteOps.Neg

negates the given value

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Math.SByte)** - no description given -

Conveyor.Math.SByteOps.SByteGenerator

generator that produces signed byte value

Endpoints:

→ **output (Conveyor.Math.SByte)** - no description given -

Configuration items:

value (8 bit signed integer) signed byte value to generate

Conveyor.Math.SByteOps.Sub

returns the difference of the two long values

Endpoints:

← **firstArgument (Conveyor.Math.SByte)** the first argument of the operation
← **secondArgument (Conveyor.Math.SByte)** the second argument of the operation
→ **result (Conveyor.Math.SByte)** the result of the operation

Conveyor.Math.ULongOps.Add

returns the sum of both integer values

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.Between

checks whether the value in between the given bounds

Endpoints:

← **input (Conveyor.Math.ULong)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

lowerBound (64 bit unsigned integer) lower bound
upperBound (64 bit unsigned integer) upper bound

Conveyor.Math.ULongOps.ByteToULong

converts an unsigned byte value to an unsigned 64 bit value

Endpoints:

← **input (Conveyor.Math.Byte)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Math.ULongOps.Compare

compares two long values

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Math.ULongOps.Div

returns the integral part of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.Equals

checks whether the input is the same as the configured value

Endpoints:

← **input (Conveyor.Math.ULong)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

compareValue (64 bit unsigned integer) value to compare to

Conveyor.Math.ULongOps.IntToULong

converts a signed 32 bit value to an unsigned 64 bit value

Endpoints:

← **input (Conveyor.Math.Integer)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Math.ULongOps.Larger

checks whether the input is larger than the configured value

Endpoints:

← **input (Conveyor.Math.ULong)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (64 bit unsigned integer) value to compare to

Conveyor.Math.ULongOps.LongToULong

converts a signed 64 bit value to an unsigned 64 bit value

Endpoints:

← **input (Conveyor.Math.Long)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Math.ULongOps.Lower

checks whether the input is lower than the configured value

Endpoints:

← **input (Conveyor.Math.ULong)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

threshold (64 bit unsigned integer) value to compare to

Conveyor.Math.ULongOps.Max

returns the larger of the two values

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.Min

returns the smaller of the two values

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.Mod

returns the integral remainder of the quotient of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.Mul

returns the product of the two operands

Endpoints:

← **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.SByteToULong

converts a signed byte to an unsigned 64 bit value

Endpoints:

← **input (Conveyor.Math.SByte)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Math.ULongOps.Sub

returns the difference of the two long values

Endpoints:

$\leftarrow$ **firstArgument (Conveyor.Math.ULong)** the first argument of the operation
$\leftarrow$ **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
$\rightarrow$ **result (Conveyor.Math.ULong)** the result of the operation

Conveyor.Math.ULongOps.ULongGenerator

generator that produces long value

Endpoints:

$\rightarrow$ **output (Conveyor.Math.ULong)** - no description given -

Configuration items:

value (64 bit unsigned integer) integer value to generate

B.21 Conveyor.Pair

Data types

Conveyor.Pair.Pair<T,U> - no description given -

Node types

Conveyor.Pair.CreatePair<T,U>

Creates a Pair from two input data elements

Endpoints:

$\leftarrow$ **firstArgument (T)** the first argument of the operation
$\leftarrow$ **secondArgument (U)** the second argument of the operation
$\rightarrow$ **result (Conveyor.Pair.Pair<T,U>)** the result of the operation

Conveyor.Pair.FirstElement<T,U>

Extracts the first element of a pair

Endpoints:

$\leftarrow$ **input (Conveyor.Pair.Pair<T,U>)** - no description given -
$\rightarrow$ **output (T)** - no description given -

Conveyor.Pair.PairSplit<T,U>

splits the pair in both elements

Endpoints:

← **pair (Conveyor.Pair.Pair<T,U>)** pair to process
→ **first (T)** first element of pair
→ **second (U)** second element of pair

