Component-based workflow framework for gene expression microarray analysis

Kristian Ovaska

Helsinki October 2, 2008

Master’s thesis

UNIVERSITY OF HELSINKI

Department of Computer Science
Computational methods have a crucial role in contemporary biomedical research. Various high-throughput measurement techniques produce large quantities of data that must be analyzed computationally. Basic requirements for data analysis include correctness, efficiency and repeatability. In addition, analysis software should be flexible as biological experiments vary in setup and in objectives. Further, the bioinformatics field is relatively young and novel data analysis methods are constantly introduced. Inclusion of such methods to analysis workflows should be straightforward.

We have designed and implemented Anduril (ANalysis of Data Using Rapid Integration of Algorithms), a component-based workflow framework for scientific data analysis. Components are the building blocks of data analysis that implement well-defined tasks such as data import or report generation. Components are connected together into a network, also called a workflow, so that the output of one component is used as the input of another. The workflow engine of Anduril provides advanced features such as nested workflows and optimized repeated execution of a workflow.

We provide ready-made components for gene expression microarray analysis. Expression microarrays measure the rate of conversion from DNA to RNA and are a basic tool in biomedical research. The usefulness of Anduril is demonstrated using a real-world microarray analysis case study concerning a cancer-related virus called Kaposi's sarcoma associated herpesvirus (KSHV).


Keywords: object-oriented frameworks, bioinformatics, microarray analysis
Contents

1 Introduction 2

2 Object-oriented frameworks, components and patterns 5
  2.1 Software reuse .......................... 5
  2.2 Components .......................... 6
    2.2.1 CORBA ........................... 7
  2.3 Object-oriented frameworks ................. 8
    2.3.1 Definition of a framework ............ 8
    2.3.2 Structure of frameworks ............ 9
    2.3.3 Advantages and disadvantages of frameworks .......................... 11
    2.3.4 Framework hierarchy .................. 12
    2.3.5 Frameworks and components .......... 13
    2.3.6 Frameworks and design patterns ........ 14

3 Basics of molecular biology 15
  3.1 DNA .................................. 15
  3.2 Gene expression ........................ 16
    3.2.1 RNA, protein and gene ............... 16
    3.2.2 Transcription ........................ 17
    3.2.3 Translation .......................... 18

4 Gene expression microarrays 20
  4.1 Microarray types ........................ 20
    4.1.1 DNA arrays .......................... 21
    4.1.2 Measuring expression with DNA arrays .......................... 22
4.1.3 Oligonucleotide and cDNA arrays .................................. 23
4.1.4 One channel and two channel arrays .............................. 23
4.2 Experiment design ...................................................... 24
4.3 Preprocessing ............................................................ 25
  4.3.1 Intensity normalization and log ratios ............................ 26
  4.3.2 Background correction ............................................. 26
  4.3.3 Combining samples and probes ................................... 27
4.4 Finding differentially expressed genes ............................... 27
  4.4.1 Fold change ......................................................... 28
  4.4.2 Student’s test ....................................................... 28
  4.4.3 Multiple comparison correction .................................. 30
4.5 Annotation and biological functionality .............................. 30

5 Architecture of the framework ............................................ 33
  5.1 Stakeholders and usage environment ................................ 33
  5.2 Design goals ........................................................... 34
  5.3 Existing work .......................................................... 35
  5.4 Architecture overview ............................................... 35
  5.5 Component model ..................................................... 38
    5.5.1 Component interface definition ................................ 39
    5.5.2 Component execution ............................................ 40
    5.5.3 Data types ........................................................ 41
    5.5.4 Language-specific mini-frameworks ............................ 43
    5.5.5 Example: adding matrices using R ............................. 44
    5.5.6 Generic components .............................................. 44
5.5.7 Delegates .............................................. 45
5.5.8 Comparison to CORBA ......................... 45
5.6 Network topology ..................................... 46
  5.6.1 Composite components ......................... 48
  5.6.2 Conditional branching ......................... 49
  5.6.3 Type checking and type parameter inference .... 50
5.7 Network execution ..................................... 51
  5.7.1 Network execution algorithm ................. 53
5.8 Support for black box testing .................... 54

6 Microarray analysis framework .......................... 57
  6.1 Mathematical formulation of microarray experiments ... 57
    6.1.1 Transformations between groups ............... 58
    6.1.2 Properties of group transformations .......... 59
    6.1.3 Derived group as a function of its ancestors ... 60
    6.1.4 Transformations between ID sets ............... 61
    6.1.5 Annotation functions ........................... 62
    6.1.6 ID conversion functions ....................... 64
  6.2 Data types ........................................... 64
  6.3 Component repository ................................ 65
    6.3.1 Automated report generation ................. 65
    6.3.2 Using SQL to transform CSV files .......... 66
    6.3.3 Statistical analysis ........................... 66

7 Case study ............................................... 67
Acknowledgements

I am grateful to Dr. Päivi Ojala for providing the microarray data used in the case study. The members of the Hautaniemi lab are thanked for implementing components and for providing valuable feedback on the Anduril framework. Finally, I thank the supervisors of this thesis for providing
1 Introduction

High-throughput measurement techniques used in biological research provide vast amounts of data that must be analyzed computationally. One important technology, gene expression microarrays, are used to measure the expression of genes on genome-wide scale [Dra03]. Gene expression is the conversion of DNA to RNA, which is the first step in protein synthesis [AJL02]. Since gene expression is a fundamental phenomenon in biology, expression microarrays have a large number of use cases [All06], including the research of diseases such as cancer.

Analysis of high-throughput data must be done efficiently while maintaining high quality and repeatability of scientific results. Flexibility in analysis software is needed because experiments vary in setup and in objectives. The field of microarray analysis is also quite young and new techniques are still being developed [All06].

Often, microarray analysis is done with interactive commercial applications or with custom-made scripts. Interactive applications are easy to use but are often inflexible. Custom scripts are flexible, but they are often ad hoc solutions that are difficult to reuse.

We present Anduril (ANalysis of Data Using Rapid Integration of aLgorithms), a component-based framework for microarray and other scientific analysis [OLH08b]. The framework provides a core that is extended to produce concrete analysis applications. New analysis techniques can be implemented as reusable components and integrated into analysis applications. Anduril provides a systematic and transparent architecture for analysis projects. Code and architecture reuse increase the productivity and quality of analysis. The framework is available under the GNU General Public License from http://csbi.ltdk.helsinki.fi/anduril/.

In Table 1, Anduril is compared to interactive applications and custom scripts. Compared to interactive programs, Anduril is much more flexible, although not as easy to use. Anduril may be a more systematic approach compared to interactive programs since the entire analysis is automated and documented.
custom scripts, Anduril is more systematic and enables code and design reuse. Anduril requires extensive programming experience if custom components are written; otherwise, only basic experience is required.

The architecture of Anduril is based on a network of components, also called a workflow architecture [GHS95]. Components are the building blocks of analysis and are responsible for a well-defined limited set of functionality. Components can be written in any language, which allows to select to most suitable tool for the task in hand. Concrete analysis applications are created by connecting components together so that the results of one component are used as the input of another. The network model provides flexibility since the components and their connections are selected based on project-specific requirements.

Scientific analysis is often iterative in nature. Anduril supports the iterative process by automatically executing only those components whose configuration or input has changed since the previous execution. Another productivity-enhancing feature is automatic parallelization of components. Components that do not depend on each other are executed in parallel, which allows to take advantage of multi-CPU architectures.

The rest of the thesis is structured as follows. Section 2 provides background for object-oriented frameworks and components, both of which are techniques for software reuse. Section 3 introduces basic concepts of molecular biology that are needed to understand microarray analysis. In section 4, techniques for microarray analysis are discussed. Section 5 introduces the basic architecture of the analysis framework, building on concepts developed in section 2. Section 6 provides details for

<table>
<thead>
<tr>
<th>Ease of use</th>
<th>Interactive</th>
<th>Custom script</th>
<th>Anduril</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexibility</td>
<td>easy</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Systematic/ad hoc</td>
<td>medium</td>
<td>ad hoc</td>
<td>systematic</td>
</tr>
<tr>
<td>Programming</td>
<td>no</td>
<td>extensive</td>
<td>basic/extensive</td>
</tr>
<tr>
<td>experience required</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Comparison of commercial interactive analysis programs, custom analysis scripts and the presented analysis framework.
microarray-specific components. In section 7, the framework is applied to a real-world microarray analysis.
2 Object-oriented frameworks, components and patterns

In this section we introduce three software engineering concepts: object-oriented frameworks, components and patterns. A framework is a partial application that can be specialized to produce a complete, custom application [FaS97]. A component is an encapsulation of code with contractually specified external interfaces that can be deployed independently [Szy98]. A design pattern is a solution to a common design problem that describes the context in which the solution works and the consequences of the solution, such as design trade-offs and alternatives [Gam95].

All three concepts are techniques for software reuse. Components represent code reuse, patterns represent design reuse and frameworks reuse both code and design [Joh97a]. Enabling reuse is the primary motivation for creating the framework for microarray analysis. The framework allows the analyst to use existing code and design instead of writing the analysis software from scratch.

2.1 Software reuse

Software reuse is the use of existing software or knowledge to construct new software [FrK05]. Software reuse is motivated by the goal to save time and consequently money in software engineering projects [Joh97a, Mat00, FrK05]. Another advantage of reuse is the uniformity it causes in products and in design [Joh97a]. For example, frameworks for graphical user interfaces enable consistent product look and feel across applications [Joh97a]. Patterns create a common vocabulary for developers that allow them to communicate efficiently and accurately [Vil04].

It can be argued that reusable components are more reliable than one-use components because they are tested and exercised more thoroughly [FrK05]. Further, developing a system with reusable components increases the reliability of the system as a whole [FrK05]. These hypotheses are still an open research problem in the software reuse community.
Components are the oldest software reuse technique, dating back to the 1960’s [Joh97a, FrK05]. However, components are limited in scope because they primarily enable code reuse and not design reuse. Design reuse is considered more valuable because it can be applied in more contexts due to its abstract nature and is applied earlier in the software process [Joh97a]. Design is the main intellectual product of a software process and is more difficult to create than code [Mat00]. Patterns are also limited in their scope because they do not enable code reuse. Frameworks stand between the two extremes of reuse techniques.

Reuse may also present challenges. We must make a tradeoff between simplicity and ease of use. A simple component with few parameters is easy to learn and use, but can be used in fewer contexts than a more complex component [Joh97a]. A complex components with many parameters is more powerful but also more difficult to use.

In some cases, reuse introduces faults into software. Two well known incidents, Therac-25 and Ariane 5 Flight 501, have been attributed to reuse [FrK05]. Therac-25 was a radiation therapy machine that caused the deaths of several patients [LeT93]. One reason for the failure was that the machine reused a software component from a previous version of the machine [FrK05]. Ariane 5 Flight 501 was the maiden flight of the Ariane 5 rocket. The rocket exploded after launch, with the cost of half a billion dollars, because the control software reused the specification of the earlier Ariane 4 [FrK05, JeM97].

2.2 Components

A component is a self-contained software element that interacts with its environment using contractually specified interfaces only [Szy98, chapter 4]. The interfaces state what the component requires from its environment and which services it provides. A component encapsulates its implementation and can be used without detailed knowledge of the internal structure of the component. A component is a unit of deployment: it is always deployed as a whole and never deployed partially. A component can be used by third parties by combining it with other components. This
A component has no persistent state. As a consequence, copies of a component are indistinguishable from each other, with the possible exception of attributes such as serial numbers that have no effect on component function. It is not meaningful to state that a software system uses multiple copies of a component, because the copies are not different from each other. A component may instantiated several times in a system, of course. The instances of components are collections of objects. An object is a unit of instantiation that has a unique identity and a state.

Components always have requirements for the environment they work in [Joh97a]. To enable component interchange, it is necessary to define component standards that define a common environment for components. This allows independent vendors to write components that can be used together in software systems. Some well known standards are OMG CORBA, Microsoft COM and Java Beans [Szy98, Ch. 13-15].

### 2.2.1 CORBA

CORBA is an elaborate standard that defines a distributed object model that is independent of programming language and computing platform [Szy98]. Object interfaces are specified using a platform-independent language called Interface Definition Language (IDL) [OMG08]. An IDL interface specifies the signatures of operations (methods) and member attributes. IDL contains a C-like type system that includes a comprehensive set of basic data types (like numbers and characters) and composite types (like structs and arrays) [OMG08].

The user and the provider of an object may be on different hosts and use different programming languages. When the user calls a CORBA method, the call is directed to an Object Request Broker (ORB) on the same host that marshals call parameters into a standardized representation. The parameters are transmitted on the network to an ORB on the host of the object provider. The ORB unmarshals the parameters and directs them to the actual method implementation. After the method is finished, the result is passed back to the caller in a similar fashion. The drawback of CORBA
is that it provides no efficient binary call interface for objects located on the same host: all calls must go through the ORB which may introduce performance overhead.

2.3 Object-oriented frameworks

2.3.1 Definition of a framework

There is no single definition for object-oriented frameworks [Joh97a]. The definition of a framework can be based on the purpose or the structure of the framework.

A definition that captures the purpose of frameworks is "a framework is a reusable, semi-complete application that can be specialized to produce custom applications." [FaS97] It emphasizes reusability and that the framework is not a complete application on its own.

A definition that refers to the structure of a framework is "a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact." [Joh97a] It makes clear that frameworks are closely related to object-oriented languages by referring to the concept of abstract classes. However, frameworks can also be written in other paradigms by mimicking the object-oriented machinery manually [Joh97a]. This definition does not emphasize code reuse, i.e., the fact that a framework also contains concrete classes whose implementation can be reused.

