Reproducible Biological Networks for Personalized Oncology

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Thorsten Tiede

aus Bayreuth

Tübingen
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	24.03.2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Oliver Kohlbacher
2. Berichterstatter:	Prof. Dr. Sven Nahnsen

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead

Erklärung

Ich erkläre hiermit, dass ich die zur Promotion eingereichte Arbeit mit dem Titel:

Reproducible Biological Networks for Personalized Oncology

selbständig verfasst, nur die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen sowie gemeinsam verfasste Teile der Arbeit als solche gekennzeichnet habe. Ich erkläre, dass die Richtlinien zur Sicherung guter wissenschaftlicher Praxis der Universität Tübingen (Beschluss des Senats vom 25.5.2000) beachtet wurden. Ich versichere an Eides statt, dass diese Angaben wahr sind und dass ich nichts verschwiegen habe. Mir ist bekannt, dass die falsche Abgabe einer Versicherung an Eides statt mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft wird.

Ort, Datum

Unterschrift

Abstract

An ever increasing amount of data in the life sciences sparked the era of truly personalized medicine. A medicine, where treatments are tailored to individual patients and therapeutic decisions are based on data driven disease profiles. Biological networks enable the computational processing of the relationships and properties of the cellular processes. They help to elucidate emerging properties and downstream effects of mutations and pharmaceutical interventions alike. Network representations of the cellular processes that aid therapeutic decisions have to be manually assembled in labor-intensive preparation phases, before they can be discussed by the experts in the field. In research, most publications that do report biological networks as part of their methodology or results, fail to adequately disclose the provenance of their networks. By not providing machine-readable representations of the processes they discuss, reproducibility of the findings is hampered.

Here, we present SBML4j, a service-oriented application that provides customizable biological networks through annotation, filtering and various graph-algorithmic computations. It is easily integrated into existing clinical tool chains through the provided RESTful interface and the intuitive Python library. We demonstrate this with the integration of SBML4j with a clinical variant annotation pipeline and a web-based frontend for visual exploration of variants in their genetic neighborhood. SBML4j creates network mappings from standardized and curated systems biology models and pathways. By using well-defined biological qualifiers and ontologies, the provenance of the networks and their biological entities is clearly defined. A comprehensive provenance report for any network provided by SBML4j ensures reproducibility. This provenance can even be tracked when external applications consume networks that are provided by SBML4j or when externally created networks are uploaded to SBML4j as derivatives of existing networks.

With this, SBML4j delivers reproducible biological-network knowledge to personalized medicine approaches. At the same time it enables research groups to provide detailed provenance information and machine-readable representations of biological networks. As an open-source project with a non-restrictive license, SBML4j will be further developed by an active research community and can help to shape the future of personalized medicine.

Zusammenfassung

Die ständig wachsende Menge an Daten in den Biowissenschaften hat die Ära der wirklich personalisierten Medizin eingeleitet. Eine Medizin, in der Behandlungen auf den einzelnen Patienten zugeschnitten sind und therapeutische Entscheidungen auf datengestützten Krankheitsprofilen beruhen. Biologische Netzwerke ermöglichen die computergestützte Verarbeitung der Beziehungen und Eigenschaften zellulärer Prozesse. Sie helfen dabei, neue Eigenschaften und nachgelagerte Wirkungen von Mutationen und pharmazeutischen Eingriffen gleichermaßen aufzuklären. Netzwerkdarstellungen, die therapeutische Entscheidungen unterstützen, müssen in arbeitsintensiven Vorbereitungsphasen manuell zusammengestellt werden, bevor sie von Experten auf dem Gebiet diskutiert werden können. Biologische Netzwerke, die in wissenschaftlichen Veröffentlichungen genutzt oder präsentiert werden, sind häufig weder maschinenlesbar noch ist ihre Herkunft eindeutig ersichtlich. Dies erschwert die Reproduzierbarkeit der Ergebnisse und verhindert ihre effektive Weiterverwendung.

Hier stellen wir SBML4j vor, eine serviceorientierte Anwendung, die anpassbare biologische Netzwerke bereitstellt. Sie lässt sich durch die vorhandene REST-basierte Schnittstelle und die intuitive Python-Bibliothek leicht in bestehende klinische Softwareprozesse integrieren. Wir demonstrieren dies anhand der Integration von SBML4j mit einer Annotationspipeline für klinische Varianten in ein webbasiertes Frontend. Dieses ermöglicht die visuelle Untersuchung von Varianten in ihrer genetischen Nachbarschaft. SBML4j erstellt Netzwerk-Mappings aus standardisierten und kuratierten systembiologischen Modellen und Pathways. Ein umfassender Provenance-Report für jedes von SBML4j bereitgestellte Netzwerk gewährleistet die Reproduzierbarkeit. Diese Provenance kann sogar nachverfolgt werden, wenn externe Anwendungen die von SBML4j bereitgestellten Netzwerke nutzen oder wenn extern erstellte Netzwerke als Derivate bestehender Netzwerke in SBML4j hochgeladen werden.

Damit liefert SBML4j reproduzierbares biologisches Netzwerkwissen für Ansätze der personalisierten Medizin. Gleichzeitig ermöglicht es Forschungsgruppen, detaillierte Herkunftsinformationen und maschinenlesbare Darstellungen biologischer Netzwerke bereitzustellen. Als Open-Source-Projekt mit einer nicht restriktiven Lizenz wird SBML4j von einer aktiven Forschungsgemeinschaft weiterentwickelt und kann die Zukunft der personalisierten Medizin mitgestalten.

General Remarks

- In accordance with the standard scientific protocol, I will use the personal pronoun ***we*** to indicate the reader and the writer, or my scientific collaborators and myself.

Contents

Chapter 1

Introduction

Motivation

In March 2022, "the Telomere-to-Telomere [...] Consortium presents a complete 3.055 billion–base pair sequence of a human genome"[1], covering the additional eight percent of missing sequence from the heterochromatin regions. Their draft genome includes gapless assemblies for all 22 autosomes and the complete sequence of the X-Chromosome. It contains around 200 million novel base pairs containing 1,956 paralogous gene copies, of which they predict 99 protein coding genes. With this the total amount of protein-coding genes is believed to be 19,969. This marks the most complete genomic sequence that has been published to date.

Which proteins are all those genes encoding? What biological function does each of them have? How do they influence the complex machinery that runs in each of our cells? Which consequences on the cellular level does a malfunction have?

These are some of the questions that we need to answer to reach a truly personalized medicine. A medicine that has the ambition to find individualized therapies for a variety of diseases. A medicine that tailors pharmaceutical interventions to a patient's genetic and transcriptomic profile. A genetic mutation in a patient's gene can lead to the malfunction or inactivation of the enzyme it encodes and subsequently disrupt a cell's metabolic, signaling or regulatory mechanics. This can lead to unchallenged proliferation or prevent the cell's natural death and ultimately result in uncontrolled growth and the emergence of tumor tissue. However, it is not the single genetic variation, called a `variant`, by itself that leads to disease but rather its impact on the cellular machinery. The ripple effects that emanate from this one malfunction in the connected network of the cell's inner mechanisms disrupt the whole system. Therefore, it is necessary to understand the effects of an observed variant on the complex network of interactions and pathways in the patient's cells. With this understanding, a personalized pharmaceutical intervention tailored to the patient's needs and genetic profile

can be found, with the goal of curing the disease or at least alleviating symptoms with minimal side effects.

In today's modern medicine and research landscape the amount of data collected is constantly growing. This makes computational methods indispensable as tools to care for patients, decide on the right therapeutic intervention and discover new pharmaceuticals. The traditional biological process for generating knowledge about cellular function involved dissecting a single part of the puzzle and learning about its function. The aim was to find a chemical compound that affects this sole part and observe the changes in its behavior.

More recently, science tries to elucidate the mechanism of the whole system, find disrupted entities, and find entry points within the system to circumvent these disruptions. The discipline that tries to understand and model the functioning of the system is called *systems biology*. Both systems biology and human comprehension need interpretable representations of the complex interactions found in the cell. Without a proper representative model experts would not be able to discuss and report their findings or calculate mathematical and statistical properties of these systems. A popular means of describing these processes are *biological networks*. The concept of nodes and edges is easily understandable by humans and, if properly represented, also machine-readable and thus processable by computational methods.

Many systems biology approaches create biological networks from experimental data of biological systems or they use networks to model these processes. Experts need to investigate patient data in the context of these networks to decide on therapeutic interventions. This emphasizes the need for biological networks in science and clinical settings. Both scenarios need clear information about the networks' origins as well as about the modifications that have been applied to them in order to reach a certain clinical decision or scientific conclusion. In short, the networks must be *reproducible*. However, many online resources and scientific publications that offer their networks for download fail to provide machine-readable representations or to disclose vital provenance information. To enable reproducible science that involves biological-network knowledge, a network resource should track the provenance and provide instructions for recreating a certain representation.

Biological networks must be *accessible*. This means that there have to be ways for retrieving networks without expert knowledge on the complex database schemata in which these types of networks are stored. Network-oriented web services offer easy-to-use search- and programming-interfaces for getting network representations of interactions between biological entities. However, using these services in a clinical context can be challenging. For instance, it is impossible to upload sensitive patient data, like genetic variants, onto these websites as data privacy laws prohibit such actions. Therefore, these mapping and analysis steps have to be performed and documented locally after retrieving a network. This leads to a lack of *interoperability*, because a manually created network representation cannot easily be reused in another computational system for further analysis. Many available research frameworks

that are intended for local use require hand-crafted input files and the execution of complex scripts to initialize the databases. Users often require advanced knowledge of database-query languages like SQL or Cypher to obtain knowledge for hypothesis generation. In addition, the integration of these frameworks into more complex research pipelines is difficult due to the nature of the required expertise for information extraction and lack of programmatic access points.

We present a service-oriented application for biological networks that provides machine-readable representations and all necessary provenance information for reproducing the networks. The knowledge graph in the database is derived from hand-curated systems-biology models. These models are annotated with terms from well-known ontologies and biological-qualifier systems. This allows us to integrate models from various sources that use different internal identifier systems. Network-mappings reduce the complexity of the contained information for automatic processing and allow different views on the cellular processes. Popular examples are metabolic networks, regulatory events or signaling cascades. These mappings can be programmatically generated, annotated, filtered, and analyzed with various graph algorithms. A local installation using containerized application- and database-instances guarantees data-protection conform access to these features, as no internet connection is required once a database is set up and all computations can be performed on readily available desktop-hardware. Access to the networks is provided via a RESTful interface and an intuitive Python library. This allows easy integration of a full range of network-biology related analysis and computations into analysis pipelines and environments. With the integrated provenance tracking the reproducibility of the provided networks is ensured.

Structure of the thesis

This thesis introduces the software `SBML4j`, a service-oriented approach for creating and working with biological networks for reproducible science and clinical decision making. It offers standardized data input for the network sources as well as data mapping, comprehensive analytical power accessible via REST endpoints, and an intuitive Python library for the integration in existing analytics pipelines. This approach offers data reproducibility, generation of individual network representations using patient data and biological network knowledge across traditional pathway boundaries within the limitations of modern data-privacy regulations.

Chapter 2 will provide the medical background of cancer and its treatment options, the systems bioinformatics terminology of the network-representations and information-technology methods at the heart of the application. Chapter 3 will present the software `SBML4j`. We show the implementation details of the Java-based software and the accompanying Python library `pysbml4j` and give details about the underlying database structure. Chapter 4 highlights how external network-analysis tools can be seamlessly integrated into the provenance tracking

mechanisms of SBML4j by providing networks to an application that calculates deregulated subnetworks. The resulting subnetworks are then integrated into the database of SBML4j. We demonstrate how additional provenance information about such an external analysis step can be integrated into the provenance graph to ensure reproducibility. In Chapter 5, we then present `PeCaX`, the `Personalized Cancer and network eXplorer`, which demonstrates the integration of `SBML4j` into a web-based analysis platform. It combines the clinical-variant annotation-pipeline `ClinVAP`, our network resource `SBML4j` and the network-visualization tool `BiographVisart`. It is a comprehensive and easy-to-use platform for the analysis of cancer patient data, which offers auto-generated patient-specific biological networks for annotated variants.

Chapter 6 concludes the thesis and gives a brief outlook on future directions of SBML4j for shaping personalized medicine and network-based research.