Conveyor.Pair.SecondElement<T,U>

Extracts the second element of a pair

Endpoints:

← **input (Conveyor.Pair.Pair<T,U>)** - no description given -
→ **output (U)** - no description given -

B.22 Conveyor.Phylogeny

Data types for phylogenic trees

Data types

Conveyor.Phylogeny.DistanceMatrix - no description given -

Conveyor.Phylogeny.EmptyDistanceMatrix - no description given -

Conveyor.Phylogeny.PhylogenyNode - no description given -

Node types

Conveyor.Phylogeny.ClearMarkedSpecies

clones the input tree and removes all group markers from the copy before returning it

Endpoints:

← **input (Conveyor.Phylogeny.PhylogenyNode)** - no description given -
→ **output (Conveyor.Phylogeny.PhylogenyNode)** - no description given -

Conveyor.Phylogeny.ContainsLabeledNode

Scans the given tree for nodes with the given label. Returns true if such a node is found.

Endpoints:

← **firstArgument (Conveyor.Phylogeny.PhylogenyNode)** the first argument of the operation
← **secondArgument (Conveyor.Text.String)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.Phylogeny.GetLabeledNode

Scans the given tree for nodes with the given label. Returns the first node found or throws an exception if no such node is found

Endpoints:

← **firstArgument (Conveyor.Phylogeny.PhylogenyNode)** the first argument of the operation
← **secondArgument (Conveyor.Text.String)** the second argument of the operation
→ **result (Conveyor.Phylogeny.PhylogenyNode)** the result of the operation

Conveyor.Phylogeny.IsEmptyMatrix

returns true if the given distance matrix is empty

Endpoints:

← **input (Conveyor.Phylogeny.DistanceMatrix)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Phylogeny.MarkedSpecies

returns a list of the names of marked species in a tree

Endpoints:

← **input (Conveyor.Phylogeny.PhylogenyNode)** - no description given -
→ **output (Conveyor.List.List<Conveyor.Text.String>)** - no description given -

Configuration items:

marker (32 bit signed integer) the marker to look for

Conveyor.Phylogeny.MarkSpeciesInTree

sets a marker for a species identified by its name, without cloning the tree

Endpoints:

← **input (Conveyor.Phylogeny.PhylogenyNode)** - no description given -
→ **output (Conveyor.Phylogeny.PhylogenyNode)** - no description given -

Configuration items:

marker (32 bit signed integer) marker to use, integer value > 0
species (literal string) species to mark

Conveyor.Phylogeny.MarkSpeciesInTreeByName

sets a marker for a species

Endpoints:

← **inTree (Conveyor.Phylogeny.PhylogenyNode)** tree before marking the species
← **species (Conveyor.Text.String)** name of species to mark in tree
→ **outTree (Conveyor.Phylogeny.PhylogenyNode)** clone of the tree with marked species

Configuration items:

marker (32 bit signed integer) marker to use, integer value > 0

Conveyor.Phylogeny.NewickToTree

converts the input string in newick format to a phylogenetic tree

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Phylogeny.PhylogenyNode)** - no description given -

Conveyor.Phylogeny.TreeToNewick

Creates a newick formatted string from the given tree

Endpoints:

← **input (Conveyor.Phylogeny.PhylogenyNode)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

B.23 Conveyor.QualitySequence

Interfaces and processing nodes for sequences with attached quality information

Data types

Conveyor.QualitySequence.ArtificalDNAQSequence<T> A quality sequence created from a plain dna sequence

Conveyor.QualitySequence.ConcatenatedQSequence<T,U> a sequence with quality values build from two other quality sequences

Conveyor.QualitySequence.InsertedQSequence<T,U> a quality sequence created by inserting one q-sequence into another

Conveyor.QualitySequence.IQualitySequence A set of position based quality information for sequences, e.g. quality values generated by a sequencer or a base calling application

Conveyor.QualitySequence.ReverseComplementedQSequence<T> Reverse complement of a DNA quality sequence

Conveyor.QualitySequence.ReversedQSequence<T> the reversed sequence of a given quality sequence