A definition that refers both to the structure and purpose of frameworks is "a framework is a set of classes that embodies an abstract design for solutions to a family of related problems." [Mat00] It takes into account the presence of both abstract and concrete classes. This definition is not limited to frameworks that are specialized to full applications. It includes frameworks that are used to create smaller software elements such as components.
2.3.2 Structure of frameworks

The abstract classes of a framework define the design of the components that belong to the framework [Joh97a]. Often frameworks have abstract classes or interfaces that directly correspond to the main concepts of the application domain, such as Button and Window for a GUI framework [Vil04]. Abstract classes provide a way to extend the framework by deriving concrete subclasses and attaching them to the framework [Mat00].

Mature frameworks often have layered structure where the highest layer contains pure interfaces [Vil04] such as Button and Window mentioned previously. A middle layer contains abstract classes that provide implementation for some, but not all, interface methods. Finally, a Default Component Layer (DCL) provides full concrete implementations of abstract classes and interfaces [Vil04]. In a GUI framework, the DCL might provide implementations for GUI primitives on a particular platform such as Windows Vista. Default components allow application writers to specialize the framework without writing new components.

A key feature of frameworks is inversion of control [Joh97a, Mat00, Vil04, FaS97]. In a traditional application that uses a third-party code library, the application owns the thread of control and calls the library when the application needs its services [Joh97a]. In a framework-based approach, the framework owns the thread of control and calls application-specific parts when appropriate [Joh97a]. The framework spawns events that the application-specific code handles by implementing hook methods of the framework [Vil04].

Naturally, application-specific parts may call services of the framework, so control can flow in both directions [Mat00]. This is illustrated in Figure 1. Inversion of control is also called the Hollywood principle: “don’t call us, we’ll call you” [Mat00].

Frameworks can be classified to black-box or white-box frameworks based on the extension mechanisms they provide. In a white-box framework extension is done by deriving subclasses from framework superclasses and overriding their hook methods.
Figure 1: In a framework, control can flow from the framework to the application and vice versa, whereas in a traditional application, control flow is one-way [Mat00].

In a black-box framework extension is done by instantiating, configuring and combining existing components [Vil04]. A black-box framework where component configuration and combining is done visually with a separate graphical program can be classified as a visual builder framework [Mat00]. Examples of this class are visual GUI builders for GUI frameworks [Mat00]. Most frameworks have elements of both black-box and white-box mechanisms [Vil04, Joh97a] and are called gray-box frameworks [Vil04].

White-box frameworks are more flexible than black-box frameworks. They are also more difficult to use, since they require knowledge about the internals of the framework superclasses that are being extended [Vil04]. Black-box extension only requires knowledge on the external interface of components [Vil04]. However, black-box frameworks are more difficult to design [Vil04]. Frameworks usually develop from a white-box framework towards a black-box framework during their evolution [Vil04].

Deriving an application from a framework is called specialization or instantiation [Vil04]. The application is composed of four parts: (i) the framework, (ii) application-specific classes that extend classes from the framework, (iii) glue code that instantiates and connects components and (iv) elements that are not related to the framework [Joh97a, Vil04]. For pure black-box frameworks, part (ii) is not present.

The reuse interface is the interface between the framework and the specialized application [Vil04]. It is illustrated in Figure 2. The interface is a collection of hot spots, i.e., aspects of the framework that must be kept flexible [Vil04].
2.3.3 Advantages and disadvantages of frameworks

The primary advantages of frameworks are related to reuse. The motivation for reuse was discussed in section 2.1. Frameworks enable code reuse because they contain concrete classes that can be reused by applications. A class that is derived from a framework superclass reuses code by inheriting methods from the superclass [Joh97a]. Frameworks enable design reuse because the application follows the architecture of the framework. The framework also provides a common vocabulary for applications, which represents problem domain analysis reuse [Joh97a]. Developers of a GUI application can use the concepts defined by the GUI framework instead of having to define their own vocabulary.

However, frameworks are not a silver bullet [BrB87] and have several disadvantages. As discussed in section 2.1, reusing a complex component is more difficult than reusing a simple component. Frameworks are often big and complex and their biggest problem is learning to use them [Vil04, Joh97a, FaS97]. The classes of a framework collaborate with each other, so they cannot be learned one at a time but must be learned as a whole [Joh97a, Vil04]. This makes frameworks more difficult to learn than traditional class libraries, where the user can focus on one class at a time [Vil04]. The indirection inherent in frameworks also contributes to their steep learning curve [Vil04]. It can take 12 months to become highly productive with a GUI framework such as MFC [FaS97]. As Johnson writes, "you can often tell that a class library is a framework if [...] programmers who are learning it complain about its complexity." [Joh97b]
Designing a framework is considerably more difficult than designing an ordinary application [Joh97a, Vil04, Mat00]. Framework design process takes iteration, even for experienced developers [Joh97a]. The primary difficulties are deciding which parts of the system should be flexible (i.e., what constitutes the reuse interface) and how to specify and implement the necessary flexibility in such a way that the system remains simple [Vil04]. Because designing a framework is a difficult and unpredictable process, it should not be on the critical path of an important software project [Joh97a].

Changing a framework changes all applications that use the framework [Joh97a], so maintenance must be done in a careful way. A framework that has not matured enough should not be widely deployed because its design is still changing and application developers have to hit a running target [Joh97a]. Of course, a framework cannot mature if it is not used at all, so a new framework should be used in a small pilot project to gain experience on using it [Joh97a].

The application that uses a framework must follow (i.e., reuse) the design of the framework [Joh97a]. This means that the framework must be selected with care. Framework evaluation can be difficult because the primary function of frameworks is to be extensible, and extensibility is difficult to measure [Joh97a]. It is also difficult to tell whether a framework is reliable and efficient enough [Joh97a].

It is sometimes necessary to use several frameworks in one application. This can introduce various framework composition problems such as several frameworks assuming they own the main thread of control [Vil04, Mat00].

2.3.4 Framework hierarchy

The traditional view is that a framework is specialized into an application in one step. However, specialization of a framework can also be a multi-step process where we start from a framework at level $n$, specialize it into a level $n-1$ framework, and continue this until we have $n$ frameworks [Vil04]. The framework at level 1, also called a first order framework [Mat00], is then specialized into an application.
A framework at level $k$ can be specialized into more than one framework at level $k - 1$. This is the case with backbone framework architecture illustrated in Figure 3 [Mat00]. A level 2 backbone framework provides core facilities and hooks for more specialized level 1 frameworks. The level 1 frameworks can be considered as flexible components of the backbone framework [Mat00].

### 2.3.5 Frameworks and components

Frameworks and components can be thought of as different technologies that work together and supplement each other [Joh97a]. As discussed in section 2.2, components need to make assumptions about their environment. Frameworks provide a standard environment for components and facilities to combine components together [Joh97a]. Component systems like CORBA can then be thought of as component frameworks [Joh97a].

Frameworks also make implementing new components easier [Joh97a]. A new component can use services from other component and can inherit method implementations from superclasses provided by the framework.

Frameworks can also be considered components, as we saw in the context of hierarchical frameworks. However, frameworks are designed to be flexible and because of this, their interfaces are much more complex than interfaces of regular components [Joh97a].

Figure 3: Backbone-based framework hierarchy [Mat00].
2.3.6 Frameworks and design patterns

Design patterns and frameworks are related in that they both allow design reuse. However, patterns are more limited in scope and describe solutions to relatively small problems [Vil04]. Frameworks are also more concrete than patterns: frameworks are code while patterns represent ideas [Vil04]. The code in a framework provides an "oracle" for answering questions about the framework [Joh97a], since ultimately the framework is unambiguously defined by the code.

Patterns are the architectural elements of frameworks in that many frameworks utilize patterns to provide extension points [Joh97a, Vil04]. In fact, the relationship also goes the other way: the design patterns of the seminal "Gang of Four" book [Gam95] were discovered by examining existing frameworks [Joh97a].

Since patterns provide a common vocabulary for developers and they are used in implementing frameworks, they can be used to efficiently document the reuse interface and architecture of frameworks [Vil04].
3 Basics of molecular biology

In this section, we develop enough molecular biology to be able to understand gene expression. The discussion is based on *Molecular Biology of the Cell* [AJL02].

All known organisms are composed of cells. A cell is separated from its environment by the membrane, a selective barrier that allows the cell to function as a coordinated chemical system [AJL02, p. 11].

Organisms are classified into *eukaryotes* (animals, plants and fungi) and *prokaryotes* (bacteria and archaea) based on the structure of their cells. Eukaryote cells, in contrast to prokaryote cells, have a *nucleus* that houses DNA.

3.1 DNA

Almost all cells are able to divide, i.e., create copies of themselves. In division the cell passes its hereditary information to the daughter cells. The hereditary information is stored in DNA. DNA influences how the cell functions, and by consequence, how the whole organism functions.

On molecular level, the building blocks of DNA are *nucleotides*. A nucleotide is composed of a sugar phosphate and a base. The base is one of adenine (A), guanine (G), cytosine (C) or thymine (T), and it gives the nucleotide its identity. Sugar phosphates are connected together to form a *strand*. DNA can be written a string of characters from the alphabet \{\text{A, G, C, T}\}.

Two strands going in opposite directions can be joined together to create a double-stranded DNA molecule. The bases connect the two strands and the sugar phosphates form the backbones of the molecule. The base A pairs with T and G pairs with C. This makes the double-stranded structure unique: given the nucleotides in one strand, we can deduce the structure of the other strand. Connected bases are called *base pairs*. The strands twist around each other to create a double helix DNA molecule. The double helix is chemically robust and facilitates easy replication of DNA. Since both strands contain all information, each strand can individually act
Structure of two-stranded DNA (adapted from Alberts et al. [AJL02, p. 5]). For clarity, the twists are not shown. The orange rectangles are sugars and yellow circles are phosphates. Together they form the backbone of a strand. A nucleotide is a combination of a sugar phosphate with one of the bases A, G, C or T. Eight base pairs containing 16 nucleotides are shown.

as a template for a new DNA molecule.

Structure of two-stranded DNA without twists is visualized in Figure 4. As can be seen, given the sequence in one strand (ACTGGCAA) the sequence in the other strand in unambiguous (TGA CCGTT).

3.2 Gene expression

DNA has little functionality on its own. In eukaryote cells, it does not normally leave the cell nucleus. DNA must be transformed into other molecules, commonly proteins, that are the actuators of DNA. The process is called gene expression. Protein synthesis has two phases. In the first phase called transcription a DNA segment is transformed into an RNA molecule. Translation takes place in the nucleus. In the second phase called translation the RNA molecule is transformed into a protein. Transcription takes places in cellular organs called ribosomes. In some cases transcription produces RNA molecules that have functionality on their own and translation does not take place.

3.2.1 RNA, protein and gene

RNA is very similar to DNA with three differences. RNA uses a different sugar (ribose) as the backbone molecule. Instead of the base thymine (T) it uses uracil
RNA is single-stranded instead of double-stranded and because of this, it is able to fold into complex 3D shapes. Most RNA molecules are used as an intermediate form in the conversion of DNA to proteins. These molecules are called messenger RNAs, or mRNAs.

A protein is a complex molecule composed of a combination of amino acids. There are 20 standard types of amino acids. A protein can be written as a sequence of characters from an alphabet of size 20. However, proteins fold into complex 3D shapes whose configuration is not readily apparent from the protein sequence. Proteins form most of the dry mass of cells. They have a large number of responsibilities. For example, receptors are proteins that bind to the membrane and allow other molecules to pass the cell wall. Enzymes catalyze, or accelerate, chemical reactions. Transcription factors initiate or regulate transcription, allowing one protein to regulate the creation of another. All these processes create feedback loops into cells and are crucial for the functioning of complex organisms.

Defining a gene is somewhat complicated and historically the definition has changed when new knowledge has challenged the previous definition [GBR07]. A traditional definition is that a gene is a segment of DNA sequence that corresponds to a single protein or a single functional RNA molecule [AJL02, p.9]. However, recent advances in molecular biology indicate that the definition is more complex [GBR07]. One currently proposed definition of gene is "a union of genomic sequences encoding a coherent set of potentially overlapping functional products." [GBR07] According to this definition, a gene can be located across several chromosomes and RNA can be a part of a gene as well. For the purposes of this thesis, the first simple definition is sufficient.

3.2.2 Transcription

The transcription mechanism has some differences between eukaryotes and prokaryotes [AJL02, p. 306–310]. We will limit our discussion to eukaryotes.

A protein called RNA polymerase is responsible for converting a DNA sequence into
the corresponding RNA sequence. The RNA polymerase can not initiate transcription on its own but it works in conjunction with additional proteins called transcription factors. Transcription factors bind near the beginning of the gene in the DNA molecule and enable transcription to initiate. Some factors bind to a specific DNA sequence and allow transcription to be accurately regulated. The RNA polymerase produces an RNA molecule that is called pre-mRNA because it must be further processed before it is exported from nucleus.

Not all regions of a gene are coded into functional products. The gene can be divided into introns (intragenic regions) and exons (expressed regions). Introns are removed from the RNA molecule in a process called splicing that occurs in the nucleus. Splicing is done by a set of interacting molecules called a spliceosome. Splicing is not necessarily unique. In some cases an exon might be removed from the RNA molecule, allowing the same DNA region to create several different types of RNA molecules. This phenomenon is called alternative splicing and it greatly increases the expressive power of the DNA mechanism [GBR07].