Chapter 2

Background

2.1 Personalized Medicine

The first mention of the term *Personalized Medicine* can be found as early as 1998 in the title of the monograph "Personalized Medicine: the impact of pharmacogenetics on drug development"[2]. Later, the author K. K. Jain, stated that "[p]ersonalized medicine simply means the prescription of specific therapeutics best suited for an individual based on pharmacogenetic and pharmacogenomic information"[3]. On that basis, *Pharmacogenetics* focuses on relating alterations in heritable traits, such as genes, to variation in drug response[4]. *Pharmacogenomics* aims towards defining genetic markers in the human genome that are able to reliably predict the carrier's response to certain drugs[5].

An early example of this principle can be found in the increased response to a specific tyrosine kinase inhibitor in patients with the *Philadelphia translocation* in their genomes[6]. This inhibitor was later patented under the name *Imatinib* and its development "as a specific molecularly targeted therapy"[7] is one of the earliest and most prominent examples of drug development driven by genetic aberrations. The advances in this area are also visible in the number of approvals of drugs that use *pharmacogenetic* information. Their numbers rose from fewer than ten in the decade from 1976 to 1985 to more than 70 in the years 1996 to 2005[8]. Since then, a multitude of tailored therapies based on genetic profiles and other biomarkers have been developed. For instance, algorithms for detecting differentially regulated pathways based on gene-set enrichment analysis (GSEA)[9] have become a popular method. More recently, the effect of functional variants in human drug-related genes on drug efficacy have been investigated[10]. Furthermore, personalized peptide vaccines are targets of current research into personalized vaccination strategies[11,12]. With the emergence of Machine Learning (ML) methods, algorithms and techniques using ML are actively investigated for their use in personalized medicine approaches[13].

2.1.1 Therapeutic Strategies

When determining a treatment one can distinguish between three different types of strategies for choosing a pharmaceutical intervention for a patient as is shown in Fig. 2.1. The classical one-drug for one disease approach (see Fig. 2.1A) determines the phenotype of the disease and assigns all patients to the same recommended uniform treatment plan. Additionally, two major strategies for personalizing medicine can be formulated, patient stratification (see Fig. 2.1B) and individualized therapy (see Fig. 2.1C).

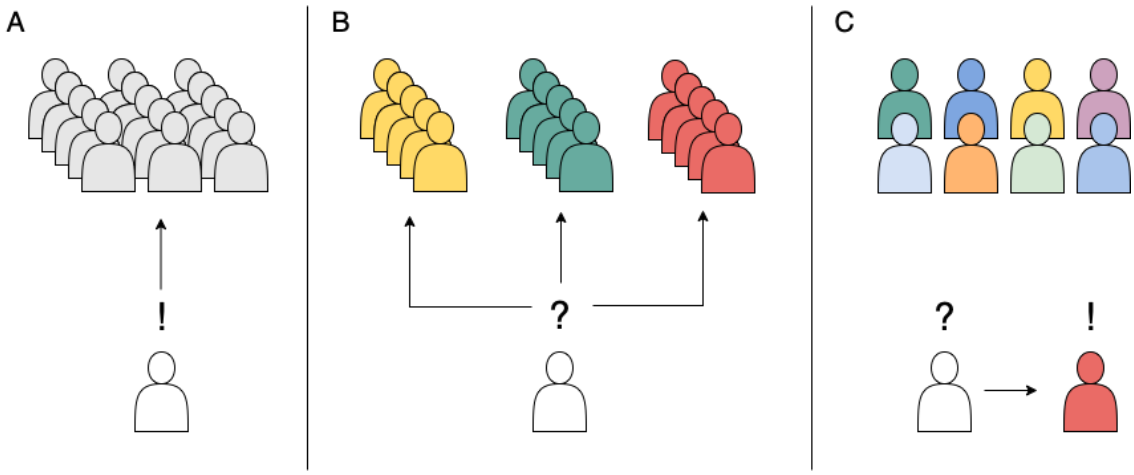


Figure 2.1: Three strategies for pharmaceutical intervention selection. A) Uniform treatment plan for a disease. A patient with a diagnosed disease receives the uniform intervention defined for this disease. B) Patient stratification. A patient gets assigned to a subgroup of patients with a similar disease-driving mechanism. For each subgroup a suitable therapy needs to be found and administered. C) Individualized therapy. The genetic profile of a patient is analyzed and a therapeutic intervention is tailored to their needs.

Patient Stratification

In the *stratification* strategy, an individual is assigned to a *patient-cohort* based on a specific set of biomarkers (see Fig. 2.1B). This cohort represents a subgroup of patients for which an improved response to a specific drug has already been shown[14]. The goal of *stratification* is to find a therapy for any of the patient-subgroups that has a better chance of helping or even curing all patients in this group than a uniform treatment plan across all patients suffering from this disease. These subtypes are defined by differences in the genetic profile of the patients. This allows for specific means of attacking the disease mechanism in these patients.

One challenge is to identify such a group of patients and reliably distinguish them from all other patients. Another challenge is to find a treatment tailored to this group of patients and the specific mechanism that led to the onset and progression of the disease in said patients.

The third challenge is to identify an individual patient as part of this group based only on the data of this single patient.

Individualized Therapy

In a truly *personalized* medicine, an appropriate drug is selected based on the biomarkers from *one patient* (see Fig. 2.1C). The genetic profile of this patient is assessed individually and the selected therapeutic intervention will only work for this specific patient. Nowadays, this can be found for difficult-to-treat tumors, whose biomarkers are assessed in so-called Molecular-Tumor-Board (MTB) Meetings[15]. For the context of this thesis, we will focus on this type of therapy in the field of oncology. The reader may be advised that this form of individualized therapeutic interventions can be found in other fields of medicine as well.

2.1.2 Personalized Therapies in Oncology

Disease Symptoms of Cancer

Cancer describes a set of similar diseases that are characterized by the development of malignant tumors. These tumors grow uncontrollably by suppressing processes that normally lead to the tumor cells' death and by activating pathways that lead to cell division and growth. The National Cancer Institute lists more than 150 types of cancer[16]. Malignant tumors that grow to macroscopically visible sizes are a serious health risk. Untreated tumors will result in failure of the affected organs and can spread to other parts of the body by forming metastases. These malignant secondary tumor sites usually worsen the symptoms and make a positive outcome of any therapy highly unlikely. Once metastases have been formed eradicating all tumor sites from the body is nearly impossible. This highlights the need for the early discovery and efficient therapy of cancer.

Standard Therapy

The standard therapy for most cancers is a combination of several interventions, with chemotherapy, radiation therapy, and surgery being the most widely used. In radiation therapy, the tumor tissue is bombarded with high doses of radiation, which directly kills cancer cells. Chemotherapy uses pharmaceuticals to target fast-growing cells, which are often found in tumor tissue. There is an abundance of different drugs for this therapy type, which have to be approved for specific cancer types. Choosing the right chemotherapeutic can already be considered to be a targeted approach, as the tumor type plays a major role in this decision. Both radiation- and chemotherapy have the goal to reduce the size of the tumors or slow their growth. The third standard therapy is surgery in which the tumor tissue is removed from the body. Surgery can be performed before or after other therapies, again highlighting the fact

that the standard therapy approach is already targeted towards the tumor and the progression of the disease. A successful therapy eradicates all traces of the tumor from the patient's body and effectively cures the patient. However, if these classical therapies do not prove effective, the next step is targeted therapy.

Targeted Therapies

Based on the genetic factors, which influence disease onset and progression, various personalized therapies can be conducted. They aim at specific proteins inside the tumor cells and have the goal to disrupt the cancer-causing mechanism inside the cells. At the beginning of targeted therapies stands the search for *biomarkers* that are specific to the tumor tissue and are not present in healthy tissue. This makes them viable candidates for therapeutic intervention, as potential drugs specifically target the tumor tissue. This keeps side effects to a minimum as healthy tissue surrounding the tumor is not affected by the drugs. To find cancer-specific *biomarkers*, the tumor tissue needs to be screened for genetic variation and differential expression of proteins compared to healthy tissue.

Genetic Basics of Tumor Development and Growth

A lot of research has been done to elucidate the mechanisms of tumor development and different theories about its genesis were formulated throughout the years. First signs of genetic influences on tumor growth were found in research conducted with the *Rous sarcoma virus (RSV)* which infects chicken[17]. Stehelin et al. discovered that a gene in the viral genome called *src* could cause an infected normal cell to become a tumor cell[17]. Genes like the *viral src (v-src)* that have the ability to cause cancer are called *oncogenes*. Comparable viruses that affect chicken are missing this gene and without it do not have the ability to cause tumors in their hosts. This hints at the fact that the progenitor of RSV did not possess this *v-src* gene but has acquired it during its normal cellular replication cycle from its host. A host variant of the *src* gene, called *cellular src (c-src)* can also be found in uninfected vertebrates[18]. *C-src* has the ability to induce cancer when mutated and is thus called a *proto-oncogene*.

Three main activating mutations affect *(proto-)oncogenes* and can lead to the development of tumors. First, there are *single nucleotide variants (SNVs)*, mutations of a single nucleotide in a gene. They can result in altered or improper protein-function, complete loss of enzymatic activity of the resulting protein or no protein production at all.

Second, a gene or a whole section of a chromosome can be multiplied, leading to *copy-number variations (CNVs)*. These alterations lead to different rates of expression of the genes on these regions of the genome, altering the transcriptional state of the cell. This in turn can have different regulatory effects within the cellular apparatus and disrupt transcriptional balance, activate, or inhibit signaling cascades and alter metabolic processes.

The third mutational mechanism is the *chromosomal translocation*. Here parts of a chromosome are transferred onto another chromosome, which puts all genes on this part of the genome under the regulatory influence of the new location. This can lead to different expression patterns that affect a whole range of proteins and regulatory factors, which alter the state of the cell and allow for potential malignant growth.

Another class of genes that play a role in the transition of a cell from a normal state to a tumor state are so-called *tumor suppressor genes (TSGs)* [19]. In their wild-type form, these types of genes govern cell proliferation. Their presence in the genome limits a cell's proliferation capacity, which keeps cell division rates in a desired range. If they are affected by mutations that limit or inhibit their function, those control mechanisms are missing. This can contribute to an increased rate of proliferation and to the development of neoplasms. As two independent mutations in both alleles of the genome must appear to show a dominant phenotypic behavior, these tumor-inducing mutations are susceptible for a germline-inherited increased risk of cancer. Numerous heritable cancer syndromes with an association to this type of inherited defective TSGs are reported in literature [20,21].

With each cell-division the genome of dividing cells is duplicated. During the copy process of the double stranded deoxyribonucleic acid (DNA) mistakes will happen, which need to be repaired by cellular repair mechanisms. A mutation in one of the genes that encode the enzymes responsible for the repair will result in a lack of adequate DNA repair. Such uncorrected errors are then persisted in the cell's genome and will be passed on to any descendants of the cell. If such an error affects, for example, a proto-oncogene or TSG, the initial mutation in the repair-encoding gene indirectly contributes to the development of tumor tissue. One example of such an effect can be found in the hereditary form of non-polyposis colorectal cancer, in which the apparatus responsible for detecting mistakes in DNA replication is mutated and its function limited [22,23].

In addition, epigenetic factors play a role in the development of tumor tissue. A prominent example is the *methylation of cytidine* residues which are present in so-called *CpG* dinucleotide sequences. These sequences are often found in the surroundings of *promotor regions* of various genes that are involved in the development of tumors [24]. Promotors are sequences of DNA where a class of enzymes, called *transcription factors*, can bind and initiate the transcription of genes that lie downstream on the DNA. The attachment of methyl-groups to these DNA regions can prevent the binding of these factors and inhibit the attachment of RNA polymerase, the enzyme that synthesizes RNA. Additionally, this modification can lead to a change in the arrangement of nearby chromatin, which in turn can suppress the transcription of whole sections of DNA [25]. Through special DNA methylases, this modification will be reintroduced into the DNA strands of daughter cells after cell division, which manifests the methylation in this cell's heritage-line. This transmits the repression of certain genes without the need of changes in the DNA sequence. In fact, it could be shown that in *ovarian clear cell carcinoma*

the loss-of-function of TSGs and DNA repair genes is more frequently due to DNA methylation than direct mutation[26].

Tumor cells often exhibit aberrations in the gene which encode the enzyme *telomerase* or in regions that regulate the expression of this gene. This enzyme can extend the *telomers*, which are long stretches of non-coding DNA at the end of each chromosome. Naturally these ends get shortened with each cell-division and result in an end-to-end fusion of chromosomes when they become too short. This leads to karyotypic instability and will send a cell into programmed cell-death[27]. An increased expression of the *telomerase* enables tumor cells to prolong the *telomers* of the chromosomes, which enables the cells to proliferate almost indefinitely[28].