Conveyor.QualitySequence.SimpleDNAQSequence - no description given -

Conveyor.QualitySequence.SplicedQSequence<T> part of a quality sequence

Node types

Conveyor.QualitySequence.AverageQuality<T>

calculates the average sequence quality

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.Math.Double)** - no description given -

Conveyor.QualitySequence.ClipQSequence<T>

Strips low quality stretches from the end of a quality sequence

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.QualitySequence.SplicedQSequence<T>)** - no description given -

Configuration items:

minLowQualityStretch (64 bit unsigned integer) min length of low quality stretches to clip
qualityThreshold (8 bit signed integer) quality value to filter (<=)

Conveyor.QualitySequence.ConcatenateQSequences<T,U>

Concatenates two sequences with quality values

Endpoints:

← **first (T)** first sequence
← **name (Conveyor.Text.String)** name of the concatenated sequence
← **second (U)** second sequence
→ **concatenated (Conveyor.QualitySequence.ConcatenatedQSequence<T,U>)** result of concatenation

Conveyor.QualitySequence.CreateFromDNASequence<T>

creates a quality sequence from the given DNA sequence, using the configured value as quality value for all bases

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.QualitySequence.ArtificalDNAQSequence<T>)** - no description given -

Configuration items:

qualityValue (8 bit signed integer) Quality value to use for all bases

Conveyor.QualitySequence.GetQualityValue

Returns the quality value at the position given as second input to this node

Endpoints:

← **firstArgument (Conveyor.QualitySequence.IQualitySequence)** the first argument of the operation
← **secondArgument (Conveyor.Math.ULong)** the second argument of the operation
→ **result (Conveyor.Math.Integer)** the result of the operation

Conveyor.QualitySequence.InsertQSequence<T,U>

creates a quality sequence by inserting the insert sequence into the template sequence at the given position

Endpoints:

← **insert (U)** sequence to be inserted
← **position (Conveyor.Math.ULong)** position to insert sequence at
← **sequence (T)** sequence to insert into
→ **result (Conveyor.QualitySequence.InsertedQSequence<T,U>)** result of the insert

Conveyor.QualitySequence.ReverseComplementQSequence<T>

Creates a new DNA quality sequence using the reversed complemented input sequence

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.QualitySequence.ReverseComplementedQSequence<T>)** - no description given -

Conveyor.QualitySequence.ReverseQSequence<T>

creates a reverse quality sequence from the input

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.QualitySequence.ReversedQSequence<T>)** - no description given -

Conveyor.QualitySequence.SpliceQSequence<T>

Returns a DNA quality sequence with a given parts spliced out

Endpoints:

← **from (Conveyor.Math.ULong)** start of spliced range
← **sequence (T)** sequence to process
← **to (Conveyor.Math.ULong)** end of spliced range
→ **splicedSequence (Conveyor.QualitySequence.SplicedQSequence<T>)** sequence after splicing

B.24 Conveyor.SequencePair

Interfaces and processing types for pairs of sequences, e.g. paired end reads from a sequencer

Data types

Conveyor.SequencePair.Orientation orientation of a sequence pair's sequences

Conveyor.SequencePair.SequencePair<T,U> A pair of sequences, e.g. a paired end read from a next generation sequencer

Conveyor.SequencePair.SimpleSequencePair<T,U> simple implementation of sequence pairs

Node types

Conveyor.SequencePair.CreateSequencePair<T,U>

creates a sequence pair from the given sequence, using the given orientation

Endpoints:

← **first (T)** first sequence of pair
← **orientation (Conveyor.SequencePair.Orientation)** orientation of sequences
← **second (U)** second sequence of pair
→ **pair (Conveyor.SequencePair.SimpleSequencePair<T,U>)** composed sequence pair

Conveyor.SequencePair.IsSameOrientation

Returns true if both orientations are known and are the same.