After the pre-mRNA molecule is spliced, it is exported from the nucleus and is called mRNA. RNA export is selective in that not all RNA sequences are exported. For example, the intron sequences that are removed by spliceosomes are destroyed in the nucleus [AJL02, p. 327].

3.2.3 Translation

Conversion of RNA to protein takes place in a cellular organ called ribosome. The ribosome is a large complex of more than 50 individual proteins and RNA molecules [AJL02, p. 10]. While transcription is a relatively simple process, translation is more complex because a single RNA nucleotide cannot be mapped to a single amino acid: there are 20 amino acids and only 4 nucleotides. The solution is to use three nucleotides for one amino acid. The combination of three nucleotides is called a codon. Since there are $4^3 = 64$ possible codons, some amino acids are coded by several codons. For example, isoleucine is coded by the codons AUA, AUC and
Figure 5: Transcription and translation in eukaryote cells. Protein folding and post-translational modifications are not shown. Transcription factors are omitted for clarity. In the pre-mRNA and mRNA images in the nucleus, green denotes exons and red denotes introns. In the ribosome, a magnified version of mRNA is shown. The translation phase on the right is based on Figure 1-10 in *Molecular Biology of the Cell* [AJL02, p. 10].

AUU [AJL02, Table 1].

Codons are read by a class of RNA molecules called *transfer RNAs* or tRNAs. Each tRNA molecule class has on one end an *anticodon*, a sequence of three RNA nucleotides that binds to a specific codon [AJL02, p. 8]. On the other end, the tRNA binds to a specific amino acid. For example, a tRNA molecule corresponding to the AUA codon has the anticodon sequence UAU and binds to the amino acid isoleucine.

In the ribosome, tRNAs loaded with amino acids bind to the mRNA one at a time. The amino acids of tRNA are bound together to form a protein chain. The process ends when the ribosome encounters one of the special codons UAA, UAG or UGA that are called *stop codons* [AJL02, p. 349]. The protein folds into a 3D shape by creating internal bonds between the amino acids. The protein may be further altered by *post-translational modifications* which are triggered by enzymes.

Transcription and translation are summarized in Figure 5.
4 Gene expression microarrays

*Gene expression microarrays* measure gene expression levels for a number of genes simultaneously, often for the whole genome [Kaw06]. They have a vast range of use cases in molecular biology and biomedicine. There are over 27000 articles with the keyword "microarray" in the PubMed database. Microarrays allow various types of research, such as comparing gene expression between different tissue types or between healthy and tumor tissue and studying effects of drugs [Dra03]. In future, microarrays may be used to diagnose diseases such as cancer [Kaw06]. Since microarrays were introduced in 1995 [SSD95], their use has increased exponentially [Kaw06] and they are now almost ubiquitous in biological research [All06].

The use of microarrays in medical research requires cross-disciplinary expertise from several fields. Microarray manufacturing applies sophisticated techniques that are similar to the technology used to build integrated circuits for computers [Dra03, Ch. 2]. Biologists are needed to formulate the research question and to interpret results. Laboratory personnel handle biological samples and conduct the experiment. Knowledge of statistics is necessary in order to make sound statistical inferences on expression levels [Dra03]. Finally, computer science is an integral part of microarray analysis because the data quantities involved are well out of reach of manual analysis. A typical microarray study could consist of 20 microarrays, each of which contains 40,000 probes. In total, there are 800,000 data points.

Figure 6 shows the steps of microarray analysis projects. In this section, we will go through the steps and gain an understanding of the process from microarray manufacturing to making biological conclusions.

4.1 Microarray types

There are several types of commercial and custom-made microarrays available. Different preprocessing methods are needed for different microarray types, and an analysis software that supports several microarray types is usable in more contexts than
a software limited to a particular type.

4.1.1 DNA arrays

Expression microarrays are a subtype of DNA arrays. A DNA array is a substrate, such as glass or plastic, that contains single-stranded DNA probes [Dra03, Ch. 2]. Depending on the array type, length of probe sequences ranges from 25 base pairs to thousands [Kaw06]. Probes are used to detect complementary single-stranded DNA molecules called targets that are extracted from a biological sample [Dra03, Ch. 2]. As was seen in section 3, when two DNA strands containing complementary sequences are placed in vicinity, they bind to create double-stranded DNA. This process is called hybridization [Dra03, Ch. 2].

As a simplified example, let us assume we want to measure the amount of DNA molecules corresponding to the sequence ACTGGCAA in the sample. We place probes containing the complementary sequence (TGA CCGTT) on the array and pour target DNA on the array, allowing DNA to hybridize.

In order to read DNA levels, target DNA is labeled with a fluorescent dye [Dra03, Ch. 2]. After hybridization, the array is illuminated with a light source that excites the dye and the array is scanned. This produces a bitmap image that can be processed computationally. The intensity of a spot correlates with the quantity of corresponding DNA in the sample [Dra03, Ch. 2].
4.1.2 Measuring expression with DNA arrays

So far, we have discussed measuring DNA levels, but our main interest is in measuring expression levels. In expression microarray experiments, we extract mRNA from the sample and convert it to complementary DNA (cDNA) using reverse transcription [Dra03, Ch. 2]. Conversion to cDNA is done because mRNA degrades quickly [ALN03], while DNA is chemically more stable. This allows us to use DNA arrays to measure mRNA levels. The process from mRNA extraction to microarray scanning is shown in Figure 7. Notice that the base U changes to base T when reverse transcription takes place.
4.1.3 Oligonucleotide and cDNA arrays

Expression microarrays can be categorized into oligonucleotide and cDNA arrays based on manufacturing technique. Oligonucleotide probes are short, between 25 and 70 base pairs in length [Kaw06]. They are synthesized on the chip during manufacturing, known as in situ synthesis [Dra03, Ch.2]. The synthesis technique restricts the length of probes [Dra03, p.18]. In cDNA arrays, probes are fragments of complementary DNA that are prepared separately from the microarray chip and deposited on the array by robots [Dra03, p.17]. The probes can be much longer than oligonucleotides, up to several thousand base pairs in length [Kaw06].

While the first microarrays were cDNA arrays, all current commercial arrays are oligonucleotide arrays [Kaw06]. Complementary DNA arrays are still used in academic and private institutions [Kaw06]. Such arrays are called home brew arrays because they are often printed by the same institution that conducts the experiments [Kaw06].

In oligonucleotide arrays, a large number of probes are printed on the same location, called a spot [Kaw06]. Spot size between different manufacturers ranges from 100$\mu$m$^2$ to over 10,000$\mu$m$^2$ and the number of molecules is between $10^6$ and $10^{10}$ [Kaw06]. Spots are visible in scanned images and they are the basic unit for computational analysis.

4.1.4 One channel and two channel arrays

Orthogonally to oligonucleotide/cDNA categorization, expression arrays can be divided into one channel and two channel arrays [Dra03, Ch.2]. In one channel arrays, one biological sample is hybridized on the array, while in two channel arrays, there are two samples.

Affymetrix is a well known manufacturer of one channel arrays [Dra03]. Affymetrix arrays use short oligonucleotides with 25 base pairs [Kaw06]. In Affymetrix arrays, probes are divided into perfect match (PM) probes and mismatch (MM) probes [Dra03]. The perfect match probe contains the sequence of a DNA segment of
interest, usually a part of a gene. Spatially close to the PM probe is a mismatch probe that has the same sequence as the PM probe except the middle nucleotide is changed [Dra03]. Ideally, the DNA molecule should bind to the corresponding PM probes but not to MM probes. For each gene, there are several PM and MM probes on the array. The expression level of the gene is the average difference between corresponding PM and MM probes [Dra03, Ch.2].

In two channel arrays, mismatch probes are not used. The samples are labeled with different dyes to distinguish between them [Dra03, Ch.2]. Two common cyanine-based dyes are called Cy3 and Cy5 [BHV06, MEM93]. When a two-channel array is scanned, Cy3 is first excited with green light [MEM93] and the luminosity levels are read [Dra03, Ch.3]. Then, Cy5 is excited with red light and the array is scanned again. This produces two monochromatic bitmap images. For visualization, the images are often combined by colorizing the Cy3 image with green color and the Cy5 image with red color and overlaying the images [Dra03, Ch.3]. The composite image contains green in spots where the Cy3 sample is more expressed and red in spots where the Cy5 sample is more expressed. When the two expression levels are similar, the composite spot is yellow. Non-expressed spots are black.

Two channel arrays may introduce dye effects, non-linear differences between the two dyes. Dye effect can be compensated for in preprocessing [Dra03, Ch.12].

4.2 Experiment design

Experiment design is an important part of the microarray analysis process [All06, Dra03]. In a designed experiment, the researcher controls experiment input variables in such a way that the reasons behind changes in output responses can be identified [Dra03, Ch.8].

The factors that influence experiment outcome must be identified and the most influential ones should be accounted for. Factors can be divided into controllable and uncontrollable [Dra03, Ch.8]. Examples of controllable factors are the manufacturer of microarrays, number of biological samples and treatment of samples (e.g.
untreated control samples and samples treated with a drug). Examples of uncontrollable factors are bad quality spots on microarrays, inherent biological variability between organisms and random variation in hybridization.

*Replication* is a technique that allows the researcher to estimate the quantity of experimental error [Dra03, Ch.8]. For example, biological variability can be estimated by repeating the experiment for several individual samples while keeping all other controllable factors constant [Dra03, Ch.8]. Two main types of replication are technical and biological replication. In *technical replication*, the same mRNA is used on multiple microarrays, while in *biological replication*, mRNA is extracted from multiple cases [All06]. Biological replication is considered essential because it provides estimates for both measurement and biological variability [All06]. Technical replication only estimates measurement error [All06].

In *spot replication*, a type of technical replication, multiple copies of a spot are present in the microarray distributed spatially [Dra03, Ch.8]. This provides an estimate for spatial variability on the microarray and for hybridization variability.

In two-channel microarrays experiments, the researcher must decide which two samples are hybridized on the same array. One common "design pattern" for this question is *reference design*, in which one common reference sample is hybridized on the same channel of each array [Dra03, Ch.8]. Case samples $S_1, \ldots, S_n$ are hybridized on the other channels of arrays $A_1, \ldots, A_n$. Reference design is simple to analyze but it has the drawback that it does not take dye effect into account [Dra03, Ch.8]. Also, it measures the least important reference sample $n$ times and each case sample only once [Dra03, Ch.8].

### 4.3 Preprocessing

After image processing is done, the data must be preprocessed before it can be analyzed further [Dra03, Ch.12]. Common preprocessing tasks include normalization, spot combining and dye-effect compensation for two-channel arrays.
4.3.1 Intensity normalization and log ratios

Microarray experiments typically use several microarray chips and the range of expression levels may not be consistent across chips [Dra03, Ch.12]. In particular, mean level may vary between chips. Mean levels can be normalized by dividing each expression value by the mean of the array [Dra03, Ch.12]. Alternatively, a Z transformation can be performed on each array by subtracting the mean and dividing the result by the standard deviation [Dra03, Ch.12].

Expression values are often transformed using the logarithm function [Dra03, Ch.12]. This brings the values to a range that is easier to interpret. For example, the linear range $[1, 2^{16}]$ becomes $[0, 16]$ when base 2 logarithm is applied. Moreover, the resulting distribution is usually approximately normal (Gaussian) in shape [Dra03, Ch.12] and this enables the use of statistical methods that assume a normal distribution.

In two-channel arrays, we often want to compare the two samples that are hybridized on the same array. We obtain the difference between a spot on green channel ($G$) and the corresponding spot on the red channel ($R$) by computing the ratio $G/R$. Depending on experiment setup, we may alternatively use $R/G$. If the log transform is performed before channel comparison, we subtract the logarithm values. This is based on the property $\log(xy) = \log x - \log y$. Resulting values are called log ratios.

4.3.2 Background correction

In background correction, background signal level is subtracted from the spot signal level [Dra03, Ch.12]. Background level can be computed from the immediate local area around the spot or from a larger neighbourhood of the spot [Dra03, Ch.12]. Background computation is done using locations that do not contain probes when such locations are available.

For two-channel arrays, the usefulness of background correction is disputed [Dra03, Qin04, All06]. The assumption behind background correction is that the spot signal is a sum of background signal and actual spot signal [Dra03, Ch.12]. Background signal is caused by DNA fragments that hybridize to a probe whose sequence does not
match the sequence of the DNA fragment. This is called non-specific hybridization. By subtracting the background, we should get a more accurate spot signal. However, there is evidence that signal levels in locations without probes (background) cannot be directly compared to locations that contain probes [Dra03, Ch.12]. DNA is more likely to stick to background substrate than to hybridize non-specifically to a probe [Dra03, Ch.12]. Background correction would then be over-correction.

4.3.3 Combining samples and probes

Replicated probes within a single microarray or between multiple microarrays can be combined into one using mean, median or other central tendency function [Dra03, Ch.12]. This provides a single expression value for each gene and makes further analysis more straightforward. The drawback is that information is lost in the transformation. The loss of information can be alleviated by including additional parameters from the original distribution, such as the number of original values or the standard deviation [Dra03, Ch.12].

4.4 Finding differentially expressed genes

In most microarray experiments, we want to compare the expression of one sample (e.g. healthy control sample) to another (e.g. case sample with disease) [Dra03, Ch.13]. We obtain expression values for a large number of genes and our goal is to find the genes that are differentially expressed between the samples. Using one of the techniques presented in this section, each gene is classified as differentially expressed or non-differentially expressed. Differential expression can be divided into overexpression (the expression value in case is higher than in control) and underexpression.