Non-genetic Factors Influencing Disease Progression

Apart from these genetic and epigenetic factors for disease onset and progression, non-genetic factors influence the disease. The micro-environment of the tumor is believed to trigger cascades, which allow the malignant tissue to recruit blood vessels in a process called *angiogenesis*[29]. This process is needed to sufficiently provide the tumor tissue with nutrients and oxygen and is one example of a growth-enabling mechanism that has no direct genetic origin. It has been subject of intense research for treating cancer and other diseases that rely on *angiogenesis*[30–32].

To form metastasis, tumor cells need to complete a complex sequence of steps, which collectively have been termed the *invasion-metastasis cascade*[33]. Some of these steps include the ability to enter blood or lymph vessels, to travel to distant sites in the body and to successfully colonize the tissue at these sites. A process called *epithelial-mesenchymal-transitions (EMT)*[34] is one possible explanation for how cancer cells manage to complete these complicated and unlikely steps. This process is known from early embryonic morphogenesis, where normally immobilized epithelial cells acquire the ability for motility and invasiveness — phenotypical traits of mesenchymal cells — by changing their differentiation program. These changes can be triggered by a few transcription factors, whose activation can repress epithelial genes and activate mesenchymal genes[35]. Thiery et al. showed in their experiments that these highly conserved processes, which are essential for the development of multicellular organisms, can be exploited by malignant carcinoma. The extracellular environment of the tumor tissue gives rise to signals that can trigger signal-transduction pathways leading to EMT, allowing a tumor to metastasize[36]. Once the cells are settled in the new tissue, the inter-cellular signals from the healthy environment revert the changes from EMT, which allows them to return to their malignant epithelial state and start growing again, thus forming a metastasis. Chemokine receptors that are mediators of this spreading of tumors could potentially be future targets for individualized therapy[37,38].

As shown, the development and progression of cancer is influenced by many factors. In rare cases, classical treatment or standardized therapy based on biomarkers do not lead to remission or slow down tumor growth. For these patients, the MTB suggestions are the last hope for successful treatment.

Molecular Tumor Boards

An MTB is a meeting where clinical experts discuss cases of difficult-to-treat tumors to find new treatment options based on the tumor's molecular profile. They assess the patient's tumor according to their individual genetic profile and propose new, unconventional therapeutic interventions based on these discussions. With growing numbers of cancer-patients and an abundance of genomic data, the need and opportunity for individualized therapy is rising. Experienced participants of MTB discussions spend less than four minutes on a single case[39]. Therefore, decisions have to be made fast and on solidly prepared casefiles.

One area where there is a need for decision support systems are the MTB meetings themselves. The Cancer Core Europe Consortium implemented a web-based portal solution for aiding MTBs, the *Molecular Tumor Board Portal*[39]. Their website offers a modern design and enables interactive investigation of variants and drugs, even for germline mutations when used on-site. It is backed by a unified and automated bioinformatics-analysis pipeline that promises to deliver consistently processed data. Their portal[40] provides statistical data about the cases but, at the time of writing, does not seem to use biological networks for relationship investigation.

Another area, which needs supporting systems is the labor-intensive preparation-phase of the MTB meetings[41]. During preparation, patient-data is analyzed with bioinformatics pipelines to extract all variants of interest for each case. These variants are then manually investigated on portals like OncoKB[42] and the results tediously assembled in static presentations. Inquiries of relationships of potential drug-targets to their genetic neighborhood are often part of this process. Popular resources for these investigations are the *String database*[43] or the *Kyoto Encyclopedia of Genes and Genomes (KEGG)*[44].

Providing automated and reproducible biological networks for preparation and presentation that can be annotated with patient data and expert knowledge can lead to better decisions for patients.

2.2 Systems Bioinformatics

Systems Bioinformatics is an interdisciplinary field of research that aims to elucidate cellular processes using Bioinformatics methods. One area of the field uses Bioinformatics to analyze and process biological data in the form of genetic, transcriptomic, or proteomic data. From

this data, interactions of biological entities can be derived and system-wide pictures of the cellular apparatus can be created. Another aspect of the field employs Bioinformatics methods to elucidate emerging properties from these networks of cellular function. For both there is a need to represent the biological processes in a machine-readable format, which describes the biological entities and their relations. This format must ensure that all entities can be identified and semantically integrated with pre-existing knowledge.

2.2.1 Biological Networks

Biological networks are a representation of biological entities and the relations between them. The entities in the network are called nodes or vertices, the relationships are represented as edges connecting these nodes. The most prominent form of a biological network is a *biological pathway*. A pathway describes a subset of cellular processes dedicated to the production of some chemical compound, the regulation of a gene, or the transmission of a chemical signal through signaling cascades. It can be understood as "a series of actions among molecules in a cell that leads to a certain product or a change in the cell"[45]. Often these actions are grouped into functional complementary units or processes with similar targets and are then depicted as so-called *pathway maps*.

One of the most recognizable visual representations of these *pathway-maps* are found in the Kyoto Encyclopedia of Genes and Genomes (KEGG) from the Kanehisa Labs in Kyoto, Japan[44]. Their hand-curated database features visual representations of many cellular processes for a variety of species, which are used by the scientific community around the world. Fig. 2.2 shows the *Notch signaling pathway* from the KEGG pathway database, which depicts the intracellular effects of the binding of an extracellular ligand to the membrane-bound receptor `Notch`. The colors and style of the layout are signature elements of the KEGG *pathway maps*. The green rectangles represent proteins, which can be grouped together to protein-complexes. The white circle depicts a chemical compound, in this case a piece of DNA. The rounded white rectangle is a reference to another pathway map, which is affected by the events depicted here. The vertical gray bars show the cell membrane, the dotted vertical line indicates the membrane of the nucleus. The black lines between the proteins represent the signaling events, an arrow indicates a positive signal, which stimulates the receiving enzyme. The T-shaped connection indicates an inhibition of the protein at the T-head, mediated by the entity on the other end. The arrow between the box labeled "CSL" and the DNA indicate that the expression is stimulated, which results in elevated expression levels of the genes that are connected with arrows coming from the DNA symbol. Dotted lines stand for indirect links or unknown reactions, in this case the reference to another pathway map and to potentially more genes whose expressions are increased. White boxes indicate proteins that have no known homolog in the currently visualized organism, but can play an important role in the same pathway in other organisms.

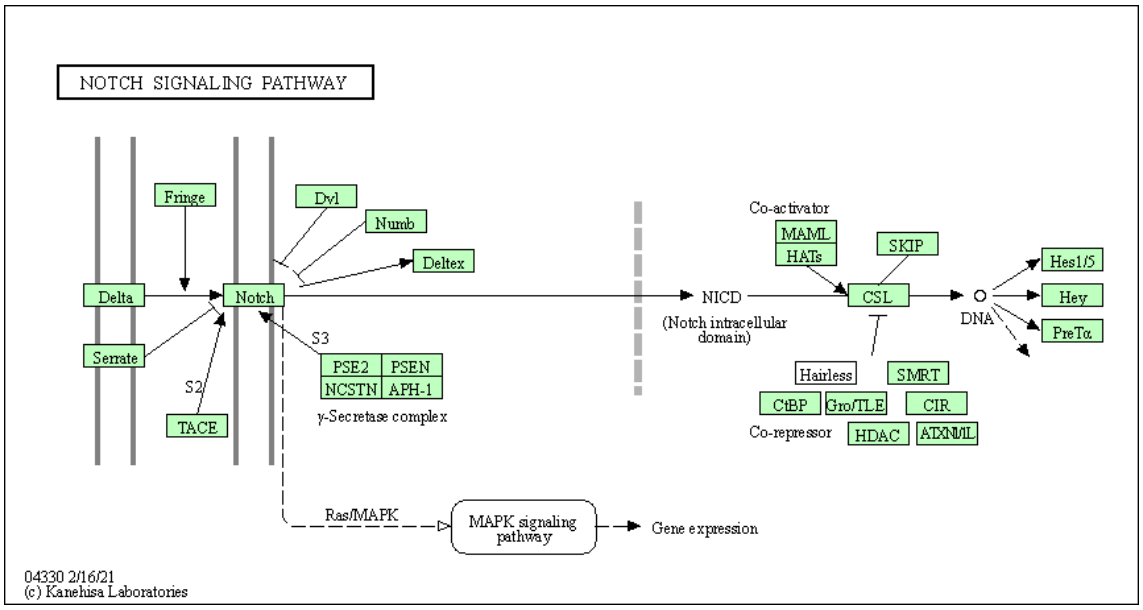


Figure 2.2: Notch signaling pathway map from the KEGG pathway database[44]. Signaling cascade of the membrane-bound protein "Notch". Green and white rectangular boxes show proteins, connecting arrows indicate activating or inhibitory signaling events. Through these events, the binding of Delta or Serrate to the membrane-bound Notch protein influences the expression of several proteins, including Hes1/5, Hey, and PreTα.

In this case the protein named "Hairless" has not been found in Homo sapiens, but can act as a co-repressor in some insects like *Drosophila melanogaster*[46].

This stylized representation conveys a lot of information about the processes it describes. For this reason, these visual depictions are a perfect medium for transporting such complex information to the human mind. Therefore, it is not surprising that these pathway maps and their recognizable features are used by scientists and clinicians alike.

A major drawback of purely visual representations is that they cannot be easily processed in a programmatic fashion and their contents cannot be used in an automated way. Visualizing more semantic information usually has a negative impact on overall visibility and understandability and is often realized by links to external sources that provide the desired content. It is necessary to represent the information of these pathways in a machine-readable way. This enables automatic processing in computational models and methods.

2.2.2 Machine-readable Network Representations

Machine readability requires a predefined structure for describing the elements and their relationships. Biological entities are described as nodes of a network. Relationships are modeled as edges between pairs of nodes.

There are many different description languages for biological networks, which use different vocabularies and cover different aspects of the networks. They are all based on the de-facto standard for machine-readable languages, the *eXtensible Markup Language (XML)* [47]. With so-called `tags`, `attributes`, and `namespaces` it is possible to describe objects in a standardized manner.

The *Systems-Biology Graphical Notation (SBGN)* [48] is used to describe the topology and visual appearance of biological networks.

The language used by KEGG, the *KEGG markup language (KGML)* [49] is a proprietary format for their own pathway maps. It is rarely used or supported outside of KEGG.

The *Graph Markup Language (GraphML)* [50] is a versatile language for the generic description of graphs and networks. It is designed as an interchange format for all applications dealing with graphs and networks and does not use a custom syntax as some other XML-based markup-languages do.

The *Biological Pathway Exchange (BioPAX)* [51] language is based on the Resource Description Format (RDF) and is a Web Ontology Language (OWL). It is used to describe biological pathways and is meant to be a universal exchange format. The development of the BioPAX specification has been dormant for almost a decade and no new features have been introduced since.

The *Systems Biology Markup Language (SBML)* [52] is being developed and extended by the "Computational Modeling in Biology Network" (COMBINE) [53] initiative. It has an active community that steadily improves and extends the language with new features. It is widely used in systems biology methods, which play an important role in elucidating cellular processes that drive new discoveries for personalized medicine. Consequently, SBML is the description language of choice when searching for a source-format to build a network-resource application in this domain.

2.2.3 SBML

SBML was initially developed to describe mathematical models of biochemical reactions and the quantitative properties of biological systems. These processes are described by the core-module of SBML. The description language is an XML format with the top-level entry "sbml". It defines the XML namespaces and the prefixes of the SBML modules and versions used by the model. The namespaces allow different modules to have elements with the same name, since they will always be prefixed by the namespace they are defined in. Each namespace defines a prefix that is used throughout the rest of the document whenever an element is defined in that namespace. The only direct child element of the "sbml" entry is the "model", which is defined by a `name` and an `id`. The child elements of this "model" are described in more detail in the next section.

```
1   <listOfCompartments>
2     <compartment constant="true" id="extracellular" />
3     <compartment constant="true" id="intracellular" />
4     <compartment constant="true" id="nucleus" />
5   </listOfCompartments>
6   <listOfSpecies>
7     <species name="Delta" id="DELTA" compartment="extracellular" ... >
8     </species>
9     <species name="Notch" id="NOTCH" compartment="intracellular" ... >
10    </species>
11    <species name="CSL" id="CSL" compartment="nucleus" ... >
12    </species>
13    <species name="DNA" id="DNA" compartment="nucleus" ... >
14    </species>
15  </listOfSpecies>
16  <listOfReactions>
17    <reaction id="react_ex" name="Example Reaction" reversible="false">
18      <listOfReactants>
19        <speciesReference species="CSL" ...
20      </listOfReactants>
21      <listOfProducts>...
22      <listOfModifiers>...
23    </reaction>
24  </listOfReactions>
```

Listing 2.1: Specification of the SBML-core module entities: `compartments`, `species` and `reactions`

SBML Core Module

The SBML core module defines tags for `compartments`, `species` and `reactions`, which are used as the basic elements for the description of biochemical reactions and the spaces they occur in (see Listing 2.1).