Endpoints:

← **firstArgument (Conveyor.SequencePair.Orientation)** the first argument of the operation
← **secondArgument (Conveyor.SequencePair.Orientation)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

Conveyor.SequencePair.IsUnknownOrientation

Checks whether the given orientation is an unknown orientation

Endpoints:

← **input (Conveyor.SequencePair.Orientation)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.SequencePair.OrientationGenerator

Generates Orientation values for CreateSequencePair and similar node types

Endpoints:

→ **output (Conveyor.SequencePair.Orientation)** - no description given -

Configuration items:

orientation (- unknown type -) Orientation to use

SequencePair<T,U>

Returns the first sequence of the pair

Endpoints:

← **input (Conveyor.SequencePair.SequencePair<T,U>)** - no description given -
→ **output (T)** - no description given -

SequencePair<T,U>

Returns the second sequence of the pair

Endpoints:

← **input (Conveyor.SequencePair.SequencePair<T,U>)** - no description given -
→ **output (U)** - no description given -

SequencePair<T,U>

Returns the orientation of the sequences within the pair

Endpoints:

← **input (Conveyor.SequencePair.SequencePair<T,U>)** - no description given -
→ **output (Conveyor.SequencePair.Orientation)** - no description given -

SequencePair<T,U>

Returns the distance between the sequence (including both sequences' length), or -1 if the distance is unknown

Endpoints:

← **input (Conveyor.SequencePair.SequencePair<T,U>)** - no description given -
→ **output (Conveyor.Math.Long)** - no description given -

B.25 Conveyor.SolexaProcessing

CeBiTec specific tools for processing Solexa sequencer data

Data types

Conveyor.SolexaProcessing.AlignedDNAQSequence<T> A quality sequence being part of an alignment of two quality sequences

Conveyor.SolexaProcessing.SequenceOverlap<T,U> - no description given -

Node types

Conveyor.SolexaProcessing.CountDistinctElements<T>

counts the occurence of distinct elements and returns a statistic

Endpoints:

← **input (T)** elements to count

Conveyor.SolexaProcessing.CountElements

reports the number of elements send to this node

Endpoints:

← **input (Conveyor.Core.Data)** elements to count

Conveyor.SolexaProcessing.CreateConsensus<T,U>

creates the consensus sequence of an overlapping sequence pair, including quality value processing

Endpoints:

← **alignment (Conveyor.Alignment.Pairwise.PairwiseAlignment<T,U,Conveyor.QualitySequence.IQualitySequence>)** pairwise alignment to process
→ **consensus (Conveyor.QualitySequence.SimpleDNAQSequence)** The calculated consensus sequence

Configuration items:

misMatchResolution (- unknown type -) Resolution of mismatching bases

Conveyor.SolexaProcessing.FuzzyMatchSequence<T,U>

Matches two sequences, allowing a number of mismatches

Endpoints:

← **pattern (U)** sequence to search for
← **sequence (T)** sequence to search in
→ **position (Conveyor.Math.Integer)** start position of the first match, or -1 if no match was found

Configuration items:

maxMismatches (32 bit signed integer) number of mismatches allowed

Conveyor.SolexaProcessing.FuzzyMatchString<T>

Matches a sequence vs. a given string, allowing a number of mismatches

Endpoints:

← **pattern (Conveyor.Text.String)** string to search for
← **sequence (T)** sequence to search in
→ **position (Conveyor.Math.Integer)** start position of the first match, or -1 if no match was found

Configuration items:

maxMismatches (32 bit signed integer) number of mismatches allowed

Conveyor.SolexaProcessing.GetCustomLinkerPosition

Calculates the position of a given linker in a Solexa read

Endpoints:

← **read (Conveyor.BioIO.Solexa.SolexaRead)** read to analyse
→ **position (Conveyor.Math.Integer)** position of the linker, or -1 if no linker is found

Configuration items:

linker (literal string) linker sequence to search for
maxMismatches (32 bit signed integer) number of mismatches allowed
searchReverse (boolean) also search for reverse complement