Two types of errors can occur in classification. If we classify a gene as differentially expressed when it is in reality not, we have made a type 1 error that results in a false positive [Dra03, All06]. A gene that is in reality differentially expressed but is classified as non-differentially expressed is a false negative and is the result of a type
Any classification method must make a compromise between type 1 and type 2 errors: reducing the risk of one increases the risk of the other [Dra03, Ch.5].

4.4.1 Fold change

The simplest method for selecting differentially expressed genes (DEGs) is the fold change method [Dra03, All06]. In this method, the ratios between case and control samples are compared and genes satisfying a given threshold such as two or three are selected. The threshold is typically chosen arbitrarily [Dra03, Ch.13]. When using log ratios as discussed in section 4.3.1, filtering can be done conveniently with the condition $|x_i| \leq \log(T)$, where $x_i$ is a log ratio value and $T$ is the fold change threshold.

Fold change alone is not a sufficient classification method because it does not take sample variance into account and provides no confidence metric that tells how reliable the classification is [All06]. In particular, the false positive error rate cannot be estimated [All06]. The fold change threshold is arbitrary and may be inappropriate for the type of experiment [Dra03, Ch.13]. For example, if a threshold of two is chosen but no gene in the experiment has fold change over two, no genes are classified as differentially expressed. The fold changes of genes depend on the experiment in question so no single threshold is appropriate for all experiments.

4.4.2 Student’s test

Student’s test, or $t$-test, is a statistical test that can be used to estimate whether random samples from two normal distributions have the same mean [Roh03, Sec. 8.7]. The use of $t$-test requires at least two samples from both distributions since it estimates the variance of the distributions by computing sample variances.

In the context of microarray analysis, we can test whether the expression of a probe $P$ in two sample groups is equal by taking the $t$-test over replicate arrays [Dra03]. For example, if we have three control samples $S_1, S_2, S_3$ and three case samples
$S_4, S_5, S_6$, we can compare the mean of $P_1, P_2, P_3$ to the mean of $P_4, P_5, P_6$ using the t-test. Here, $P_i$ denotes the expression value of probe $P$ in array $i$. The t-test produces a probability, or p-value, for obtaining a mean difference at least this extreme by chance assuming the means of the distributions are equal. If the p-value is lower than a threshold $\alpha$, we conclude that the means are different and the probe $P$ corresponds to a differentially expressed gene.

More formally, let $X_1, X_2, \ldots, X_n$ be random samples from normal distribution $N_1$ with mean $\mu_1$. Correspondingly, $Y_1, Y_2, \ldots, Y_m$ are random samples from another normal distribution $N_2$ with mean $\mu_2$. We assume that the variances $\sigma_1^2$ and $\sigma_2^2$ are equal and unknown to us [Roh03, Sec. 8.7]. Let $\overline{X}$ and $\overline{Y}$ be the sample means computed from the data. When testing the null hypothesis $\mu_1 = \mu_2$, the t-test gives the probability of obtaining a sample mean difference at least as extreme as $|\overline{X} - \overline{Y}|$ assuming that the null hypothesis is correct. If the p-value is lower than a $\alpha$, we reject the null hypothesis and conclude that $\mu_1 \neq \mu_2$. We can also use one-sided null hypotheses $\mu_1 \leq \mu_2$ or $\mu_1 \geq \mu_2$.

One assumption for the t-test is the normality of data. One study tested the normality assumption in microarray experiments using four software packages for pre-processing 59 Affymetrix chips [GiK03]. They found strong support for expression data normality, although one software package produced a non-normal distribution for low-expressed genes.

The p-value estimates the probability of making a type 1 (false positive) error [Dra03, Ch. 13]. This provides a clear advantage over the fold change method. The probability of making a type 2 error can be reduced by replicating the experiment a sufficient number of times [Pou06].

However, the t-test also has some complications. It cannot be used if there are no replicates and it is suboptimal if the number of replicates is low, since the sample variances tend to be large [Roh03].
4.4.3 Multiple comparison correction

A significant problem associated with the t-test and other statistical tests is the presence of multiple hypotheses [Dra03, All06, Pou06]. When we have an array with 40000 probes, we test 40000 hypotheses in parallel. If the expression data were taken randomly from a normal distribution and we used a p-value threshold 0.05, we would get $40000 \times 0.05 = 2000$ DEGs, all of which are false positives. The p-value threshold $\alpha$ controls the probability of making a single type 1 error, but when the test is repeated, the risk of making at least one type 1 error is greater [Dra03].

There are various multiple comparison correction methods available [Dra03, Ch.9]. The simplest method is the Bonferroni correction, which divides the p-value threshold $\alpha$ by the number of experiments $N$ [Dra03, Ch.9]. This is equal to multiplying the p-values by $N$ and keeping $\alpha$ unchanged. Bonferroni correction is conservative: it greatly reduces false positives but this results in many false negatives. It is too conservative for use in microarrays [Pou06, Dra03]. Bonferroni correction controls the family-wise error rate (FWER) [Pou06]: the probability of producing at least one false positive.

False discovery rate (FDR) is a measure that estimates the proportion of false positives among the DEGs [Pou06]. FDR based multiple comparison correction methods are suggested to be useful for microarray analysis [Pou06, All06]. They are less stringent than FWER based methods and have less false negatives, while at the same allow to control the false positive rate. Since FDR was introduced in 1995, numerous FDR based correction methods have been developed and the selection of a method requires some care [Pou06].

4.5 Annotation and biological functionality

After DEG analysis produces a list of genes, their biological significance and functionality must be determined. While DEG analysis is a well-researched area in microarray analysis, biological annotation is a topic of current research [DKT07, STM05, TGK05]. Often, DEG analysis produces too many genes for manually as-
sessing the biological functions of the genes. This necessitates the use of further data mining methods, such as clustering where similar genes are grouped into one unit [HaK06].

Gene Ontology (GO) is a well-known database for the biological functionality of genes [ABB00]. GO provides annotation for genes and gene products related to their cellular location, associated biological processes and molecular functionality. For example, GO might provide the information that the proteins associated with a certain gene are located in the nucleus of a cell and are involved in the biological process of cell division.

One way to utilize GO in microarray analysis is to cluster genes based on their GO annotations so that genes sharing similar annotations belong to the same cluster [OLH08a]. The formal structure of the GO database enables to define distance metrics that are used in hierarchical clustering. The analysis produces a set of genes for each cluster and the GO annotations that are shared between all genes in the cluster. The GO based clusters together with expression profiles are visualized using a heat map. An example heat map is shown in Figure 8. The heat map allows to inspect differences in expression patterns inside GO clusters.
Figure 8: Heat map that visualizes GO clustering (the dendrogram on the left) as well as expression values and clustering based on expression profile (top dendrogram). Five GO clusters G1–G5 are shown. Overexpressed genes are shown with white or yellow color and underexpressed genes with red color. Sample labels are displayed on the bottom.
5 Architecture of the framework

We introduce Anduril, a component based workflow framework for microarray analysis and other bioinformatics analyses. The framework consists of multiple hierarchical levels as shown in Figure 9. In this section, we discuss the backbone (level 2) framework, as well as the environment in which Anduril is used and its design goals. The backbone framework is not specific to microarray analysis. In sections 6 and 7, we explain how the backbone framework is specialized to a level 1 microarray framework. The backbone could also be specialized to other types of level 1 frameworks, shown as Framework X in the figure. A case study in section 7 also shows how the microarray framework is specialized to an analysis application. This is shown on the leaf level in the figure.

5.1 Stakeholders and usage environment

Anduril is implemented for Computational Systems Biology Laboratory at Biomedicum Helsinki, University of Helsinki. The stakeholders of the system are:

- The client, usually a biomedicine researcher, initiates the analysis project and

![Figure 9: Framework hierarchy of Anduril. The backbone (level 2) framework implements core functionality that is not related to any particular application area. Level 1 frameworks implement application area specific components. Concrete applications are shown on the leaf level.](image)
provides microarray data for the analyst. The client provides research questions and oversees sample hybridization. The client does not directly use the framework.

- The analyst, usually a bioinformatics researcher, uses Anduril to analyze the data provided by the client. The analyst reports results to the client. It is assumed that the analyst has basic Computer Science skills such as elementary programming and command line usage.

- The framework extender implements new components for Anduril. The roles of analyst and extender are related: the analyst may need to implement a custom component for an analysis, and the extender often acts as an analyst.

The analysis process is often iterative: the client requests further analysis after the analyst provides initial results.

5.2 Design goals

The main purpose of Anduril is to support gene expression microarray analysis. Through its extension mechanisms, the framework implements common microarray analysis functionality that were discussed in section 4. Anduril is also suitable for other bioinformatics and scientific analysis. As an example, functionality for another type of DNA arrays, single nucleotide polymorphism (SNP) arrays [LFG99], has been implemented. Such extensions are beyond the scope of this thesis.

A key design goal is flexibility. When new microarray analysis techniques are developed, they can be integrated into the framework and into analysis pipelines. In our experience, there is often considerable variability between microarray analysis projects and the framework must be flexible enough to support almost any kind of analysis.

Anduril provides a well-defined architecture for analysis projects, i.e., projects reuse the design of the framework. This encourages a systematic approach for microarray analysis.
The framework can be extended using several programming languages. This allows to select to most suitable language for the task in hand. For example, high performance algorithms may be implemented in Java, while R code may be used to take advantage of the Bioconductor library [GCB04] for R that contains functionality for microarray analysis.

One requirement for the framework is productivity: it must be faster to the use Anduril rather than do analysis manually, e.g. with a custom program. Since frameworks typically have a learning period, it is expected that productivity gains are not immediate but occur after the researcher has been acquainted with the framework.

5.3 Existing work

There are a few existing workflow-based frameworks and interactive applications for bioinformatics analysis. Two of the most notable are GenePattern [RLG06] and Taverna [OGA06]. GenePattern follows a client-server architecture where the components are executed on a server and the user interface is implemented using WWW techniques. Component models of GenePattern and Anduril are similar, both based on executable files. Taverna focuses on integrating various third-party web services using SOAP and also supports locally executed scripts. Web services make installation simpler, but remote execution increases network traffic which makes analysis of large data sets challenging. GenePattern provides a ready-made component repository for microarray analysis, while Taverna does not. Compared to both GenePattern and Taverna, Anduril provides advancements in the workflow engine, such as optimized re-execution of component networks. In Table 2, the features of Anduril, GenePattern and Taverna are compared.

5.4 Architecture overview

The backbone framework is based on components connected to form a network. A component is an executable that reads files as input and writes files as output.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Anduril</th>
<th>GenePattern</th>
<th>Taverna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application type</td>
<td>local executable</td>
<td>client/server</td>
<td>local executable</td>
</tr>
<tr>
<td>Component model</td>
<td>executable</td>
<td>executable</td>
<td>web service/script</td>
</tr>
<tr>
<td>Graphical user interface</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Emphasis on local/remote components</td>
<td>local</td>
<td>local</td>
<td>remote</td>
</tr>
<tr>
<td>Extension language</td>
<td>any language</td>
<td>any language</td>
<td>Java, R, SOAP</td>
</tr>
<tr>
<td>Language mini-frameworks</td>
<td>Java, Matlab,</td>
<td>Java, Matlab, R</td>
<td></td>
</tr>
<tr>
<td>Command line/SSH operation</td>
<td>full</td>
<td>no</td>
<td>running existing</td>
</tr>
<tr>
<td>Level of integration between components</td>
<td>high</td>
<td>high</td>
<td>workflows</td>
</tr>
<tr>
<td>Test support for components</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Component type system</td>
<td>subtyping and</td>
<td>basic</td>
<td>basic</td>
</tr>
<tr>
<td></td>
<td>generics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrating into other applications</td>
<td>Java API</td>
<td>remote call</td>
<td>Java API</td>
</tr>
<tr>
<td>Visualization of results</td>
<td>PDF, CSV files</td>
<td>interactive</td>
<td>interactive</td>
</tr>
<tr>
<td>Distributed equipped for microarray analysis</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Final report generation</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Workflow visualization</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Nested workflows</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Conditional branches</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Optimized re-execution of workflows</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Parallelization</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Pausing workflows</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Static workflow configuration error checking</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2: Main feature comparison between Anduril, GenePattern and Taverna.

Components have a number of input and output *ports*; each port represents a file or a directory. An input port may be optional, in which case its input file may be missing. In addition to file inputs, components have simple parameters such as integers and strings. The ports and simple parameters have types associated with them. The type system facilitates static error checking and interface documentation. The set of port types is extensible, while the set of simple parameter types is fixed.
A component can be considered as a function $f(x_1, \ldots, x_m) \rightarrow (y_1, \ldots, y_n)$. The interfaces of components are documented in an uniform manner, which allows users to consider components as black boxes.

Components can be reused between analysis projects, which makes components a mechanism for code reuse. It is also possible to write project-specific components that are used only in one analysis. Level 1 and 2 frameworks provide default components that can be used in analysis applications. The backbone framework provides a few core default components, but most default components are provided by level 1 frameworks.

Components can be written in any language since the only requirement is the ability to read and write files. However, in order to make writing components as easy as possible, language specific mini-frameworks are available for the most common languages such as Java, R and Perl.

Concrete applications are created by wiring selected components into a directed network where the output port of a component is connected to the input port of another. An example network is shown in Figure 10. Since ports are typed, Anduril is able to statically detect type mismatches in port connections. Static error checking is advantageous when the analysis run times are several hours.