They are organized in "listOf" XML-tags, which is a special listing type in SBML (see Listing 2.1, lines 1, 6, 16). `Compartments` describe the physical spaces in which the described reactions take place. There are three different `compartments` in the *Notch signaling pathway* example: the extracellular space, the intracellular space and the cellular nucleus, which is encoded in their "id"-attribute (Listing 2.1, lines 2–4). `Species` are used to describe simple chemicals, like CO_2, and biological entities like stretches of RNA or DNA and polypeptide-chains (Listing 2.1, lines 7–14). They have the mandatory attributes "name", "id" and "compartment", where the latter one references a previously defined `compartment` by its "id" attribute. Species themselves are referenced by their "id" attribute in `reactions`. Here, only a limited number of species is shown for demonstration purposes, while in reality, each entity and entity-state

in the model is defined by an entry of this type. The `reaction`-element (Listing 2.1, lines 17–23) contains lists of `reactants` and `products`. They link to previously defined `species` that make up the chemical compounds that take part in this reaction (see Listing 2.1, lines 18–21). Additionally, a `reaction` defines a list of `modifiers`, where each such modifier references a `species` that has been defined earlier in the document. In most cases, a `modifier` is an *enzyme* that catalyzes the reaction by lowering the activation energy needed for the reaction to take place. Each reaction partner is wrapped in a "speciesReference", which allows the definition of additional properties like "stoichiometry" (see Listing 2.1, line 19). Other important definitions in the core module of SBML that are important for the modeling aspect of systems biology are omitted here for brevity, as this work does not rely on them. The reader is referred to the specification-document[54,55] of SBML for details.

SBML Qualitative Models Extension

For describing qualitative aspects of biological processes, like regulation or signal transduction, the vocabulary of the language has to be expanded. SBML was designed with these kinds of expansions in mind. It is a highly extensible language where every entity defined in the core specification, including the model itself, can be extended with attributes or entirely new entities. For the description of qualitative aspects of cellular processes, SBML offers the *qualitative models*[56] extension. This extension defines two new elements, the `qualitativeSpecies` and the `transition`. A `qualitativeSpecies` refers to an entity, or a pool of entities that are indistinguishable, which are characterized by their qualitative influence on the processes or other `qualitativeSpecies` that are described in the model (see Listing 2.2, lines 1–7). They are given an "id" within the "qual" XML namespace by which they can be referenced by transitions. Not depicted are shared attributes with the `SBMLSpecies` entity from the core-model, like the compartment, which can be defined separately for `qualitativeSpecies` if needed. Optional attributes can be provided to declare initial and maximal levels of the described entity in the compartments.

The `transition` element (see Listing 2.2, lines 9–18) defines a list of `inputs` and a list of `outputs`. Each `input` and `output` refers to a `qualitativeSpecies` that participates in this `transition`. The kind of effect the `transition` has on these `qualitativeSpecies` is given by the attribute `transitionEffect`. Additionally, a `sign` attribute can determine whether the effect is positive, negative, both or unknown. The `qualitative models` extension further defines a `functionTerm` attribute. In the context of this thesis this attribute is not relevant and is thus omitted. The reader is referred to the official specification document[57] for details.

```
1   <qual:listOfQualitativeSpecies>
2     <qual:qualitativeSpecies qual:name="q_Delta" qual:id="q_DELTA" ...>
3     </qual:qualitativeSpecies>
4     <qual:qualitativeSpecies qual:name="q_Notch" qual:id="q_NOTCH" ... >
5     </qual:qualitativeSpecies>
6     ...
7   </qual:listOfQualitativeSpecies>
8   <qual:listOfTransitions>
9     <qual:transition qual:id="transition_1">
10      <qual:listOfInputs>
11        <qual:input qual:qualitativeSpecies="q_DELTA"
12          qual:transitionEffect="none" />
13      </qual:listOfInputs>
14      <qual:listOfOutputs>
15        <qual:output qual:qualitativeSpecies="q_NOTCH"
16          qual:transitionEffect="assignmentLevel" />
17      </qual:listOfOutputs>s
18    </qual:transition>
19  </qual:listOfTransitions>
```

Listing 2.2: Specification of the SBML qualitative-models-extension elements `qualitativeSpecies` and `Transition`.

2.2.4 Semantic Description with Ontologies

An `ontology` in computer science is understood as a catalog of properties, entities and relations for a certain field of study. It standardizes the use of certain terms for describing a system by defining one or more controlled vocabularies. The *Systems Biology Ontology* (SBO)[58] is primarily used in the mathematical modeling of biological systems in the systems-biology community, but also contains elements for other aspects of the cellular processes.

SBML incorporates the use of SBO terms in their specification by specifying an attribute called `sboTerm` that is optionally available on every entity defined (see Listing 2.3, lines 3, 6, 11). This allows modelers to use terms from the SBO to semantically describe the nature of entities and relations in their models. Table 2.1 lists some common terms from the SBO used in modeling of metabolic models, regulatory processes, and signaling cascades. The SBO is hierarchically organized, which allows the use of more general terms where either no specific one fits the entity, or where an entity can be of any of the subtypes defined in the `ontology`.

2.2.5 Entity Annotations using Qualifiers

Many entities in SBML models are annotated with additional resource information. This annotation is organized in the RDF format and provides qualification information to the annotated

```
<qual:listOfQualitativeSpecies>
  <qual:qualitativeSpecies qual:name="q_Delta" qual:id="q_DELTA"
    sboTerm="SBO:0000252" ... >
  </qual:qualitativeSpecies>
  <qual:qualitativeSpecies qual:name="q_C02" qual:id="q_C02"
    sboTerm="SBO:0000247" ... >
  </qual:qualitativeSpecies>
  ...
</qual:listOfQualitativeSpecies>
<qual:listOfTransitions>
  <qual:transition qual:id="transition_1" sboTerm="SBO:0000170">
    ...
  </qual:transition>
</qual:listOfTransitions>
```

Listing 2.3: Demonstration of the "sboTerm" attribute on the entities `qualitativeSpecies` and `Transition`.

```
<qual:qualitativeSpecies qual:name="q_Notch" qual:id="q_NOTCH"
  sboTerm="SBO:0000252" ... >
  <annotation>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:bqbiol="http://biomodels.net/biology-qualifiers/">
      <rdf:Description rdf:about="#meta_qual_NOTCH1">
        <bqbiol:hasProperty>
          <rdf:Bag>
            <rdf:li rdf:resource="http://identifiers.org/omim/164951" />
          </rdf:Bag>
        </bqbiol:hasProperty>
      </rdf:Description>
    </rdf:RDF>
  </annotation>
</qual:qualitativeSpecies>
```

Listing 2.4: Demonstration of the RDF annotation with biomodels.net biological qualifiers on a `qualitativeSpecies` entity.

Table 2.1: Selected examples for common terms from the SBO that are useful for describing cellular processes like metabolic reactions, regulatory events and signaling cascades. The Accession Number gives the term used in an SBML document, the name is provided by the ontology. The Definition column gives the first sentence of the definition as given in the ontology provided by the COMBINE initiative[59].

Accession number	Name	Definition
0000010	reactant	Substance consumed by a chemical reaction.
0000011	product	Substance that is produced in a reaction.
0000015	substrate	Molecule which is acted upon by an enzyme.
0000252	polypeptide chain	Naturally occurring macromolecule formed by the repetition of amino-acid residues linked by peptidic bonds.
0000247	simple chemical	Simple, non-repetitive chemical entity.
0000170	stimulation	Positive modulation of the execution of a process.
0000216	phospho-rylation	Addition of a phosphate group (-H2PO4) to a chemical entity.
0000656	activation	A conformational change in a protein resulting in its activation.

data. The COMBINE network actively maintains the Biomodels.net lists of *model qualifiers* and *biological qualifiers* for this purpose[60,61]. While elements in the first list are used to qualify the modeling concept of an entity, those in the latter one are used to qualify the biological object itself that is represented by an entity. An example of such an RDF annotation element can be seen in Listing 2.4. The “bqbiol” tag with type “hasProperty” provides a “resource” that gives qualified information about the annotated entity. In this case, the resource type qualifies as providing additional properties for the entity that cannot be described directly in SBML.

Table 2.2 shows some Biomodels.net qualifier names along with their description. By using these qualifiers on elements of biological networks their relationship to uniquely identifiable resources can be established. This allows the identification of the network entities themselves. In addition, the annotation objects can give more semantic meaning to the entities without the need for the model language to offer these specific attributes. For a complete list of Biomodels.net qualifiers we refer the reader to the respective COMBINE website[60,61].

Different systems-biology methods require different types of networks. Depending on which part of a biological system is to be analyzed, certain aspects are of greater interest than others. To focus on the important aspects, different types of networks are defined. Each type highlights a different layer of the biological machinery and allows different types of data to be

Table 2.2: Example qualifier from the biomodels.net qualifier list[60,61]. Qualifier name and a partial description are shown as provided on the website of the COMBINE initiative.

Model qualifier	
Qualifier name	**Description**
is	The modeling object represented by the model element is identical with the subject of the referenced resource.
isDerivedFrom	The modeling object represented by the model element is derived from the modeling object represented by the referenced resource.
isDescribedBy	The modeling object represented by the model element is described by the subject of the referenced resource.
Biological qualifier	
Qualifier name	**Description**
is	The biological entity represented by the model element has identity with the subject of the referenced resource.
hasVersion	The subject of the referenced resource is a version or an instance of the biological entity represented by the model element.
isEncodedBy	The biological entity represented by the model element is encoded, directly or transitively, by the subject of the referenced resource.

used with it. In the following section we are going to list example systems-biology methods and introduce the types of networks — so-called network mappings — that are inferred by these methods or used for computational analyses.

2.2.6 Network Mappings

Common abstractions of the biological processes are represented in *network mappings*. In these, the functional units and their interactions are limited to the respective viewpoint on these processes. This section briefly introduces the four main types of mappings and gives examples of systems biology methods that are either creating or using them.

Gene expression data is used in mechanistic modeling approaches to estimate the values of unknown states in the network. This type of analysis requires a *signaling mapping* of the network. In this mapping the nodes of the network are proteins or more general polypeptide chains and non-covalent complexes of these proteins or of proteins with chemical compounds. The edges of the network are directed and represent events that are associated with signaling events. Those include activation of an enzyme via a conformational change and the general positive and negative modulation of the execution of a process.

Scientists attempt to map protein interaction with methods like affinity purification-tandem mass-spectrometry and co-elution–mass spectrometry[62], protein-fragment complementation assays[63] or yeast two-hybrid screening[64]. The resulting network mapping is a *protein-protein-interaction (PPI)* network, where the nodes represent polypeptide chains and undirected edges connect any two proteins that are interacting with each other in any way. The type of interaction is not defined or sometimes unknown. If a complex of proteins is interacting with another protein, the hierarchy of the complex in this type of network is broken down, meaning that every individual part of the complex will be connected to the interacting protein.

To integrate metabolic processes with gene expression data, so-called `flux balance analyses`[65,66] are performed. A *metabolic mapping* is a bipartite graph with reaction partners in one node partition and the reactions in the other. The reaction partners are either simple chemicals when they represent substrates and products or are enzymes catalyzing the reaction. The nodes are connected with directed edges that indicate their respective role in the reaction.

Regulatory analysis can reveal upstream master regulator genes and infer gene regulatory networks from gene-wide expression profiles[67,68]. Those *regulatory mappings* show the effects of proteins on gene regulation and gene expression.

Those network mappings are the essential representations that are used in publications and computational analysis. A central network resource needs to offer easy-to-use interfaces for accessing those mappings and offer intuitive ways to work with them. Storing them in such a way that facilitates fast computations and efficient searches is another important feature. Those topics will be discussed in the next chapter.

2.3 Information Technology Methods

Three important concepts from information science are at the heart of SBML4j, the software presented in this thesis. The first is the so-called RESTful interface, a type of application programming interface (API) that a service-oriented software offers to potential client applications. The second is the graph-database backend, which SBML4j uses to store the network data. The third concept revolves around tracking the origin of data, the so-called provenance.

2.3.1 RESTful Interface

***Re**presentational **s**tate **t**ransfer (REST)* is a form of software architecture initially developed by Roy Fielding[69] for distributed systems, which exchange messages using hypermedia. Hypermedia is an extension of Hypertext, which was introduced by Theodor Nelson in 1965 as “a file structure for the complex, the changing and the indeterminate”[70]. Hypermedia extends the contents of the file structure beyond text-based information to any kind of media, like image data or video material.