Conveyor.SolexaProcessing.GetLongestHomopolymerStretchSize<T>

returns the length of the longest homopolymer found in the sequence (ignoring 'N's)

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.SolexaProcessing.GetNumberOfOverlappingBases<T,U>

Returns the number of overlapping bases

Endpoints:

← **input (Conveyor.SolexaProcessing.SequenceOverlap<T,U>)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.SolexaProcessing.GetOverlapSequences<T,U>

Returns the sequence pair involved in the overlap

Endpoints:

← **input (Conveyor.SolexaProcessing.SequenceOverlap<T,U>)** - no description given -
→ **output (Conveyor.SequencePair.SequencePair<T,U>)** - no description given -

Conveyor.SolexaProcessing.GetPELinkerPosition

Calculates the position of a PE linker in the Solexa read

Endpoints:

← **read (Conveyor.BioIO.Solexa.SolexaRead)** read to analyse
→ **position (Conveyor.Math.Integer)** position of the linker, or -1 if no linker is found

Configuration items:

maxMismatches (32 bit signed integer) number of mismatches allowed

Conveyor.SolexaProcessing.OverlapPairedEndSequences<T,U>

calculates the overlap of CeBiTec style Solexa paired end reads (perfect matching)

Endpoints:

← **firstSequence (T)** first sequence to overlap
← **secondSequence (U)** second sequence to overlap
→ **overlap (Conveyor.SolexaProcessing.SequenceOverlap<T,U>)** the calculated overlap for further processing

Configuration items:

gapValue (8 bit signed integer) quality value for gaps at start and end of aligned sequences
maxMismatches (64 bit unsigned integer) number of mismatches allowed
minBases (64 bit unsigned integer) the minimal number of bases to consider the pair overlapping

B.26 Conveyor.Table

Data types

Conveyor.Table.TableHeader - no description given -

Conveyor.Table.TableRow - no description given -

Node types

Conveyor.Table.AddField

adds the content to a table row

Endpoints:

← **contentIn (Conveyor.Core.Data)** content to be added
← **rowIn (Conveyor.Table.TableRow)** row to add data to
→ **rowOut (Conveyor.Table.TableRow)** row after adding data

Configuration items:

tableHeader (literal string) header of this column

Conveyor.Table.AddRow

adds a row to an existing one

Endpoints:

← **appendeeRow (Conveyor.Table.TableRow)** row to be added
← **rowIn (Conveyor.Table.TableRow)** row to append row to
→ **rowOut (Conveyor.Table.TableRow)** row after adding

Conveyor.Table.EmptyRow

creates new, empty rows

Endpoints:

→ **output (Conveyor.Table.TableRow)** - no description given -

Configuration items:

firstRowIsHeader (boolean) flag to indicate that first row send by this node is a table header

Conveyor.Table.NewRow

creates a new row and put the content in it

Endpoints:

← **input (Conveyor.Core.Data)** - no description given -
→ **output (Conveyor.Table.TableRow)** - no description given -

Configuration items:
tableHeader (literal string) table header of this column

Conveyor.Table.TableCSVWriter

converts table rows to a CSV table

Endpoints:
← **input (Conveyor.Table.TableRow)** row to be added

Conveyor.Table.TableTSVWriter

converts table rows to a TSV table

Endpoints:
← **input (Conveyor.Table.TableRow)** row to be added

B.27 Conveyor.Text

String and string manipulation for Conveyor

Data types

Conveyor.Text.String - no description given -

Node types

Conveyor.Text.Concat

concatenates two string with an optional delimiter

Endpoints:
← **firstArgument (Conveyor.Text.String)** the first argument of the operation
← **secondArgument (Conveyor.Text.String)** the second argument of the operation
→ **result (Conveyor.Text.String)** the result of the operation

Configuration items:
delimiter (literal string) optional delimiter

Conveyor.Text.JoinElements

concatenates the string representation of the elements

Endpoints:
← **input (Conveyor.List.List<Conveyor.Text.String>)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Configuration items:
delimiter (literal string) delimiter for elements