![Example network](image)

Figure 10: Example network. For each node, the name of the instance (e.g. in1) as well as the name of the component (e.g. IN) are shown. There can be several instances of the same component. An output port may be connected to multiple input ports, but an input port may have at most one connection. Here, some output ports are unconnected. The second input port of A is optional and no connection for it has been specified. Names and types of ports are omitted for clarity.
Network topology is configured using a high-level domain specific language (DSL) that resembles programming languages such as Java. The advantage of a DSL is that it is possible to use Anduril without writing code in an actual programming language.

The network is executed by launching components in any order permitted by port dependencies. Components that do not depend on each other are executed in parallel. This allows to take advantage of multi-core CPUs without requiring the user to take care of details such as process creation and synchronization. Depending on network topology and the number of CPU cores, parallel execution can significantly reduce run times.

All intermediate and final results of components are stored on disk. Also, configuration settings of all components are stored. When the same network is executed again, the execution engine can infer which components have changed in configuration. Only changed components and those that depend on changed components are executed. Partial re-execution of the network supports the iterative nature of the microarray analysis process. When the client requests an analysis run with different parameters, the analyst makes changes to a few components and only those components that are affected by the changes are executed. Partial re-execution is transparent to the user.

5.5 Component model

As mentioned in section 2, components have requirements for the environment in which they operate and frameworks provide a standard environment for components. This section describes the component environment defined by Anduril.

Regular components are executed by creating a new process. For example, a regular Java component executes in a different virtual machine (VM) than the engine, which is also written in Java. Special internal components written in Java are executed inside the same VM as the engine and have access to internal data structures. This allows to write components that modify execution logic. For example, a pause
component halts network execution when it is invoked.

5.5.1 Component interface definition

A component is used through its external interface. The interface consists of input and output ports, simple parameters and various attributes such as name and version. An important part of the interface is documentation. All aspects of the interface must be documented in order to make the component usable. The interface is specified with an XML file called the descriptor file.

An example descriptor file is shown in Figure 11. The example component, AddMatrix, takes two or three numeric matrices as input and writes their sum as output. One of the input ports (m3) is optional, i.e., the input file may be missing when the component is invoked. A numeric bias, set by the simple parameter bias, is added to the output matrix. Bias has a default value of 0. Legal types for simple parameters are boolean, float, int and string. The data type of the ports, NumMatrix, is defined elsewhere.

The default value of a simple parameter is passed to the component when the component is invoked. Hence, the default value is specified only in the descriptor file and is not embedded into the source code of the component. This has the advantage of avoiding redundancy problems. When the default value is changed, the change is done only to one place.

Components may have one or more category tags that help to maintain a large repository of components. The requires element gives a free-format description of the external requirements of the component. In the case, AddMatrix requires an installation of R.

The launcher element defines how the component is invoked. The engine provides launchers for programming languages (currently R and Java) as well as a generic Bash script launcher. Launchers are configured with named arguments. In the example, AddMatrix is executed by the R launcher by invoking the R script AddMatrix.r. Java components are invoked using the Java launcher, which takes the name of the
Figure 11: Example descriptor XML file.

class as argument.

Anduril creates HTML manual pages for all components based on descriptor files. This provides uniform documentation for components written in different languages.

5.5.2 Component execution

When the component is invoked, it needs to know file names for input and output ports as well as simple parameter values. These are supplied by the engine via a command file, a simple properties file. The command file is passed as a command line argument. The command file also contains metadata such as the name of the component instance in a network. Instance name is needed when there are several instances of a component and they need to create files with unique file names.
An example command file for AddMatrix is shown in Figure 12. In this instantiation, the optional third matrix is missing and the bias parameter is set to unity.

In addition to regular output ports, there are two special output ports, error stream (\_errors) and log message stream (\_log). They are used to pass error and log messages from the component to the engine. The engine displays them to the user. Special ports, like regular ports, also have a data type that defines the structure of the result files.

The component process signals exit status by returning a zero status on success and non-zero on failure. Anduril defines standard exit statuses for various error conditions. The error stream gives more detailed information on the error condition. Component execution is successful if the exit status is zero and the error stream is empty.

5.5.3 Data types

Each port has a data type that is used for type checking in component connections and interface documentation. The set of data types is extensible. The backbone framework defines a few core data types, such as the types for error and log streams, but most data types are defined by level 1 frameworks. Commonly, components in the same application area share data types and the components and associated data types are defined in the same resource.

Data types are defined with XML files. Data types do not necessarily have executable code associated with them on the framework level; the XML files mainly
contain documentation. Data types are described in sufficient detail to enable component writers to use them.

An example XML fragment is shown in Figure 13. The fragment is part of a larger XML file that defines several data types. The example defines the type NumMatrix that is used by AddMatrix. Both the logical purpose (desc) and physical layout (presentation-desc) are described. NumMatrix has one example file (example1.csv, not shown) that provides a concrete instance of the data type. In general, data types may have any number of example files.

The physical representation, given by presentation-type, may be a single file (as is the case with NumMatrix) or a directory. The data type may optionally have an associated file extension, such as .csv for NumMatrix. The extension makes it possible for the operating system to launch a spreadsheet program to edit the file, for instance, when the file is activated by the user.

The type system supports single inheritance, i.e., each data type may have one parent type. For NumMatrix, the parent type is CSV. If type A is the parent type of B, each file of type B is also a valid byte-level representation of type A. The type hierarchy is used in type checking; see section 5.6 for details. It is not mandatory to have a parent type, which implies that there is no single root type that would be the grandparent of all types. Rather, the type hierarchies form a forest. There is little need for a root type since types do not have associated functionality like classes do.
in object-oriented languages.

5.5.4 Language-specific mini-frameworks

In principle, any programming language can be used to implement components. However, writing components is faster and less error-prone if parts of components can be reused. To this end, there are mini-frameworks for most commonly used languages that provide common functionality for components. Mini-frameworks provide a parser for the command file and convenient write access to error and log streams. If the format of the command file would be changed, only mini-frameworks would need to be changed.

Mini-frameworks may also provide read and write support for data types. Data types may need specific data structures that allow to access the data. The NumMatrix type could be implemented as a `double[][]` array in Java, but a more complex data type might need a set of classes to represent the data. The choice of data structure depends on the language. The natural representation in a functional language is different from an object-oriented language.

Since the set of types is extensible by third parties, a single mini-framework can not support all data types. Rather, the level 1 framework that defines a set of common data types will also provide I/O support for the data types in selected languages. It is a design goal the use data types that are easy to parse; for example, we have favoured CSV over XML. Since CSV files can often be read with little effort, some data types may not need explicit support by the framework. If support for all data types would be implemented in all supported languages, the amount of work would be proportional to $T \times L$, where $T$ is the number of data types and $L$ is the number of languages. Designing simple to parse data types reduces $T$ and focusing the mini-frameworks only for specific languages reduces $L$. 
5.5.5 Example: adding matrices using R

Figure 14 contains R source code for the example component, AddMatrix. The mini-framework for R is called componentSkeleton (line 1). The main function (line 12) is defined by the mini-framework, as are all other functions in the source code. The main function parses the command file that is passed to the process as command line argument and calls the execute function with the command file structure as parameter. Input matrices are read with NumMatrix.read (lines 3, 4 and 7); input file names associated to each port are given by get.input. On line 9, the bias parameter is added to the result matrix. The result matrix is written to the output port on line 10.

5.5.6 Generic components

A generic component has one or more type parameters. A type parameter is a placeholder for a type that is fixed when the component is used in a network. The type parameter can be used for input and output ports in place of a concrete type. The mechanism is similar to generics in Java. An example of a generic component is one that copies a file from one location to another. It does not care about the type of the file; hence, the type is parametrized.

A type parameter may optionally specify that the actual type must be a subtype of a given type. For example, T extends A denotes a type parameter T whose
corresponding actual type must be a subtype of $A$. Subtype constraints are useful for filter components that modify the contents of the input file but preserve the type. Usually, the filter must assume some properties about the contents of the file. This is represented by the constraint $T \text{ extends } A$.

Type parameters increase the flexibility of components but may also steepen the learning curve for users who have not worked with generics before. However, the actual types for type parameters can usually be assigned by an inference algorithm introduced in section 5.6.3, so much of the generics machinery works transparently to the user.

5.5.7 Delegates

Components do not normally invoke other components; rather, they communicate through files on the network level. However, a delegate mechanism is provided for components that need flexibility that can not be provided by simple parameters. A delegate is a component $D$ that is given to another component $C$ as parameter. Now, $C$ can invoke $D$ during its execution and use the results of $D$ for its own processing. The delegate mechanism is similar to passing functions as first-class arguments in functional programming.

A component $C$ that uses delegates specifies the name and interface of delegate parameters in the descriptor file. When the component is used in a network, the delegate parameter is assigned a value $D$. When $C$ is executed, the engine provides a script for $C$ that is used to execute $D$. Currently, the delegate mechanism requires the Bash shell.

5.5.8 Comparison to CORBA

In comparison to CORBA, both component models are programming language independent and external interfaces of components are specified using a platform-independent language. Data types are an integral part of component interfaces in both models. In CORBA, the interfaces have finer granularity and this is also re-
lected in the type system. Types are specified as composites of atomic types whereas in Anduril, types are specified on file format level.

A major difference between the models is how the components are executed and whether they interact with each other. CORBA is based on server processes and components that communicate with each other. In Anduril, components are executed as distinct processes and do not directly communicate with each other, with the exception of delegate components. The framework model is simpler and requires less supporting infrastructure, but implementing distributed systems is more difficult. The coarser granularity may limit the flexibility of component interfaces, but it also makes the interfaces simpler and easier to use.

5.6 Network topology

An application is created by wiring selected components together into a network using a custom domain specific language. Network configuration includes connections between components (topology) and values for simple parameters of components. In the network, connection $A \rightarrow B$ implies that component $A$ must be executed before $B$. Network configuration is a mechanism for black box reuse.

An example network configuration file is shown in Figure 15. The network is visualized in Figure 16. The visualization is generated automatically by an internal component, ConfigurationReport.

The syntax of the language resembles common languages such as Java or Pascal. The language is strongly and statically typed [Sco99], i.e., types of all names are known before the network is executed and type errors are caught. Unlike Java, types are not explicitly declared but are inferred from the context.

A component instance, i.e., a node in the network, is created with

$$\text{inst} = \text{Component}(\text{port1}=x_1, \ldots, \text{portN}=x_N, \text{param1}=y_1, \ldots, \text{paramM}=y_M).$$

Values for input ports and parameters are given with a unified syntax. For input ports, the name may be omitted. Port values $x_i$ refer to output ports of other component instances. Output ports of $\text{inst}$ are accessed using $\text{inst}.\text{port}$. If the
component has only one output port, the port name may be omitted. A component can be considered as a function that returns a record with named fields that represent output ports. The effect of the function call is not to execute the component; at this stage, only the network topology is created and the network is executed later. Javadoc-like comments (/** */) describe the component instances and are used to
\begin{verbatim}
x = 5
p = someInstance.port
if x*2 > 5.3 { myInstance = ComponentA(p, param=x-5) }
else { myInstance = ComponentB(p, param=(x*8)+1) }
\end{verbatim}

Figure 17: Using named constants and if-statements in a network configuration file.

create a report for the network configuration.

The example network utilizes a number of components, some provided by the backbone framework (INPUT, OUTPUT) and some by a test framework that is used to test the execution engine and demonstrate the use of Anduril. The components in the test framework are related to arithmetic operations between matrices and scalars. The previously seen component, AddMatrix, is part of the test framework.

The INPUT component, provided by the backbone framework, is used to import data files into the network. It has a single string parameter, path. The OUTPUT component is used to gather final results into an output directory where they are easily found. INPUT and OUTPUT are examples of generic components: they can work with any type of files.

The example network computes the sum of two scalars (\texttt{in1} and \texttt{in2}) and the matrix sum of two matrices (\texttt{m1} and \texttt{m2}). The MatrixStats component extracts various statistics about the input matrix, such as the maximum value. The final result of the network is \((\texttt{in1} + \texttt{in2}) \times \texttt{in2} \times \text{maximum-of-sum-matrix}\).

The language also includes if-statements and common arithmetic and comparison operators. If-statements can be used to create parametrized networks. They can only refer to simple values (Booleans, numbers and strings) and not dynamic component results. Named constants can be used to store values that are used for multiple components. Figure 17 demonstrates these features.

5.6.1 Composite components

A composite component is a network of components \(C_i\) that functions as a single component. Each component \(C_i\) implements a part of the composite component.
function AddMatrices(NumMatrix m1, NumMatrix m2, float bias=0)
  -> (NumMatrix out1, NumMatrix out2)
  {
    doubled1 = AddMatrix(m1, m1)
    doubled2 = AddMatrix(m2, m2)
    result1 = AddMatrix(doubled1, m2, bias=bias)
    result2 = AddMatrix(m1, doubled2, bias=bias)
    return record(out1=result1, out2=result2)
  }

z = AddMatrices(someMatrix1, someMatrix2, bias=2.5)

Figure 18: Declaring and invoking a composite component.

This corresponds to the Composite pattern in the Gang of Four book [Gam95]. Composite components allow to reuse network topologies between analysis projects, much like function libraries in traditional programming languages. Further, a large network can be broken into more manageable subnetworks, each of which is a component in the main network.

The syntax for declaring a composite component is illustrated in Figure 18. The composite component is a function that, like other components, has a number of input and output ports and simple parameters. The component AddMatrices computes \((2m_1 + m_2) + \text{bias}\) and \((m_1 + 2m_2) + \text{bias}\). It is used like any other component. The function may invoke other functions, but not recursively. The expression `record(out1=result1, out2=result)` creates a new record with two named fields. Functions use lexical scoping [Sco99], i.e., names defined in the parent block are visible inside the function body. A name defined in the function block overrides the name defined in the parent block.