Fielding's software architecture facilitates the communication between a client and a server. Each *representation* that a server delivers to the client contains links to other *representations* in the server application, for example a blog-post entry or a thread in a message board. By clicking such a link in a web browser, the receiving server is instructed to `transfer` this *representation* to the client, which is addressed by the link that was clicked on. It is important to note that no information about the client session must be stored on the server side in order to serve a request by the client. The interaction is then said to be `stateless`. Each request contains all necessary information for the server to understand the request.

The *Hypertext Transfer Protocol (HTTP)* [71] is a well-known protocol for this kind of communication. It defines a set of methods that a client can issue to a server in the form of `requests`. The four main types of interest are `GET`, `POST`, `PUT`, and `DELETE` operations. They are issued with a so-called uniform resource identifier (URI), which specifies the resource that is subject of the request. In the web-page example, this URI would be the uniform resource locator (URL) of the link that is clicked in the browser.

Each such request consists of the path to the URI, followed by optional query parameters. These query parameters are separated from the path by the question mark symbol and multiple query parameters are separated from each other with the ampersand character. In addition, the `POST` and `PUT` requests contain a body element, which can consist of xml or json-formatted text, or binary data, like image-files or other documents that are to be uploaded to a REST service.

The simplest operation is the `GET` request. By issuing a `GET` operation a client requests a resource from the receiving end. The resource to be fetched is identified by the URI of the request.

A `POST` operation transfers an entity enclosed in the request from the client to the server. The server is expected to accept this entity as a subordinate of the resource that is given by the URI of the request. This requires that the provided URI must point to a resource that is capable of handling the provided entity. Such a request usually stores the entity in the internal storage or database of the server. It can then be accessed using a newly provided URI or as part of the URI of the initial request.

The `PUT` operation is similar to the `POST` operation in the sense that an enclosed entity is uploaded to the server. However, the `PUT` request identifies an exact resource on the server side in the provided URI with the instruction that "the enclosed entity should be considered as a modified version of the one residing on the origin server" [71]. In case the provided URI does not denote an already existing resource, it can be created by the server instead.

The `DELETE` operation instructs the server to remove the resource identified by the provided URI of the request. Whether the server actually deletes the identified resource or just marks it as deleted and hides it in future requests is implementation specific. Future `GET` requests that would contain the undeleted entity as part of listings are required to hide it in any case.

```
1  MATCH p = (start:Protein)-[rel:activation|inhibition *..3]->(end:chemicalCompound)
2  WHERE start.name = "Delta"
3  RETURN p
```

Listing 2.5: Example Cypher Query. Query for retrieving a path with directed relationships with labels *activation* or *inhibition* from a start-node with label *Protein* whose name property has the value "Delta", and an end-node with label *chemicalCompound*. The paths consisting of nodes and relationships that match these criteria are returned.

The operations `GET`, `PUT` and `DELETE` share a property called `idempotence`. This denotes that any number of identical requests these types must result in the same side effects as a single request of that type. Side effects mean any changes of state on the server side that affect other resources or would change the response to this or any other request. The same cannot be required from a POST request, as such a request usually results in the creation of a new resource on the server side. Multiple identical requests for the creation of a resource can result in the creation of multiple identical resources, which only differ in their identifier. The side effects of multiple such requests change the response to `GET` operations that list these resources.

An API that implements these protocols and adheres to their restrictions is said to be *RESTful*. Such *RESTful* APIs are often used in service-oriented software architectures, where multiple services dedicated to individual tasks are combined to perform a larger unit of work. This is usually the design of choice in bioinformatics pipelines and analysis settings. A popular description language for RESTful APIs is the OpenAPI standard[72].

2.3.2 Graph Databases

Graph databases are database systems that use graph structures to represent and store data. Individual data points are represented as nodes and their relationships are modeled as edges that connect these nodes as shown in Fig. 2.3A. The example illustrates how parts of the *Notch signaling pathway* could be stored in a graph database. The data is stored as a so-called *labeled property-graph*. The nodes and relationships are *first-class citizens* in the database, and thus can have labels and properties attached to them. This means that the types of nodes, (e.g., *Protein*) or relationships, (e.g., *inhibition*) are implemented as *labels* (see Fig. 2.3B), which can be used in queries to determine the types of nodes or relationships. All additional attributes on nodes or relationships are *properties*, which are realized using key-value pairs (see Fig. 2.3C). Storing the data as a graph enables queries that follow the natural path on the graph data structure. Listing 2.5 shows an example of a query in the *Cypher*[73] query language that is used by the graph database *Neo4j*[74].

It starts with the keyword **MATCH**, which indicates a search query. Then a path is defined, which is given the name "*p*". The path starts with a node of type *Protein*, where *Protein* is a

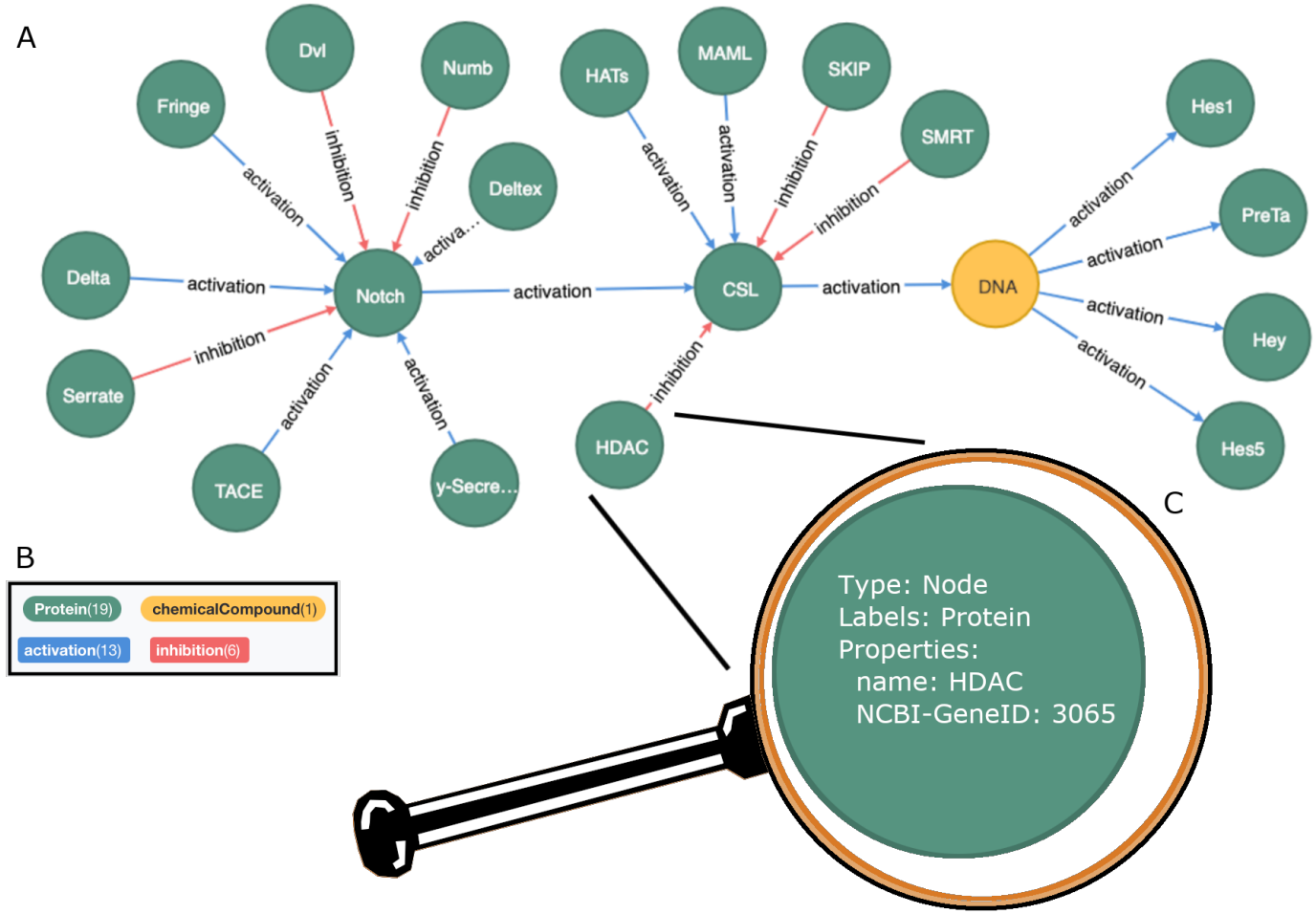


Figure 2.3: Graph database representation example. Possible representation of parts of the Notch signaling pathway. A) Visualization by the Neo4j browser of the nodes and edges stored in the database. B) Legend from the Neo4j Browser for the visualization. Rounded boxes indicate node labels, rectangles indicate relationship labels. The number of respective entities is given in parentheses. C) Detailed view of the labels and properties of the node with name "HDAC".

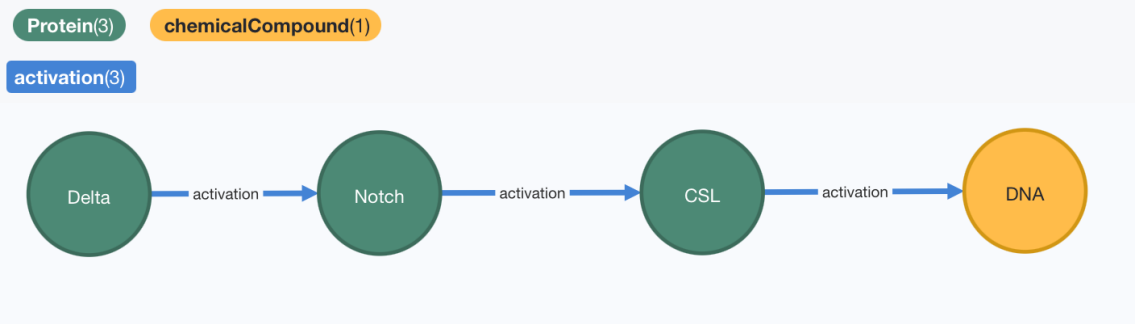


Figure 2.4: Example query result of the query in Listing 2.5. Visualization of the query result from the Neo4j browser including legend for node and relationship labels. Nodes are named using the *name* property of the node entity, relationships are named with their assigned *label*.

label on the node. The parentheses indicate that the first element of the path is supposed to be a `node`. The word "*start*" before the colon specifies the name of this node for the rest of the query. The colon indicates that a mandatory label follows. The next part in square brackets defines a directed relationship, where the ">"-symbol gives the direction. After the colon the required relationship labels are provided, which are connected with a logical "*OR*" in the form of the "|"-symbol. This query searches for relationships of type "*activation*" or "*inhibition*". The character sequence "∗..3" denotes that up to three relationships may lie between the start and end node of the requested path. The relationship ends in a node of type *chemicalCompound* and is assigned the name "*end*" for this query.

The **WHERE** clause is used to limit the search to certain properties of nodes or relationships, in this case to start in a node whose name-property value is equal to "Delta".

The **RETURN** statement defines what the database should return as the result set. In this example, the result set contains all paths that satisfy the query conditions including the start node, the relationships, and the end node. The result of this query on the graph from the *Notch signaling pathway* example can be seen in Fig. 2.4. Note that the output is also a graph that contains nodes and relationships from the result set. The visualization consists of the nodes and edges from the query result and is configured to show the name properties for the nodes and the labels of the relationships. The color assignment of the respective types can be seen in the legend of the figure.

With these features a graph database can offer more elaborated access compared to traditional relational databases. A graph database can query data along the relationships defined on the graph and extract all related entities to a requested node. Similarly, it is possible to write new data into the database, as related entities and their relationships are persisted alongside the initial entity. This makes graph databases ideal candidates for storing and working on biological networks since the structure of the data directly matches the structure of the database system. Additionally, nodes can have multiple labels defined on them simultaneously, which

allows them to be part of multiple subsets or levels of granularity at the same time. This is a vital feature for managing data on the network level, on a warehousing level, and on the level of keeping track of the origin of data in the database.

For advanced computational power, the Neo4j database offers procedures that can be executed directly on the data in the database core. The Awesome Procedures on Cypher (APOC)[75] is an add-on library for Neo4j, which makes extensive use of this ability. It provides many procedures and implementations of graph algorithms, which can be used in custom Cypher queries.

2.3.3 Provenance Tracking

The term `provenance` in the computational sciences refers to the origin of data and the formal representation of all consecutive steps performed on this data by computational processes[76].