Conveyor.Text.Length

returns the length of the input string

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Text.LexicalCompare

compares two strings lexically

Endpoints:

← **firstArgument (Conveyor.Text.String)** the first argument of the operation
← **secondArgument (Conveyor.Text.String)** the second argument of the operation
→ **result (Conveyor.Comparison.ComparisonResult)** the result of the operation

Conveyor.Text.LineReader

reads single lines from a given file

Endpoints:

→ **line (Conveyor.Text.String)** a single line read from the file

Configuration items:

filterEmptyLines (boolean) flag to indicate whether empty lines should be filtered
inputFile (input file) file to process

Conveyor.Text.LineSplitter

trims a string and splits it at a given regular string

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.List.List<Conveyor.Text.String>)** - no description given -

Configuration items:

delimiter (literal string) string to use a delimiter

Conveyor.Text.RegExpCut

use a regular expression to process a string and remove matching parts

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Configuration items:

regularExpression (literal string) .NET regular expression to use

Conveyor.Text.RegExpMatch

applies a regular expression to a string and returns a list of matches

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.List.List<Conveyor.Text.String>)** - no description given -

Configuration items:

regExp (literal string) regular expression to match

Conveyor.Text.StringGenerator

generator that produces a string value

Endpoints:

→ **output (Conveyor.Text.String)** - no description given -

Configuration items:

value (literal string) string value to generate

Conveyor.Text.StringMatch

returns a bool indicating whether a configured substring is part of the input

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Configuration items:

substring (literal string) substring to search

Conveyor.Text.StringReplace

replaces every occurrence of a given string with a given substitution

Endpoints:

← **input (Conveyor.Text.String)** - no description given -
→ **output (Conveyor.Text.String)** - no description given -

Configuration items:

match (literal string) the string to replace
replacement (literal string) the string to use as replacement

Conveyor.Text.SubstringMatch

returns true if hte input string contains the sub string

Endpoints:

← **firstArgument (Conveyor.Text.String)** the first argument of the operation
← **secondArgument (Conveyor.Text.String)** the second argument of the operation
→ **result (Conveyor.Logic.Boolean)** the result of the operation

B.28 Conveyor.Tree

Generic tree implementation for Conveyor plugins

Data types

Conveyor.Tree.ITreeVertex<T> Common base class of leaves and branches

Node types

Conveyor.Tree.GetChildren<T>

Returns a list containing the child vertices of the given vertex

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.List.List<T>)** - no description given -

Conveyor.Tree.GetDepth<T>

Returns the depth of a vertex (0 for root vertex, > 0 for nested vertices)

Endpoints:

← **input (T)** - no description given -
→ **output (Conveyor.Math.ULong)** - no description given -

Conveyor.Tree.GetLCA<U>

Get the lowest common ancestors to the given tree vertices

Endpoints:

← **vertices (Conveyor.List.List<U>)** list of vertices to calculate LCA of
→ **lca (U)** LCA of list of vertices

Conveyor.Tree.GetRoot<T>

Returns the root of the given tree vertex

Endpoints:

← **input (T)** - no description given -
→ **output (T)** - no description given -

Conveyor.Tree.IsLeafVertex<T>

Returns true if the vertex is a leaf vertex

Endpoints:

← **input (Conveyor.Tree.ITreeVertex<T>)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Conveyor.Tree.IsRootVertex<T>

Returns true if the given tree vertex is the root vertex of a tree

Endpoints:

← **input (Conveyor.Tree.ITreeVertex<T>)** - no description given -
→ **output (Conveyor.Logic.Boolean)** - no description given -

Bibliography

[ABJ$^+$04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher und S. Mock. *Kepler: an extensible system for design and execution of scientific workflows. Scientific and Statistical Database Management, 2004. Proceedings.*, Seiten 423–424, 2004.

[AMS$^+$97] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller und D.J. Lipman. *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research*, 25:3389–402, 9 1997.