5.6.2 Conditional branching

Conditional branch components are used to dynamically select alternative execution paths in the network. They correspond to case statements in programming languages. In contrast to if-statements, conditional branches are evaluated dynamically. A conditional branch component is associated with two or more optional branches and it can enable each branch independently. At least one branch must
compareResult = Compare(x, y)
join = switch compareResult {
    case equal = Comp1(in1, in2)
    case less = Comp2(in1, in2)
    case greater = Comp3(in1, in2)
    return Xor(equal.result, less.result, greater.result)
}

Figure 19: Using a conditional branch component. The Compare component has three associated choices, less, equal and greater. Each choice has one associated component that is executed when the choice is selected. Composite components may, of course, be used. The joining Xor component has three optional input ports and it selects the first input that is defined. In this case, at most one input is available.

be enabled. Each conditional branch has a unique associated join component that joins the branches.

The descriptor file of a conditional branch component specifies two or more named choices. For example, a component that compares two numbers would have choices less, equal and greater. The component has a special output port that specifies one or more choices. In the case of the comparator component, only one choice is present at a time, but in general, there may be several. Branches that are not present in the choices file are disabled in the network. In the network configuration file, branch components are used with a syntax demonstrated in Figure 19.

5.6.3 Type checking and type parameter inference

The engine checks types of all component connections for correctness. Let us consider a connection between ports $P_1$ and $P_2$ having types $T_1$ and $T_2$ respectively. The type of the connection is the most specific type of $\text{ANC}(T_1) \cap \text{ANC}(T_2)$, where $\text{ANC}(T)$ denotes the ancestor types of $T$, including $T$ itself. If the intersection is empty, $T_1$ and $T_2$ are incompatible and the engine signals a type error.

The engine infers actual types for type parameters using Algorithm 1. Inference is done based on the connections of generic components. The algorithm iteratively computes the set of possible types $\text{POS}(C_i, T_i)$ for each type parameter $T_i$ of each generic component $C_i$. Initially, $\text{POS}(C_i, T_i)$ is the set of all types, or the descendants
of type $A$ if $T_i$ must extend $A$.

Given an incoming edge whose destination port has the type $T_i$, $\text{POS}(C_i, T_i)$ is reduced using the type of the source port. If the source port is generic, the updated $\text{POS}(C_i, T_i)$ is $\text{POS}(C_i, T_i) \cap \text{POS}(C_j, T_j)$, where $\text{POS}(C_j, T_j)$ is the set of possible types for the source component. If the source port is not generic, the updated $\text{POS}(C_i, T_i)$ is $\text{POS}(C_i, T_i) \cap \text{DESC}(T)$, where $T$ is the actual type of the source port and $\text{DESC}(T)$ is the set of descendant types of $T$. Similar processing is done for outgoing edges of $C_i$. The process is continued until no $\text{POS}$ set changes. The algorithm halts because the initial sets are finite and their size is reduced in each step. Finally, the actual type corresponding to each $T_i$ is the most general type of $\text{POS}(C_i, T_i)$. If the $\text{POS}$ set is empty, the engine signals an error.

5.7 Network execution

When the network is executed, components write output files into an execution directory. The configuration settings are also stored under the execution directory. Each component produces a configuration digest, a string that describes the configuration settings for the component. On subsequent network execution runs, the component is re-executed automatically only if the configuration digest changes. The user can also force re-execution with a command line switch even if the digest is unchanged.

The format for the configuration digest depends on the component, but for regular components, the following information is included: component version; source component name and source port name for each input connection; and parameter values. For instances of the INPUT component that import external files into the network, the timestamp of the file is included in the digest.

Configuration digests provide the mechanism for automated partial re-execution of the network. A consequence of partial re-execution is that the network is restricted to acyclic networks. In our architecture, each output port must have a unique file or directory that represents the contents of the port. This means that each component instance may be executed at most once. Therefore, the network topology
Algorithm 1 Algorithm for type parameter inference.

let $\text{DESC}(T)$ = set of descendant types of $T$, including $T$ itself

let $\text{POS}(C_i, T_i)$ = set of possible actual types for type parameter $T_i$ of component $C_i$

for all component instances $C_i$ do
  for all type parameters $T_i$ of $C_i$ do
    if $T_i$ must extend type $A$ then
      $\text{POS}(C_i, T_i) \leftarrow \text{DESC}(A)$
    else
      $\text{POS}(C_i, T_i) \leftarrow$ the set of all types
    end if
  end for
end for

progress $\leftarrow$ true
while progress do
  progress $\leftarrow$ false
  for all component instances $C_i$ that have type parameters do
    for all incoming edges $e$ whose destination port is generic do
      if source port is generic then
        $C_j \leftarrow$ source component
        $T_j \leftarrow$ type parameter of source port
        $\text{POS}(C_i, T_i) \leftarrow \text{POS}(C_i, T_i) \cap \text{POS}(C_j, T_j)$
      else
        $T \leftarrow$ type of source port
        $\text{POS}(C_i, T_i) \leftarrow \text{POS}(C_i, T_i) \cap \text{DESC}(T)$
      end if
      if $\text{POS}(C_i, T_i)$ was changed then
        progress $\leftarrow$ true
      end if
    end for
  for all outgoing edges $e$ whose source port is generic do
    repeat processing with source and destination swapped
  end for
end for
end while
for all component instances $C_i$ do
  for all type parameters $T_i$ of $C_i$ do
    if $\text{POS}(C_i, T_i) = \emptyset$ then
      signal error
    else
      assign type for $T_i$ as the most general type in $\text{POS}(C_i, T_i)$
    end if
  end for
end for
is not Turing-complete, but obviously individual components may perform arbitrary processing which makes the network as a whole Turing-complete. Although the restriction to acyclic networks reduces the computational expressiveness of the network topology, it has the advantage that network execution always halts if all individual components halt.

5.7.1 Network execution algorithm

In the network execution algorithm, each component instance $C_i$ has a state $\text{State}(C_i) \in \{\text{NOT\_READY}, \text{READY}, \text{FINISHED}, \text{ERROR}, \text{SKIPPED}\}$ that indicates whether $C_i$ has been executed. Unfinished components also have a count indicator $\text{Count}(C_i)$ that tells how many components must be finished before $C_i$ is ready for execution.

The meaning of states are as follows.

NOT\_READY: $C_i$ has not been executed and depends on another component instance $C_j$ that has not been executed. In other words, $C_i$ needs an input file from $C_j$ that is not yet available. Implies that $\text{Count}(C_i) > 0$.

READY: All inputs of $C_i$ are available but the component has not been executed yet. Implies that $\text{Count}(C_i) = 0$.

FINISHED: $C_i$ has been executed successfully.

ERROR: $C_i$ has been executed but execution failed. Output files of $C_i$ are not available.

SKIPPED: $C_i$ has been skipped. The output files of $C_i$ are not available, but the execution status of $C_i$ is considered successful. Components $C_j$ that depend on $C_i$ can be executed if $C_i$ is connected to an optional port of $C_j$. Effectively, the connection is removed before $C_j$ is executed. This state is needed by conditional branch components and the manual pause component.
The network is executed using Algorithm 2. In the first phase, the states are initialized (lines 2–8). If a state from a previous run is not available, the initial state is NOT_READY. Otherwise, the previous state is used as the initial value. The ERROR state is converted to NOT_READY, since we try to execute the failed component again. If the configuration digest of a component $C_i$ is changed, the states of $C_i$ and all components that depend on $C_i$ are changed to NOT_READY; this is not shown in the pseudo-code.

In the second phase, counts are initialized (lines 9–19). Each component instance $C_j$ that provides input for $C_i$ increases the count of $C_i$ by one. If $C_i$ has no unfinished input components and its current state is NOT_READY, its state is changed to READY. In particular, components that have no input ports can always be executed.

The actual execution phase is on lines 21–37. Component instances $C_i$ that have $\text{State}(C_i) = \text{READY}$ are selected for execution one at a time. Component execution (lines 24–36) is done in parallel. Components are executed using a maximum of $T$ threads; $T$ is configurable by the user. Component execution does not block if there are less than $T$ threads currently active and blocks otherwise. If the execution of $C_i$ is successful, the counts of other component instances $C_j$ that depend on $C_i$ are decremented. When $\text{Count}(C_j)$ reaches 0, $C_j$ is inserted into the set of ready-to-execute components.

5.8 Support for black box testing

Anduril provides support for component-level and network-level black box testing. Component tests are composed of input files corresponding to each input port and expected result files corresponding to each output port. Also, simple parameters can be provided. Success is determined by executing the component and comparing actual results to expected results. Some output ports may be omitted in testing if it is difficult to determine success by byte-level comparison. For example, generated binary files that contain time stamps are difficult to compare for equality. A test
Algorithm 2 Algorithm for network execution.

1: // Initialization of State and Count
2: for all component instances $C_i$ do
3:     if state $S$ from previous execution is available and $S \neq ERROR$ then
4:         State($C_i$) ← $S$
5:     else
6:         State($C_i$) ← NOT_READY
7:     end if
8: end for
9: for all component instances $C_i$ do
10:    Count($C_i$) ← 0
11:    for all $C_j$ that provide input for $C_i$ do
12:        if State($C_i$) ∉ \{FINISHED, SKIPPED\} then
13:            Count($C_i$) ← Count($C_i$) + 1
14:        end if
15:    end for
16:    if Count($C_i$) = 0 and State($C_i$) = NOT_READY then
17:        State($C_i$) ← READY
18:    end if
19: end for
20: // Network execution
21: ReadySet ← \{ $C_i$ | State($C_i$) = READY \}
22: while ReadySet ≠ ∅ do
23:    $C_i$ ← PopItem(ReadySet)
24:    execute $C_i$
25:    if $C_i$ had errors then
26:        State($C_i$) ← ERROR
27:    else
28:        State($C_i$) ← FINISHED
29:        for all $C_j$ that directly depend on $C_i$ do
30:            Count($C_j$) ← Count($C_j$) - 1
31:        end for
32:        if Count($C_j$) = 0 then
33:            State($C_j$) ← READY
34:            ReadySet ← ReadySet ∪ \{ $C_j$ \}
35:        end if
36:    end if
37: end while
case may also expect failed execution. Components may have any number of test cases and the engine can execute the tests for all components at once or only for one selected component.

Network tests are composed of a network topology configuration file, input files for the network and expected final result files. Like component tests, a network test case may also expect failed execution.
6 Microarray analysis framework

The microarray framework provides components and data types related to microarray analysis. Also included are components for automated PDF report generation. The reports are used for communication to the client and describe both the configuration of the network and the results.

We first develop a mathematical formulation of microarray analysis that is used as the basis for components. Most components are not introduced in detail since there are over 50 of them. The most often used components are demonstrated in a case study in section 7. Also, some relevant components that are not used in the case study are introduced in this section.

6.1 Mathematical formulation of microarray experiments

Microarray experiments consist of a number of biological samples and their measured expression values. Typically, samples are combined into sample groups such as case and control samples. Each sample (or sample group) has an associated set of genes that are measured for the sample. The analysis produces gene subsets of particular interest, such as differentially expressed genes. In addition to expression values for each gene, we are often interested in other attributes of genes, such as their human-readable descriptions for reporting purposes. Here, we define a mathematical formulation of microarray experiments that is based on sample groups and their transformations, expression values, gene sets and their transformations, and gene annotations. The formulation provides a solid foundation for the actual components.

Let $G = \{g_1, g_2, \ldots, g_n\}$ be the set of all sample groups. The set $G$ is composed of individual samples and derived sample groups. The function $t : G \rightarrow \{\text{channel, logratio}\}$ gives the type of the group. Channel groups are individual samples or median transformations of channel groups. Log ratio groups are formed by taking the log ratio of two channel groups.

Given group $g$, $L_g$ is the set ("list") of all IDs associated to $g$. The IDs represent
genes or other genomic sequences. There can also exist other ID sets $L_{g,x} \subset L_g$ associated to $g$, called ID subsets. They mark interesting IDs among the ID set, such as differentially expressed genes.

There are a number of biological databases that store information on genomic sequences. Each database represents a namespace. Each ID set belongs to a specific ID namespace, denoted by $s(g)$. Two IDs are equal if they belong to the same namespace and their string representations are equal. Examples of namespaces are the namespace of a specific Agilent microarray chip and the namespace of Ensembl gene IDs [HBB02].

The function $e_g : L_g \rightarrow \mathbb{R}$ gives expression values for the IDs of $g$. The values are in base 2 logarithm scale. The function $w_g : L_g \rightarrow [0, 1]$ gives corresponding weights. The weight represents the quality of the probe associated to the expression value. Notice that $e_g$ and $w_g$ also apply for ID subsets $L_{g,x}$ since $L_{g,x}$ are subsets of $L_g$.

### 6.1.1 Transformations between groups

The following transformations between sample groups are defined.

**Ratio transformation** The ratio transformation denoted by $g_1/g_2 = g$ creates a group whose expression values are ratios between two other groups. Since expression values are stored as logarithms, the ratio is computed using $\log(x) - \log(y) = \log(x/y)$. When $g$ is the new group, $e_g(id) = e_{g_1}(id) - e_{g_2}(id)$ and $w_g(id) = \min\{w_{g_1}(id), w_{g_2}(id)\}$. For weight, the minimum function is used because the quality of the derived expression value depends on the quality of both source values. If one of them is low, then the quality of the derived value is also low.