In bioinformatics, the `process provenance` describes the sequence of transformative processes that were applied to a set of source data. The standard means of documenting the steps of this sequence are workflow managers and their respective description languages, such as *Nextflow*[77], *KNIME*[78] or *Snakemake*[79]. They allow the documentation of analysis-tool metadata, parameters as well as input and output data. At the same time, they manage the execution of the correct tool with the matching data. When a publication includes these workflow files and offers access to the original raw data, it enables other researchers to reproduce the data analysis part of a study.

For biological networks these processes are oftentimes not as well documented and only the resulting network is provided in publications[80]. This is largely because many steps are performed manually and can therefore not be documented by the bioinformatics workflow language. The same is true for the clinical setting of MTBs, where the presentations for the expert meetings are created by hand and simply stored as static images.

This emphasizes the need for a network resource that tracks the provenance of internal and external modification steps alongside the networks themselves. To enable this tracking, the data model of an application needs to incorporate structures for storing this kind of information. The *World Wide Web Consortium* (W3C) put out a set of specifications named **PROV** that enables the publication of provenance information and was designed for use in the World Wide Web[81].

The W3C PROV Data Model

The W3C PROV data model[82] defines three basic core concepts, an `Agent`, an `Activity`, and the `Entity`. In addition, relationships between these concepts are defined, as can be seen in Fig. 2.5.

An Entity is any element in a system whose provenance is to be tracked. An Activity represents any process that either `generates` or alters the state of an Entity by `using` it. The

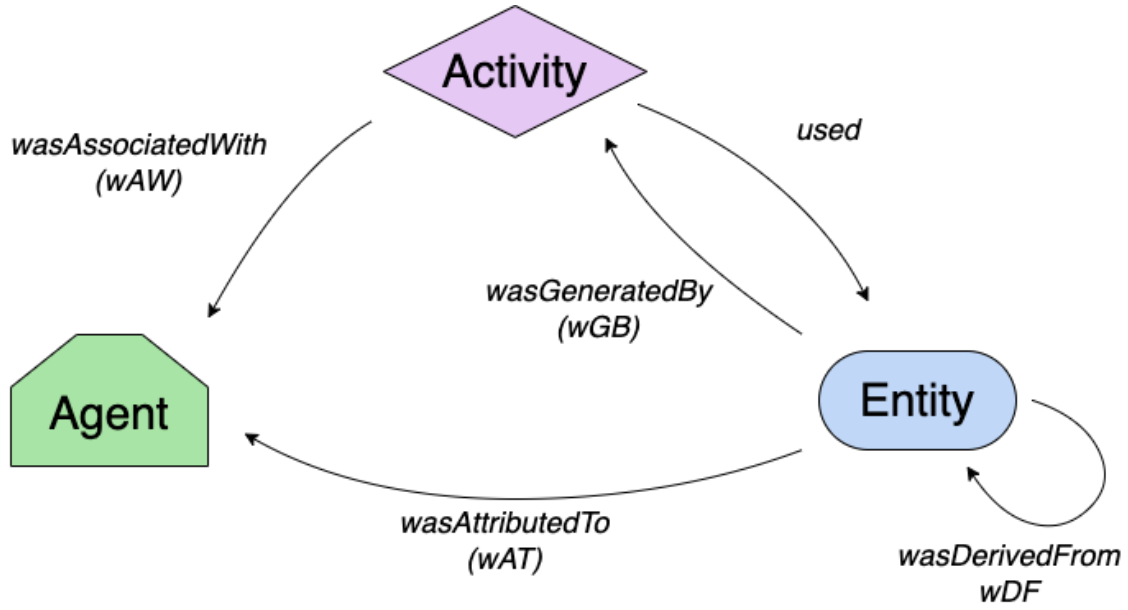


Figure 2.5: Core concepts of the PROV data model with its main relationships. Entity, Activity and Agent denote the core concepts to describe provenance in information systems. An Entity in a certain state is generated or used by an Activity, whereby the Entity gets derived from another Entity or Entity-state. An Agent gets associated with the Activity and the new Entity is attributed to that Agent.

Entity thereby gets `derived from` some other Entity, which can also be the same Entity in a different state. The Agent indicates the user that `was associated with` this Activity and to whom the Entity in its new state is `attributed to`.

With the implementation of the W3C PROV data model in the graph database, it is possible to store the network provenance alongside the networks in the database. This offers the unique opportunity to provide a resource for biological networks that keeps track of and allows the extraction of network provenance alongside the network data. It can be seamlessly integrated in existing infrastructure via provided RESTful interfaces, which facilitates complete reproducibility of biological networks in research and clinical settings alike. The details of the implementation of the software SBML4j will be discussed in the next chapter.

Chapter 3

SBML4j – A Service Hub for Reproducible Biological Networks

3.1 Introduction

Biological networks are essential for understanding mechanisms in biomedical phenomena. They explain the relationships between individual biological components, the genes, proteins and chemical compounds that make up the cellular apparatus. Often it is these relationships, like interactions between proteins, stimulation of enzymatic activity or regulation of gene expression, that govern the onset and progression of diseases. Typically these networks are collated from literature or based on experimental findings and predictions and made available in comprehensive databases. A wealth of tools is available to manipulate and visualize them. Due to their size and complex nature, network data is best managed in database management systems. In the past, relational systems were used to organize any kind of data in tabular form. To answer biomedical questions, many individual elements and their relationships need to be evaluated, which requires complex queries on these tables. A graph-based data architecture promises better performance and more comprehensible querying mechanics.

Here we present SBML4j, a service-oriented application that leverages the efficient management and querying of graph data in a modern graph database for the analysis of biological network data. SBML4j relies on SBML as a standardized representation of biological network data and renders these networks accessible in an efficient manner through a RESTful interface. In addition, it provides an easy to use Python client library to query the database and manipulate query results.

SBML4j can be deployed conveniently through prebuilt Docker containers, making it available on all major platforms. The interface permits graph-based queries, for example shortest path queries or adjacency queries. Additional scalar and categorical data (e.g., expression

levels, predefined pathways) can be mapped to the network through the REST API as well as via the Python client library.

A special focus has been put on tracking the provenance of the networks. SBML files that are uploaded via the RESTful interface are integrated into a biological knowledge graph and the origin of the created entities is recorded. During the creation of a specific network mapping from this knowledge graph the chosen parameter settings are tracked and stored alongside the mapping data. For each processing step that alters a network the REST request is recorded and parameters as well as data are stored alongside a new instance of the network. When external tools are used to manipulate or process a network retrieved from SBML4j, the resulting network can be reuploaded into the database management system. SBML4j offers specific REST endpoints that allow the addition of custom provenance information regarding the external tool and parameters or data used for the processing step. This way the complete provenance is tracked in the SBML4j database and is provided in addition to the GraphML representation of the network. This enables reproducible biological networks, where each analysis step is recorded and every intermediary network is available for inspection.

Related Work

Many public repositories for network and pathway related information can be found online, like the String database, KEGG, the Biomodels Database[83] and Wikipathways[84]. These databases are used widely and provide a comprehensive view of many physiological processes.

The Network Data Exchange, NDEx[85,86], is an online resource where scientists can upload their own networks, share them with collaborators and publicly distribute them. By linking an uploaded network with an already existing one a provenance connection between the two is established. However, this does not document the details of the changes or the methods applied to create the one from the other. Resources like Reactome[87] or Recon[88], for example, provide organism-wide networks containing expert-curated cellular processes. They regularly update their representations with new data but do not disclose the provenance of their networks in a machine-readable way. The SEMS Project[89] explored how to use a graph database to store biological models and accompanying simulations with their *Management System for Models and Simulations — MaSyMoS*[90]. Due to a lack of funding, their project is now dormant.

Software tools like SBGN to Neo4j (STON)[91] and Recon2Neo4j[92] aim to add extended query features to these kinds of networks by transferring them to local graph database instances. They use custom translations of the entities and relationships and give users the ability to explore the contents using the Cypher Query Language. Lysenko et al.[93] demonstrated the feasibility of this approach by using the Recon2Neo4j framework to fill a graph database with assembled data from various sources. They successfully showed how to use Cypher to generate new hypotheses from the graph data model. These frameworks rely on hand-crafted input

files to populate their databases, which makes tracking of its origin difficult. Additionally, data in these frameworks is only accessible via direct database queries. To generate new hypotheses from these data sources, knowledge about the Cypher query language as well as the underlying database schema is necessary. Proprietary, commercial tools like MetaCore[94] promise to fulfill all needs of industrial professionals regarding network based analysis but are not freely available to the research community and are therefore not suited for scientific workflows.

The network-based navigation and analysis tool BiNA[95] with the corresponding network resource framework UniPAX[96] aimed to provide a scalable system for analyzing data in a network context. The software performed well but failed to transparently track and report network provenance. In addition, the project fell victim to the often observed loss of knowledge in the scientific community when one generation of researchers leaves a group and the next generation fails to reinstate the dormant software.

To enable reproducible results the documentation of the exact transformation steps taken to integrate the data as well as the queries to extract knowledge from the database is of utmost importance.

Motivation

We address these shortcomings with SBML4j by reading biological models in the well established SBML format as input source. By replicating the structure of this standard in the knowledge graph data layout the data remains understandable for researchers proficient in SBML. Additionally we are deriving easy to understand network mappings from this graph structure which enables reproducible programmatic access to the networks using a RESTful interface. SBML4j makes extensive use of the SBO terms and the biomodels.net qualifier information that are found in the SBML models. This means that a scientist who is familiar with the SBML syntax can explore the knowledge graph directly on the database using Cypher and the built-in web-based browser. In addition, we implemented a well-documented RESTful interface to query for information in the SBML structured data and to build, extend, annotate and explore network mappings from this knowledge base. All created networks are reported in the versatile GraphML format which can be processed by many tools and be visualized using open source software like Cytoscape[97], commercial software like yFiles[98] or the freely available Jupyter[99] widget ipycytoscape[100]. We provide a python client library, pysbml4j[101], which offers intuitive access to the full functionality of the RESTful interface. It enables script-based automatic loading of models, creation of collections and network mappings. It offers methods for annotating network nodes with data from CSV files which allows the use of classical Excel-based data in network analysis as well as JavaScipt Object Notation (JSON)-based annotation of nodes and relationships. In the following sections we present the design and implementation of

the software SBML4j and all of its components as well as the python client library pysbml4j. We will demonstrate the viability of the service-oriented approach and its easy integration by showcasing the creation of a knowledge graph built from the KEGG pathway database. We show how to create network mappings from this knowledge graph and how we can use different mappings to answer real world biological questions with just a few lines of python code. Finally, we demonstrate the provenance-tracking mechanism and show how SBML4j enables reproducible biological-network analysis for research and the clinical settings.

3.2 Design Considerations

In software engineering certain considerations have to be made before implementation can begin. First, the goals of the software have to be formulated. Then, to reach these goals, specific choices regarding base technologies must be made that influence the development substantially. Rated among the most important are the choice of the database backend as well as the programming language.

3.2.1 Design Goals

A software that offers versatile persistence, exploration and provenance tracking of biological networks, lends itself to be an essential building block in a broad spectrum of disciplines. It must therefore stay open to new developments and provide a solid foundation for future extensions and use-cases. The software should reflect those needs and the design should strive to reach the following goals.

Standardized Source

The first major goal of this software is the ability to read in biological models from a standardized input source that is already adopted in a large proportion of the target audience. It should also be able to integrate multiple models together in a knowledge graph and create network mappings from this graph which can then be provided as network resources. No prior or additional knowledge should have to already exist in the database or at application level to build a network resource from these source-models. Only information present in the models and data that can directly be retrieved from online-source with the information provided in the models should be taken into account when building the knowledge graph. As we laid out in Section 2.2.2, SBML is the most widely used format in systems biology with an active community. Since SBML is a highly extensible language, this software needs to be able to deal with these extensions, if needed. To be able to map regulatory, signaling and protein-protein-interaction networks the SBML extension “qualitative models” must be supported from the start. In addition, many available visualization tools for network data and programming libraries

dealing with networks operate on the versatile GraphML format. All network exchanges should therefore be performed via GraphML.

Accessibility

The second major goal is the possibility to access the networks managed by the software in a way that does not require complex database queries. No knowledge about the internal workings of the database or software itself should be needed to obtain or perform an operation on a network. The most broadly used means of access in distributed knowledge systems, of which SBML4j should become a part of in many settings, is a RESTful interface. All operations on the networks should be made available via REST endpoints. Since many research settings favor automated analysis processes over manually performed steps, a means of scripted access to this interface should be developed and provided alongside the main software. This should be realized in the popular language `Python` and the goal is to give access to a network and any offered method on or with this network in just a few lines of code. In addition, the database layout should still enable manual queries using a suitable query language to facilitate complex use cases and manual inspection of the data.