[BKML$^+$10] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell und Eric W. Sayers. *GenBank*. *Nucleic Acids Research*, 38:D46–51, 2010.

[BO99] J.H. Badger und G.J. Olsen. *CRITICA: coding region identification tool invoking comparative analysis. Molecular biology and evolution*, 16(4):512, 1999.

[Con10] The UniProt Consortium. *The Universal Protein Resource (UniProt) in 2010*. *Nucleic Acids Research*, 38:D142–148, 2010.

[DHK$^+$99] A.L. Delcher, D. Harmon, S. Kasif, O. White und S.L. Salzberg. *Improved microbial gene identification with GLIMMER*, 1999.

[DRGS08] David De Roure, Carole Goble und Robert Stevens. *The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows*. *Future Generation Computer Systems*, 25:561–567, May 2008.

[DSS$^+$05] E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob und D. S. Katz. *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems*. *Scientific Programming Journal*, 13:219–237, 2005.

[Edg04] Robert C. Edgar. *MUSCLE: multiple sequence alignment with high accuracy and high throughput*. *Nucleic Acids Research*, 32:1792–1797, 2004.

[EJL$^+$03] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs und Yuhong Xiong. *Taming heterogeneity - the Ptolemy approach*. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[F$^+$89] J. Felsenstein et al. *PHYLIP (phylogeny inference package)*. *Cladistics*, 5(1):164–166, 1989.

[GNT$^+$10] J. Goecks, A. Nekrutenko, J. Taylor et al. *Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences*. *Genome biology*, 11(8):R86, 2010.

[HCL$^+$10] D. Hyatt, G.L. Chen, P.F. LoCascio, M.L. Land, F.W. Larimer und L.J. Hauser. *Prodigal: prokaryotic gene recognition and translation initiation site identification*. *BMC bioinformatics*, 11(1):119, 2010.

[HWS$^+$06] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li und T. Oinn. *Taverna: a tool for building and running workflows of services. Nucleic Acids Research*, 34:729–732, 2006.

[LGG11] B. Linke, R. Giegerich und A. Goesmann. *Conveyor: a workflow engine for bioinformatic analyses*. *Bioinformatics*, 27(7):903, 2011.

[NMM$^+$09] B. Neron, H. Menager, C. Maufrais, N. Joly, J. Maupetit, S. Letort, S. Carrere, P. Tuffery und C. Letondal. *Mobyle: a new full web bioinformatics framework*. *Bioinformatics*, 25(22):3005, 2009.

[RLB00] P. Rice, I. Longden und A. Bleasby. *Emboss: The european molecular biology open software suite (2000)*. *Trends in Genetics*, 16(6):276–277, 2000.

[SRBU09] M. Senger, P. Rice, A. Bleasby und M. Uludag. *Soaplab: open source web services framework for Bioinformatics programs*. In *The 10th Annual Bioinformatics Open Source Conference. Satellite Workshop of the Joint 17th Annual International Conference on Intelligent Systems for Molecular Biology (ISMB 2009) and 7th European Conference on Computational Biology, ECCB*, Band 168, Seiten 27–28, 2009.

[Ste92] W.R. Stevens. *Advanced Programming in the UNIX Environment* (Addison Wesley, 1992).

[THG94] J.D. Thompson, D.G. Higgins und T.J. Gibson. *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice*. *Nucleic acids research*, 22(22):4673, 1994.

[TRHD07] P. Troeger, H. Rajic, A. Hass und P. Domagalski. *Standardization of an API for Distributed Resource Management Systems.* In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, Seiten 619–626, May 2007.

[TTL05] Douglas Thain, Todd Tannenbaum und Miron Livny. *Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[WSK+08] MD Wilkinson, M. Senger, E. Kawas, R. Bruskiewich, J. Gouzy, C. Noirot, P. Bardou, A. Ng, D. Haase, A. Saiz Ede et al. *Interoperability with Moby 1.0 - it's better than sharing your toothbrush*. *Brief Bioinform*, 9(3):220–231, 2008.