Source groups must be of the same type, i.e., $t(g_1) = t(g_2)$. It then holds that $t(g) = \logratio$. The ID set of $g$ is $L_g = L_{g_1} \cap L_{g_2}$. ID namespaces of $g_1$ and $g_2$ must be equal; $g$ then inherits the namespace.
**Median transformation**  The median transformation \( M(g_1, g_2, \ldots, g_n) = g \) creates a group by taking the median of expression values. When \( g \) is the new group, \( e_g \) is the weighted median of \( e_{w_1}, \ldots, e_{w_n} \) and \( w_g \) is the median of \( w_{g_1}, \ldots, w_{g_n} \). For simplicity, the transformation is called the median transformation, but in place of the median function some other central tendency function, such as arithmetic mean, could be used. All source groups must be of the same type, i.e., \( t(g_1) = t(g_2) = \ldots = t(g_n) \). The target group is then of the same type. The ID set of \( g \) is \( L_g = L_{g_1} \cap L_{g_2} \cap \ldots \cap L_{g_n} \). ID namespaces of \( (g_1, g_2, \ldots, g_n) \) must all be equal; \( g \) then inherits the namespace.

**Arbitrary transformation**  A transformation of the form

\[
A(g_1, g_2, \ldots, g_n) = (g'_1, g'_2, \ldots, g'_n)
\]

takes \( n \) groups, mutates their attributes and produces \( n \) new groups. The transformation may mutate expression and weight values as well as ID sets, group types and ID namespaces. The definition presented here is not for a single transformation but a class of transformations. The exact method of creating new groups depends on the transformation in question. Commonly, arbitrary transformations are used for normalization and other preprocessing in the early stages of the analysis network. If \( A \) only mutates expression and weight values but leaves ID sets, group types and ID namespaces intact, it is called a property-preserving transformation.

### 6.1.2 Properties of group transformations

Repeated transformations form a chain \( g_1 \to g_2 \to \ldots \to g_n \). Transformations must not be recursive; each group must appear at most once in a chain. Chains can be visualized with a directed acyclic graph (DAG). An example is seen in Figure 20. For ratio and median transformations, the graph has the edge \( (g_1, g_2) \) if \( g_1 \) is used as a parameter in the transformation that produces \( g_2 \). For arbitrary transformations, the graph has a node \( A \) that represents the transformation, and edges \( (g_i, A) \) and \( (A, g'_i) \) for all \( i \). The graph has no cycles because recursive transformations are not
Let us consider chains where only ratio, median and property-preserving arbitrary transformations are used. It is easy to see that once the ratio transformation is used, all subsequent groups have the type logratio. If only median transformations are used, all groups have the type channel. Therefore, the sequence \((t(g_1), t(g_2), \ldots, t(g_n))\) has the form \((\text{channel}^+, \text{logratio}^*)\).

In a transformation chain, ID sets form a superset sequence: \(L_{g_1} \supset L_{g_2} \supset \ldots \supset L_{g_n}\). If all individual samples \(i\) have the same ID set, i.e., \(L_i = L\) for all \(i\), then all derived groups \(g\) have \(L_g = L\).

### 6.1.3 Derived group as a function of its ancestors

When given a derived group \(g\) and its expression values \(e_g\), we can express \(e_g\) as a formula that uses expression values from more primitive groups. When \(g\) is a node in the DAG for transformations, \(e_g\) may depend on all ancestors of \(g\). When only ratio and median transformations are considered, the exact formula can be written using the definitions of said transformations. This is done be repeatedly expanding the group using its immediate ancestors.

Arbitrary transformations are essentially black boxes and create a "visibility barrier" into the DAG so that the formula can not be expanded further when an arbitrary transformation is encountered. This somewhat limits the utility of the method, but if arbitrary transformations are used near the leaf nodes, expansion can still done for the other parts of the DAG.

In the example network in Figure 20, \(C\) can be written as \(M(R_1, R_2) = M(NS_1/NS_2, NS_3/NS_4)\). Expansion can not continue further because \(A_1\) and \(A_2\) act as barriers. Expression values of \(C\) are \(e_C(id) = \text{median}(e_{NS_1}(id) - e_{NS_2}(id), e_{NS_3}(id) - e_{NS_4}(id))\). Weights of \(C\) are computed in a similar manner.
Figure 20: Example directed acyclic network that illustrates an experiment with four samples on two two-channel chips. Cases (S1, S3) are on the green channel and controls (S2, S4) on the red channel. Transformations A1 and A2 perform within-chip normalization. After this, normalized values on each chip are combined into log ratios. Finally, the median transformation C combines both log ratio groups. The type (channel or logratio) of each group is written next to the group.

6.1.4 Transformations between ID sets

Transformations between ID sets have the form $T(L_{g_T}, L_{g_1,x_1}, \ldots, L_{g_n,x_n}) = L_{g_T,x}$. Groups $g_i$ are called source groups and group $g_T$ is called the target group. Target and source groups could be all distinct, all equal or any combination of them. Since all sets $L_{g_i,x_i}$ are ID subsets, it holds that $L_{g_i,x_i} \subset L_{g_i}$. In particular, $L_{g_T,x} \subset L_{g_T}$. A transformation that would not satisfy the latter condition would not be a valid transformation. The transformation may need $L_{g_T}$ as a parameter in order to be able to produce valid ID sets.

Filter A function of the form $F(L_{g,x}) = L_{g,y}$, where $L_{g,y} \subset L_{g,x}$, is called a filter function. It operates within a single group and produces a set of IDs that is a subset of the IDs given as parameter. Notice that F is not a single function but a class of functions. Filtering criteria depends on the filter function in question.
Set difference  The function $\text{DIFF}(L_{g_1,x}, L_{g_2,y}) = L_{g_1,x} \setminus L_{g_2,y}$ removes those IDs from $L_{g_1,x}$ that are present in $L_{g_2,y}$. The target group is $g_1$. Figure 21 illustrates the DIFF function.

Generalized union and intersection  The function

$$\text{UI}(k, L_{g_T}, L_{g_1,x_1}, \ldots, L_{g_n,x_n}) = L$$

gives a set of those IDs that are present in at least $k$ sets out of $L_{g_1,x_1}, \ldots, L_{g_n,x_n}$ and belong to $L_{g_T}$. The parameter $k$ is between 1 and $n$ inclusive. When $k = 1$, the function has semantics similar to the union function:

$$L = (L_{g_1,x_1} \cup \ldots \cup L_{g_n,x_n}) \cap L_{g_T}.$$  

When $k = n$, UI is the intersection function:

$$L = L_{g_1,x_1} \cap \ldots \cap L_{g_n,x_n} \cap L_{g_T}.$$  

When $k$ is between the extremes, UI implements a $k$-out-of-$n$ function. For example, when $n = 3$ and $k = 2$, UI returns those IDs that are present in at least 2 sets (in addition to $L_{g_T}$). Figure 22 illustrates the UI function.

6.1.5 Annotation functions

An annotation function provides attributes for gene IDs. Basic examples are gene descriptions and the list of biological pathways the gene is involved in. Annotations
Figure 22: Venn diagrams for generalized union and intersection when \( n = 3 \) and \( k \) varies from 1 to 3. For simplicity it is assumed that \( L_g = L_{g_1,x_1} \cup L_{g_2,x_2} \cup L_{g_3,x_3} \).

may be strings, numbers, sets of strings or any other kind of data. The simplest annotation function is of the form \( f(id) = x \). The two examples mentioned above belong to this class.

Some functions take the group as a parameter: \( f(g, id) = x \), where \( g \in G \) and \( id \in L_g \). Functions of this form are called group-specific annotation functions. The expression function \( e_g : L_g \to \mathbb{R} \) is an annotation function of this type since it is associated with a specific group. In addition to expression values, other sample-specific data are extracted from chips. These include whether a probe is a control probe or a normal probe, whether a probe is saturated and the background noise level of a probe.

Finally, annotations may also be specific to an ID subset: \( f(L_{g,x}, id) = x \), where \( L_{g,x} \subset L_g \). Functions of this form are called ID-subset-specific annotation functions. Group-specific or ID-subset-specific functions may not generally transfer between group transforms. As we saw earlier, expression and weight values do transfer but they also require special transfer rules in the definitions of group transformation.
Consider the annotation that tells whether a probe is located on the green or red channel of a two-channel microarray. When two samples are combined using the ratio transformation, the annotation is no longer meaningful.

6.1.6 ID conversion functions

Since annotations and expression values are defined in the context of a specific ID namespace, arbitrary group transformations that modify the namespace are problematic because annotations using the old namespace can no longer be accessed. In this situation, annotation functions called ID conversion functions must be used.

A conversion function is an annotation function whose values are IDs. When $S_1$ and $S_2$ are namespaces, the function $f_{S_1 \rightarrow S_2}(id_1) = id_2$ maps IDs from $S_1$ to $S_2$. Generally, conversion functions are not group-specific or ID-subset-specific. Conversion functions can be chained: given $f_{S_1 \rightarrow S_2}$ and $f_{S_2 \rightarrow S_3}$, we can construct $f_{S_1 \rightarrow S_3}$.

Conversion functions are not necessarily bijective: if we map a protein ID to a gene ID, we may not be able to map the gene back to the same protein ID since genes encode many proteins. In fact, a protein may also be encoded by many genes, so the mapping from protein IDs to gene IDs is not, strictly speaking, a valid conversion function.

6.2 Data types

Data types for microarray analysis are mostly based on CSV (comma separated) files. CSV files may contain annotation in the form of key-value properties. Properties are stored on the first line of the CSV file. The columns of CSV files have a unique name.

Two main subtypes of the root CSV type are Matrix and Table. Matrix is a numeric rectangular matrix with named rows. A subtype of Matrix, Expression, is used for expression data. Expression data may be either channel values or log ratios; this is defined by a property.
The Table type is used for annotation. The columns are defined into key column and annotation columns. There may be any number \( K \geq 0 \) of key columns. For the most common cases of \( K = 1 \) and \( K = 2 \), the subtypes AnnotationTable and SampleAnnotationTable are defined.

Sample groups \( G_i \) are defined using SampleGroupTable, a CSV file that contains columns for the target group ID, group transformation type (ratio \( g_1/g_2 \) or median \( M(g_1, \ldots, g_n) \)) and a list of source groups. ID sets \( L_{g,x} \) are stored in using the GroupIDList type. The type defines columns for group \( g \), ID subset \( x \), ID namespace \( s(g) \) and the set of actual IDs. Support for ID set transformations is implemented using the type ListTransformation.

### 6.3 Component repository

The framework defines approximately 50 components for microarray analysis and 20 for report generation. The case study in section 7 demonstrates the use of the most often needed components. In this section, we focus on a few selected components.

#### 6.3.1 Automated report generation

Report generation is based on \LaTeX{}. Individual components produce \LaTeX{} fragments, i.e., incomplete documents. Fragments may refer to auxiliary files such as images. Fragments are compiled into a complete \LaTeX{} document by LatexCombiner. LatexTemplate provides customized headers and footers and allows to set page margins, for example. Finally, the complete \LaTeX{} document is compiled into PDF using LatexPDF.

ConfigurationReport is one of the most important documentation producing components. It produces a topology graph of the component network and a subsection for each component that describes the purpose of the component and values of all simple parameters. ConfigurationReport is an internal component that can access the network data structure that is being executed. An example of network visualization was seen in Figure 16. If a component implements methods published in a
Journal, the component description can refer to the publication using BibTeX.

6.3.2 Using SQL to transform CSV files

CSV files are used heavily in Anduril and a common need is to create custom reports for customers. CSV files are semantically close to relational tables; the Table type and its subtypes even define a primary key (unless $K = 0$). The Matrix type also has a primary key in practice since the rows are named. Consequently, the TableQuery component is used to execute SQL queries on tables based on CSV files.

TableQuery takes CSV files table1, ..., table8 and an SQL query as input. The CSV files are loaded into an in-memory database using HSQLDB, an open source embeddable database engine. A column col in tableN is referred to as tableN.col in the SQL query. The result of the query is written as a CSV file.

6.3.3 Statistical analysis

The StudentTest component selects DEGs using a t-test. The result is a SampleAnnotationTable file that contains p-values for each gene. Another output is a GroupIDList file that contains those IDs whose p-value is below a given threshold. Multiple hypothesis correction is implemented by the MultipleComparisonCorrection component. Supported methods are Bonferroni correction and two types of FDR. The component inserts another annotation column to the result file, containing either corrected p-values or so called q-values, depending on the method selected. Given the annotation table, genes with a low corrected p-value (or q-value) can be extracted into a GroupIDList file using the SampleAnnotationFilter component. The component selects those IDs whose p-values (or q-values) are on a user-configurable range $[a, b]$. The annotation table can also be queried using TableQuery.
7 Case study

We demonstrate the usefulness of Anduril for microarray analysis with a case study related to Kaposi’s sarcoma associated herpesvirus (KSHV), a cancer-related virus. KSHV, also known as human herpesvirus-8, causes Kaposi’s sarcoma (KS), a cancer that is particularly common and severe among AIDS patients [Gan06]. The focus of the study is on a KSVH protein v-cyclin (viral cyclin).

The effects of v-cyclin were studied by creating a knockout virus that removes the v-cyclin gene from the KSHV genome. There are two variants of the knockout virus designated 5i and 17d. Cells from a U20S (human osteosarcoma) cell line were used as a host for KSHV infection. Expression microarrays were used to study the gene expression effects of v-cyclin.

The experiment setup is shown in Table 3. There are three microarray experiments conducted on different dates. Agilent G4112F two channel microarrays were used that contain $4 \times 44000$ probes, i.e., a maximum of four hybridizations can be performed on one chip. In total, nine hybridizations M1, . . . , M9 were performed. One hybridization (M3) was repeated (as M6) due to quality problems noticed by the laboratory that conducted the microarray experiments.