Reproducibility

The third major goal is to enable the complete tracking of data provenance for biological network data. This is a vital building block towards FAIR data principles, which any new software should adhere to. For any network that is retrieved from the software a complete history should be available that tracks the network's provenance from the input-sources to the final product. If provenance cannot be tracked automatically for a specific step, the software should provide means to manually add provenance-information to the documentation of the performed step. In addition, any network that was the result of an intermediate step in the creation of one specific network should be retrievable and its data provenance transparent.

Readability

This software is developed in a scientific research setting, where doctoral candidates or post-doctoral researchers that work on such a software are only assigned to this project for a limited amount of time. This can often lead to limited support after launch and a loss of knowledge about the software when the initial developers leave the working group. It is therefore essential to build a code base, which can be picked up by the next generation of students without the absolute need for a knowledgeable peer around to teach them. To achieve this, the documentation needs to be extensive and kept up-to-date. To further this development with design decisions, the source code must be understandable and documented well enough to enable readability and insight into the functionality. To facilitate this, variables and method

names should be chosen to be non-cryptic and self-explanatory. A name of a function should preferably be spelled out to convey the purpose it is serving. Variable names should describe the content they contain and, in places where it is beneficial to the understanding of the code, have suffixes added to them, which denote their type. This ensures that the source code can be read and understood by the future generation of scientists that aim to improve on the work presented here.

3.2.2 Choice of Database

The first choice to be made was which database should be used for the persistence of the models, the creation, management and exploration of the networks and the provenance tracking. For instance, UniPAX made use of an object-relational mapping for object persistence. This allowed for a flexible backend implementation of a relational database on the one hand, but, on the other hand, resulted in a complex relational database-schema. In doing so UniPAX was able to store its data in an Oracle database or a MySQL database, depending on user choice, which also resulted in programming overhead to accommodate both SQL implementations. To generate a single network representation, complex table joins and index traversals where needed.

In contrast, graph databases store the network data in a graph format, which enables data queries based on relations of objects. This mode of database access especially facilitates the work with and operation on biological networks. As Recon2Neo4j has shown, graph databases are able to closely represent the entities and relationships encoded in an SBML model. The MaSyMoS application used a graph database to relate different models to one another as well as with ontology information. These examples demonstrate the viability of graph databases for storing biological network information.

The Neo4j graph database has widely available database drivers in a number of programming languages and frameworks, which makes it a suitable pick for development purposes. Neo4j is already used in many production environments, for instance for fraud detection [102] or drug-repurposing [10]. This proves the suitability for the future productive use of SBML4j in the clinical context.

3.2.3 Choice of Programming Language

The second choice to be made is concerned with the programming language to use. The first requirement for this decision was the ability to process SBML files. There are two main implementations of the SBML standard, libSBML and jSBML for the C programming language and the Java programming language respectively. Both implementations are suitable for processing SBML core models as well as those using the qualitative models extension, which we need to support in order to process non-metabolic models.

The second requirement concerns the availability of a database driver, which was a crucial factor in the decision for or against a certain programming language. At decision time no stable C/C++ database driver for neo4j was available, which made this language not feasible for this project despite potential performance benefits of C++ over Java. Neo4j provides a well-documented Java driver. The *Java Spring* framework[103] features a fully functional data module that communicates with the database using the provided driver. It offers graph-based enhancements to common repository elements. The integrated object-graph mapper (OGM) enables a direct mapping of database structures to the domain model and vice versa. The *Java Spring* framework therefore provides a stable, extensible and well tested framework for working with the neo4j graph database.

3.2.4 Organizational Layout

As a service-oriented network resource with full provenance-tracking, SBML4j sets out to handle various tasks.

The first task is to persist biological models from multiple SBML files and to create an integrated knowledge graph. A model representation in the database is termed a `pathway` in SBML4j, multiple such pathways can be combined to collections. The union of all pathways builds the knowledge-graph in the `SBML` layer.

The second task is the creation of `network mappings` from this knowledge-graph or subgraphs thereof. This is achieved by `flattening` the relationships and hierarchies in the models. Those flat network mappings are the core data-structure that is available to clients of the SBML4j service. Certain `warehousing` elements are required to track the different networks and provide inventory-type information about them.

The third task is to provide diverse functionality for these networks. These include general operations, like filtering a network by type of entity and annotating the nodes or relationships with arbitrary data. In addition, popular graph algorithms should be offered for use in analysis, like the calculation of shortest paths, or the generation of neighborhood networks.

The fourth task is to track each of these operations, store this `provenance` information as metadata alongside the network data and provide a detailed provenance report for each network per user request. This includes provenance tracking even for network-based tasks that are performed in external applications, where the source networks are obtained from SBML4j, or the results networks are provided to SBML4j.

To facilitate these tasks, the application and the data model are divided into organizational units. The main units are `SBML`, `Network`, `Warehouse`, and `Provenance`.

From a graph perspective on the database level, this layout creates three main graph subtypes: The provenance graph, the warehouse graph and the content graph. The latter one combines the nodes and relationships from the SBML knowledge graph and the contents of the

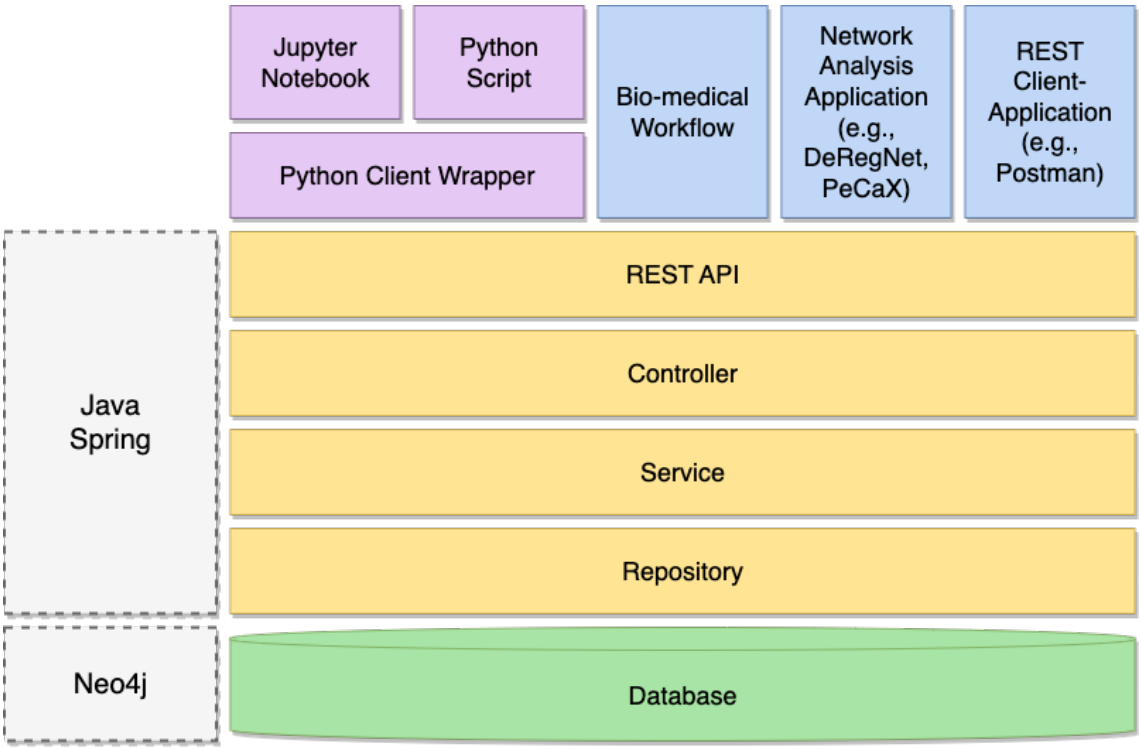


Figure 3.1: Architecture of SBML4j with client applications and database backend. The SBML4j application (yellow) consists of the REST API and the application layers for controller, service and repository interfaces. The database backend (green) is accessed by the repository interfaces and stores all data, metadata and provenance information in a single graph. Clients can directly issue HTTP requests (blue) or make use of the pysbml4j client library for python (purple) to communicate with the REST API. Application and database are realized with the Java Spring framework and the Neo4j Graph database, respectively.

network mappings. They are typically not requested in the same request and the distinction will be made depending on the connection of these contents to the respective type in the warehouse graph. By limiting queries to specific subsets of node and relationship types, the different subgraphs can be selected and their contents extracted.

This organizational layout can be found throughout the rest of this chapter where the data model, the services and the controller functionality are presented.

3.3 Implementation

3.3.1 Architecture

The overall structure of the SBML4j ecosystem is shown in Figure 3.1. It consists of the core application, the independent Neo4j graph database instance, and the Python client library

pysbml4j. Additionally, examples of client applications that directly interact with the REST interface of SBMl4j are included for illustration.

The database holds a single database graph. It contains all models that are uploaded to the service, which are combined to a knowledge graph as well as all created network mappings with user-generated networks and all warehousing and provenance-tracking information. This graph is composed of entities and relationships which reflect the classes of the application data model. They are mapped via the OGM of the Spring Data component. This data model is described in detail in Section 3.3.2.

The SBML4j core application consists of four layers, which are implemented in the Java Spring framework. The lowest layer contains the `repository interfaces`, which manage the persistence and retrieval of elements from the underlying graph database. All repository interfaces extend the *Neo4jRepository*, which adds special graph-based query features to the standard operations `create`, `read`, `update` and `delete` that are found in traditional interfaces for relational databases. The communication between the `repositories` and the `database` use the Neo4j `Bolt` protocol[104]. Details about the repository interfaces can be found in Section 3.3.3.

The second layer consists of the `Services`. Each service provides functionality for handling entities and relationships, manipulating the database-graph structure or performing other tasks required to fulfill requests by a client. The structure of the services and selected details are described in Section 3.3.4.

The third layer consists of the `controller classes` that implement the API interfaces that are defined by the `REST API`. These classes deal with general user handling and control the sequence of steps that are needed for each individual endpoint. The general outline of these tasks is laid out in Section 3.3.5.

The topmost layer contains the `REST API interfaces`. These interfaces define the endpoints that are available for the interaction with SBML4j. The API is described in detail in Section 3.3.6.

The architecture also includes potential client applications that communicate with the REST API directly by sending HTTP requests. These can be general biological, medical, or other workflows that create, manipulate and request networks. Other, more specific, examples include *DeRegNet*[105], an application for calculating maximally-deregulated subnetworks (Chapter 4), *PeCaX*[?] , the *Personalized Cancer and Network Explorer*, which uses SBML4j for network-neighborhood creation, annotation and storage (Chapter 5), and other REST client applications for manual interaction with the API, like the Application *Postman*[106].

Another method for making use of the features of SBML4j is given by the python client library `pysbml4j`. It offers convenient python methods for all available REST functions. The `pysbml4j` client library is described in detail in Section 3.3.8.

3.3.2 Data Model

The data model used by SBML4j is a combination of a hierarchical model and an entity graph model (Fig. 3.2). In entity graph models each entity in the data model is represented by a node or relationship in the database graph. This mapping is done automatically by the OGM, which is part of the Spring Data package. Annotations on the Java classes denote whether an entity is mapped to a node or relationships, as well as the name of the graph element. The hierarchy is introduced to be able to separate different aspects of the data into different layers and in order to uniformly define common attributes on different types of nodes and relationships.

The base layer consists of a single entity, the `GraphBaseEntity`, responsible for general graph-database specific attributes, like `id` and `version`. As every other entity in SBML4j inherits from this base entity, all entities can be persisted in the graph database and can be managed by the OGM. At the core lies the PROV data model, whose basic entity is the `ProvenanceEntity`, which directly inherits from `GraphBaseEntity`. All further entities in the class hierarchy inherit from this entity, which provides the basic features for tracking the provenance. This means that every entity in the database has one of the three provenance types, `Activity`, `Agent` or `Entity`. While the `Activity` and `Agent` entities are specifically tracking the provenance and user relationship to the data, the `Entity` type denotes all entities that are to be tracked by the PROV data model. These entities are divided into the `Warehouse` and `Content` graphs. The warehouse graph entities have the purpose of keeping an inventory of available data. The `Pathway` and `PathwayCollection` entities bundle all entities from the SBML layer, grouped by the original SBML models, and user-defined collections of these models, respectively. Each such model is derived from an input file, represented by the `FileNode` entity, which itself is derived from a certain source. This source is represented by the `DatabaseNode`. Additionally, each SBML model, and thus each `Pathway` is associated with a specific organism, which is captured by the `Organism` entity. The `MappingNode` entity is the inventory entity for network mappings, which were created from one or more SBML models.