All arrays have uninfected U20S RNA on one channel and either a wildtype KSHV or knockout KSHV on the other. This is an example of reference design. However, eventually uninfected U20S samples were not used in analysis and only KSHV infected samples were analyzed. The analysis was unconventional since one channel of each chip was not used in the analysis and channels from different hybridizations were compared to each other. This resembles an experiment setup used in one channel microarrays. The fine granularity of components allows to construct an analysis network that performs an unconventional analysis.
Table 3: Experiment setup for the KSHV case study. All microarrays have uninfected U20S cells as a control. The other channel contains one of KSHV wildtype (wt), KSHV v-cyclin knockout variant 5i or variant 17d.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Microarray</th>
<th>Green channel</th>
<th>Red channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>M1</td>
<td>U20S (uninfected)</td>
<td>U20S (KSHV wt)</td>
</tr>
<tr>
<td>E1</td>
<td>M2</td>
<td>U20S (uninfected)</td>
<td>U20S (KSHV 5i)</td>
</tr>
<tr>
<td>E2</td>
<td>M3</td>
<td>U20S (KSHV wt)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E2</td>
<td>M4</td>
<td>U20S (KSHV 5i)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E2</td>
<td>M5</td>
<td>U20S (KSHV 17d)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E2</td>
<td>M6</td>
<td>U20S (KSHV wt)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E2</td>
<td>M7</td>
<td>U20S (KSHV wt)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E3</td>
<td>M8</td>
<td>U20S (KSHV 5i)</td>
<td>U20S (uninfected)</td>
</tr>
<tr>
<td>E3</td>
<td>M9</td>
<td>U20S (KSHV 17d)</td>
<td>U20S (uninfected)</td>
</tr>
</tbody>
</table>

7.1 Quality control

Quality control and the actual analysis were done with separate component networks. While these tasks are normally combined into the same network, the unconventional analysis methods applied here would make the combined network inelegant for demonstration purposes. However, in the chosen approach some phases such as preprocessing must be done twice so in a large experiment it would be advantageous to combine the functionality into one network.

The microarray framework provides composite components for common Agilent microarray analysis tasks, such as data preprocessing and quality control. Using composite components, the quality control network consists of only 10 lines of configuration. Network execution takes 15 minutes and the resulting PDF document is over 50 pages long.

As microarray M3 was repeated due to quality problems noticed by laboratory, it is particularly interesting to see if the quality problems are visible using tools from the framework. Indeed, spatial analysis indicates that M3 has defects in the upper left section. Figure 23 shows a modified TIFF scan image of the red channel of M3. The scan image was modified using a logarithm transform to lower contrast. Intensity levels in the upper left section are lower than in other parts of the chip. The effect is similar on the green channel of M3. The low intensity of the extreme corners follows
Figure 23: Left: logarithm transformed scan plot of the red channel of M3. Right: duplicate probes for the red channel of M3. Expression values of each duplicate probe are transformed using $z = (x - \mu)/\sigma$ to bring them to the same range. Here, $\mu$ is the mean of the probe type and $\sigma$ is the standard deviation. Green spots denote probes whose expression value is below the average value of the probe type, i.e., $z < 0$. Red spots denote probes with $z > 0$ and yellow spots probes with $z \approx 0$. Black spots contain probes that have no duplicates.

Spatial effects can be quantified by analyzing the distribution of duplicate probes using the component DuplicateQuality. Agilent arrays contain 10 types of positive control probes, i.e., probes that show relatively high expression levels in most experiments. There are 32 copies of each probe. In the absence of spatial effects, control probes of the same type have consistent expression values; that is, their standard deviation (SD) is low. On both channels of M3, the mean SD = $\frac{1}{10} \sum_{i=1}^{10} \text{SD}(C_i)$ of control probes $C_i$ is 0.34. On other arrays, the mean SD is below 0.10. In particular, the repeated array M6 has a mean SD of 0.05 on both channels. Duplicate probes are visualized in Figure 23. The image shows all duplicate probes at once, also including non-control probes.

Relationships between hybridizations were analyzed using hierarchical clustering.
[HaK06] and correlation coefficients. Hierarchical clusters are illustrated in Figure 24. Both analyses show a clear batch effect where the samples from the same experiment correlate with each other. In particular, the M1-M2 arrays from experiment E1 correlate moderately with each other ($r^2 = 0.53$) and not with any other array ($r^2 < 0.05$ for other correlations). Correlation between M3 and its repeated version M6 is relatively high ($r^2 = 0.66$).

7.2 DEG analysis

In the main analysis, differentially expressed genes between KSHV wildtype and knockout variants were computed using the fold change method. The fold change method was used instead of the t-test since the t-test does not work well with a low number of samples. To compute fold changes, green channels were extracted from arrays M4-M9. Wildtype samples M6 and M7 were used as controls. The following ratios were computed: M4/M6 (5i vs. wt), M5/M6 (17d vs. wt), M8/M7

![Hierarchical clustering of sample groups.](image)

Figure 24: Hierarchical clustering of sample groups. Expression values of non-control probes with good quality were used to compute the distance between sample groups. Groups with low distance (y-axis) belong to the same cluster. The microarrays from the first experiment (M1 and M2) are similar to each other but dissimilar to all other microarrays. The microarrays from the two other experiments (M3–M9) are more comparable to each other. The repeated microarray M6 is similar to M3 despite M3 having local spatial defects.
(5i vs. wt) and M9/M7 (17d vs. wt). All ratios were computed using arrays from the same batch to minimize the batch effect. No normalization other than mean correction was done for the green channels. Experiment E1 was not used because arrays M1-M2 do not correlate with other arrays. Array M6 was used in place of M3.

Visualization for the DEG analysis network is shown in Figure 25. First, AgilentReader reads raw data files from an Agilent data directory and produces CSV files that contain expression values $e_g$ for green and red channels. AgilentReader also produces an AnnotationTable that contains basic annotation for the probes, such as gene name, description and whether the probe is a control probe. The file corresponds to annotation functions $f_{name}(id)$, $f_{description}(id)$, etc. AgilentReader takes as a second input a CSV file that maps file names to sample IDs.

ProbeWeights computes probe weights $w_g$ based on channel values and control probe status. DuplicateProcessor combines duplicate probes into one by taking a median of replicate values. Next, ExpressionFilter is used to select only those channels (green channels of M4-M9) that are used in the analysis. This step also removes control and bad quality probes (that have weight = 0). Mean correction is done by MeanNormalizer. The four types log ratios groups mentioned above are computed by an instance of SampleCombiner, logratio. The SampleGroupTable gives the definition of log ratio groups. Another SampleCombiner instance, combinedLogRatio, takes the median of the four log ratios ($M(M4/M6, M5/M6, M8/M7, M9/M7)$). The median group is used later in report generation.

DEGs are computed by FoldChange. The component takes log ratios as input and produces ID sets $L_{g,fcOver}$ and $L_{g,fcUnder}$ as a result. The ID sets contain over- and underexpressed genes for each group $g$. Also, the sets $L_{g,fcDeg} = L_{g,fcOver} \cup L_{g,fcUnder}$ are produced. In this analysis, fold change threshold 2 was used. The sets are named with a prefix $fc$ to differentiate between sets generated by other components. For example, the StudentTest component (not used in this analysis) produces $tDeg$ sets that represent statistically differentially expressed genes. Now, both $fcDeg$ and
Figure 25: Network topology for DEG analysis.
Figure 26: Visualization of sample groups and gene sets, generated by the ExperimentSetup component. Green channels of arrays M4-M9 are shown on the leaf level. On the next level, ratio groups derived from individual samples are shown. Each ratio group has associated gene sets that represent DEGs. Sizes of the gene sets are shown. Finally, the all group is defined as the median of the ratio groups. Gene sets $L_{all,over}$ and $L_{all,under}$ are intersections of $L_{gi,fcOver}$ and $L_{gi,fcUnder}$ sets, respectively.

tDeg sets can co-exist. Their intersection represents genes that satisfy both the fold change and t-test criteria.

Next, ListTransformer computes (i) genes that are overexpressed in all four cases ($L_{g1,fcOver} \cap L_{g2,fcOver} \cap L_{g3,fcOver} \cap L_{g4,fcOver}$) and (ii) genes that are underexpressed in all cases. These ID sets are denoted $L_{all,over}$ and $L_{all,under}$. They are computed using the UI function with $k = 4$. ListTransformer supports all transformations mentioned in section 6.1.4. Sample groups and ID sets are visualized by ExperimentSetup and are shown in Figure 26.

GeneTable produces CSV report files that contain gene names, median fold changes and descriptions for all genes in $L_{all,over}$ and $L_{all,under}$. There are 59 overexpressed and 30 underexpressed genes. The CSV files are handed to the client. Details of the gene lists are out of scope of this thesis and are being validated with additional biological experiments.
<table>
<thead>
<tr>
<th>Cluster</th>
<th>Size</th>
<th>Most specific common GO term</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>3</td>
<td>multicellular organismal development (3.7)</td>
</tr>
<tr>
<td>G2</td>
<td>7</td>
<td>intracellular part (1.2)</td>
</tr>
<tr>
<td>G3</td>
<td>2</td>
<td>substrate-specific transmembrane transporter activity (4.9)</td>
</tr>
<tr>
<td>G4</td>
<td>10</td>
<td>protein binding (2.7)</td>
</tr>
<tr>
<td>G5</td>
<td>7</td>
<td>cell part (0.1)</td>
</tr>
<tr>
<td>G6</td>
<td>2</td>
<td>response to chemical stimulus (6.2)</td>
</tr>
<tr>
<td>G7</td>
<td>6</td>
<td>catalytic activity (2.0)</td>
</tr>
<tr>
<td>G8</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4: Gene Ontology clusters that were identified in the case study. For each cluster, the number of genes and the most specific GO annotation term are shown. A measure of the specificity of each GO term is shown in parenthesis. A large number corresponds to a more specific, and generally more interesting, GO term. Cluster G8 has no common GO terms.

### 7.3 Gene Ontology analysis

Gene Ontology annotations for the gene set $L_{all,over}$ were retrieved using Ensembl [HBB02] and genes were clustered based on the annotation using the method described in section 4.5. Out of 59 genes, 44 had GO annotation. Eight clusters with more than one member were identified. The clusters are shown in Table 7.3. These results indicate subsets of DEGs that have a specific functionality or location in cells. The clusters can be used to design further laboratory experiments that verify the predicted functionality.
8 Conclusions

Anduril provides a modular and systematic architecture for microarray analysis. New analysis methods are constantly being developed and using the component model of the framework, they can be implemented and integrated into analysis networks. Components provide a mechanism for code reuse which increases productivity and reliability. Even more importantly, analysis projects are able to reuse the architecture of the framework. The framework may be used in a black box manner by connecting existing components together, or as a combination of black and white box by writing additional components.

An inherent tradeoff for using frameworks is that they may be challenging to learn. However, the reuse interface of Anduril is much simpler than for more complex frameworks, such as the GUI framework Swing. As network configuration is done with a custom high-level language, it is not necessary to write any code in a traditional language, unless new components are implemented. The component network has a natural visual representation and in the future the network could be specified using a GUI.

The component model is based on executables that read and write files. The model is relatively simple and enables writing components in multiple programming languages. This allows the use of the most suitable tool for a task and to leverage existing libraries in many languages. Language-specific mini-frameworks implement common functionality and make writing components easier. A tradeoff of the component model is that using several languages and mini-frameworks increases complexity. In particular, installation of the system may be difficult if a large number of languages and external libraries are used. Thus it is beneficial to focus efforts on a few key languages. Another tradeoff in the model is that a new process is launched when a component is executed. This may add overhead to component execution. For example, the overhead is one second for R and several seconds for Matlab.

Components are not only code: uniform documentation and black box test cases are important elements of components. In addition to checking component correctness,
test cases are used as example use cases that supplement component documentation. Components do not normally have explicit dependencies between each other. Rather, they communicate indirectly using a network of components. They do have implicit dependencies, however. The output of a component must be in a form that is usable for others. This means that component interfaces cannot be designed in isolation of each other. Components have explicit dependencies for data types. This makes it difficult to change a data type that is used by several components. Consequently, a component repository and the corresponding type system must be carefully designed. They should initially be used in internal pilot projects and released to public after the repository has matured.

The component network is statically typed which enables to catch many errors before execution begins. This is especially important for long analysis runs. The comprehensive type system supports subtyping and type parameters. Type parameters increase the flexibility of components while being mostly transparent to the user. Actual types for type parameters can usually be inferred automatically.

Often microarray analysis is iterative in nature. Anduril supports iteration by automatically executing only those components whose configuration has been changed. Since the component model is simple, it is usually possible to accurately determine whether a component needs to be re-executed. A consequence of partial re-execution is that each component instance must be executed at most once, i.e., the network must be acyclic. This limits the theoretical expressive power of the network topology. However, network execution is only one phase in the larger iterative analysis process, which may be cyclic.

The PDF document generating components produce high quality documentation that is used for communication to the client. Network topology visualization provides an intuitive overview of the analysis. In the future, generated reports could be used as article supplements in published analysis results.

Our focus has been on the backbone framework and the microarray-specific component repository, but the framework is suitable for other bioinformatics and scientific
analyses as well. The framework supports the iterative nature of scientific analysis and the visualization of a component network can be used to enhance communication between the analyst and the client.
References


OLH08b  Ovaska, K., Laakso, M. and Hautaniemi, S., Component-based workflow framework for rapid and systematic data analysis. In review.