The `Content` graph is divided into the SBML layer and the `Network Layer`, which contain the SBML knowledge graph and the flattened network mappings, respectively. The SBML layer consists of the core-model entities and is based on the SBML specification model, albeit in a simplified version for the purpose of this application. The main entities of the core model are `SBMLSpecies` and `SBMLSimpleReaction`, which together represent the metabolic aspects of the biological models. Multiple `SBMLSpecies` can form a group, which are indicated in the SBML model as being of type *group* and are represented in SBML4j as the entity type `SBMLSpeciesGroup`. The compartments in which these metabolic reactions take place are represented by the `SBMLCompartment` entity. Species and reaction entities are derived from the `SBMLCompartmentalizedSBase` type, which holds the reference to such one compart-

Figure 3.2: Overview of the SBML4j data model. Data model classes of SBML4j in an inheritance graph. Node shape denotes entity type, combined arrows give the direction of inheritance from a common ancestor. Color of nodes indicates the layer to which this entity belongs.

ment and which itself inherits from the SBMLSBase entity. The SBMLSBase defines the basic attributes of the SBML model.

In addition, the two main entities from the *qualitative species* extension are included in this layer. The SBMLQualSpecies is the qualitative representation of the SBMLSpecies entity. It is also derived from the SBMLCompartmentalizedSBase entity and can be grouped in SBMLQualSpeciesGroup entities, too. The SBMLSimpleTransition entity describes non-metabolic processes, like signaling events through chemical interaction, regulatory events or general protein-protein interactions. To represent the *biomodels.net* qualifier annotations, three additional entities are part of the SBML layer. The ExternalResourceEntity represents the annotation elements, which are typically in the form of links, or URIs to an external source giving detailed information about the annotated entity. The BiomodelQualifier relationship entity is used to encode the type of relationship that exists between the entity in the model and the external qualifier. For all external resources that get resolved and the external web service queried for additional data (e.g., the KEGG REST API), NameNode entities are created for all secondary names found for the connected entity.

When creating network mappings from the knowledge graph that is formed by the SBML layer, the hierarchies of entities in the models get flattened. This is why the node and relationship entities in the Network Layer are called FlatSpecies and FlatEdge, respectively. One FlatSpecies represents one set of ExternalResourceEntities that describe one biological entity, or multiple variants of said entity that takes part in one or more metabolic and non-metabolic activities. These activities are subtypes of the FlatEdge entity, which connect the nodes in the flattened network mappings.

The FlatEdge relationship entity is one of the explicit relationships defined in SBML4j, next to the BiomodelsQualifier relationship and the WarehouseGraphEdge. The latter one is used to link elements from the content-derived layers (i.e., SBML layer and Flat layer) to the warehouse layer, showing which contents are contained in the different pathways and networks that can be accessed via the RESTful interface. All other relationships, for instance the affiliation of an SBML entity to a compartment, are implicit. They are defined on the node-entity by reference and are built on the database upon persistence of the nodes.

Explicit relationships can be accessed through the OGM independently from the node entities they connect, while implicit relationships have to be extracted using the node entity, which holds the implicit reference to another entity.

In addition to the entities shown in Figure 3.2, the data model includes definitions of API request and response bodies, such as the inventory items for networks and pathways, the annotation and filter items for manipulating networks as well as the components of the provenance-report response. The section concludes with a brief introduction of enumerations and exceptions, which are defined in SBML4j.

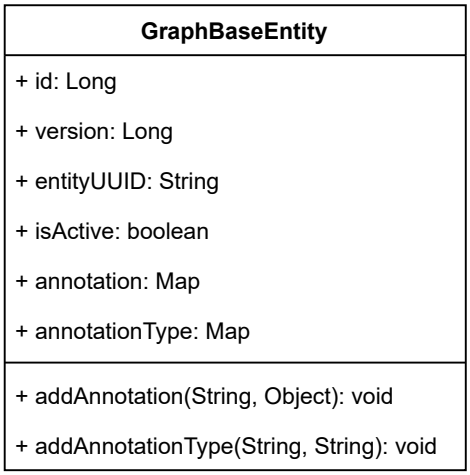


Figure 3.3: UML Diagram of the GraphBaseEntity. This basic entity holds the common attributes for each node or relationship, the *id* and *version* fields required by the database, the *entityUUID* as global unique identifier of this entity across all layers, the *isActive* flag to allow deactivation of parts of the graph, and two maps for storing annotation data for this entity. Additionally, custom methods for adding individual annotations to an entity are included. Standard methods, like getters and setters are omitted for brevity.

Base Layer

The most basic graph element is the `GraphBaseEntity` (Fig. 3.3). It holds the database-curated *id* and *version* attributes of type *java.lang.Long*. The *id* attribute is used by the database to identify individual entities and reconnect them to the graph that is stored in memory. It is unique to one database instance, but is not guaranteed to point to the same entity throughout the lifetime of the database. Deleted *id*s will be reused after a certain period of time. The *version* attribute is an optional database-managed attribute that will be used to support *optimistic locking*. This mechanic is useful when multiple threads access the data in parallel and try to modify the same entity. On a collision of the version number the *OptimisticLockingException* will be thrown allowing the implementation to resolve this issue. For instance, this can be done by refetching the entity and reapplying the desired modification before attempting to save the entity again. SBML4j offers support for *optimistic locking* by using the *version* attribute on the base class. Since all entities that are used in SBML4j are supposed to be stored in the neo4j database, all objects need to be derived from this base class.

In addition, the `GraphBaseEntity` object contains the *entityUUID* attribute, which is used as the primary key to uniquely identify any database object. In contrast to the database-managed *id* attribute, the *entityUUID* is managed by SBML4j and needs to be generated using the `GraphBaseEntityService` method *setGraphBaseEntityProperties*, which will generate a new *UUID* using the generation strategy *randomUUID* from the *java.util.UUID* package. With an index on the `entityUUID` attribute across all entities, this is the primary means of retrieving

elements from the database. The Boolean *isActive* can be used to deactivate any object and is set to *true* when the *entityUUID* is generated. The `GraphBaseEntity` also holds two Java HashMaps *annotation* and *annotationType*, which are used to store arbitrary data on any of the database objects. We store the datatype of each element in addition to the actual annotation as those data are converted to *String* representations when persisted in the database. When the `GraphBaseEntity` is retrieved from the database again, the annotation elements need to be cast into their appropriate type if they are to be presented to the user or used in calculations. Without the help of the *annotationType* attributes this would be more time consuming as several types need to be checked rather than fetching the appropriate type from the HashMap and converting it accordingly. For convenience we provide `addAnnotation(String, Object)` and `addAnnotationType(String, String)` methods to add an annotation and its type respectively. These methods will initialize the underlying HashMaps when the first elements are added and they have not yet been initialized.

Provenance Layer

To ensure reproducibility of generated networks, SBML4j implements the W3C PROV data model (Fig. 3.4). The data model includes the `Agent`, `Activity` and `Entity` types from the specification and connects appropriate nodes with provenance relations of the types `hadMember (hM)`, `used`, `wasAssociatedWith (wAW)`, `wasAttributedTo (wAT)`, `wasGeneratedBy (wGB)`, and `wasDerivedFrom (wDF)`.

The `ProvenanceEntity` is the only class that directly inherits from `GraphBaseEntity`. It is the basic provenance component from which all further entities are derived. It provides a Java HashMap *provenance* which enables the addition of provenance-related annotation-information on any object in the database.

The three provenance node types are implemented as *NodeEntity* objects that inherit from `ProvenanceEntity`. Their names are prefixed with `ProvenanceGraph` and suffixed with `Node`, as all entities that derive from them must be node entities. Additionally, the basic relationship entity `ProvenanceGraphEdge` is also derived from `ProvenanceEntity`. It uses the same prefix, but instead the suffix *Edge* to denote that all entities that inherit from it must be relationship entities in the graph. It receives the custom label `PROV` and a `type` attribute, which enables the specific traversal of this part of the database graph based on the type of provenance relationship.

To capture metadata in the provenance graph, a `ProvenanceGraphMetadataNode` entity is added to the W3C model. Each entity that is derived from the `ProvenanceEntity` contains a list of these nodes. As `ProvenanceGraphMetadataNode` derives from `ProvenanceEntity`, it itself contains one such list. This enables the storage of nested metadata information where each nesting layer is represented by one node in the graph. The relationship that gets cre-

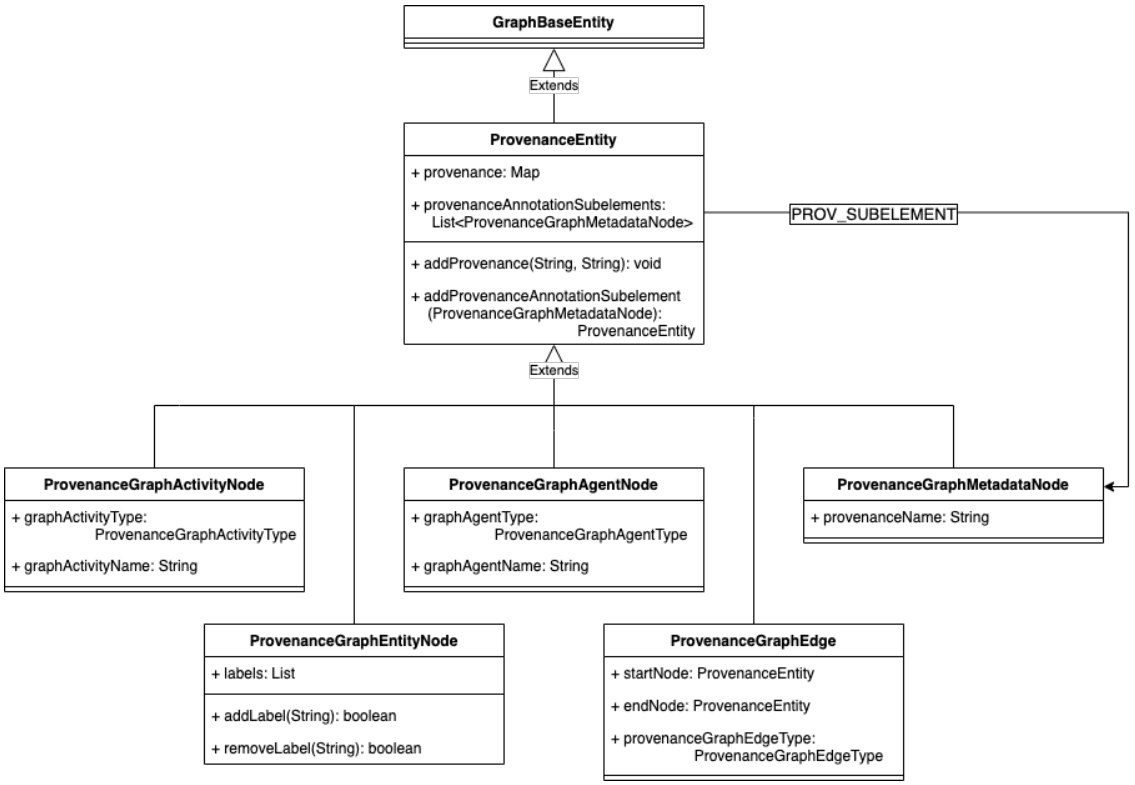


Figure 3.4: Provenance layer class hierarchy. The *ProvenanceEntity* inherits from the *GraphBaseEntity* and all other entities are derived from *ProvenanceEntity*. Standard methods like getters and setters are omitted for brevity.

ated between a parent and a child `ProvenanceGraphMetadataNode` as well as to the initial `ProvenanceEntity` are of type `PROV_SUBELEMENT`.

A typical provenance graph is shown in Figure 3.5. The `wDF` edges form a route from the latest network back to the database-representing node from which the source files originated. With the `wGB` and `used` edges the `Activity` nodes are connected to the warehouse nodes that contain the inventory information about the networks and pathways as well as other warehouse entities that were created in the process. The `Activity` nodes contain the provenance information about the task being carried out, which led from the network they used to the network that was generated by them. The `Agent` nodes get attributed to the warehouse entities which were created by the activities they are associated with.

To enable additional labels on nodes, the `ProvenanceGraphEntityNode` offers a list of string elements that represent the additional `labels` on the underlying entities. This list is recognized by the OGM via the `@Labels` annotation and creates node labels when persisting these entities. Neo4j does not allow custom labels on relationship entities. Therefore, this custom-label feature is added to this specific entity, as it is the first entity in the inheritance hierarchy that has the `@NodeEntity` annotation. All entities that extend this entity will also

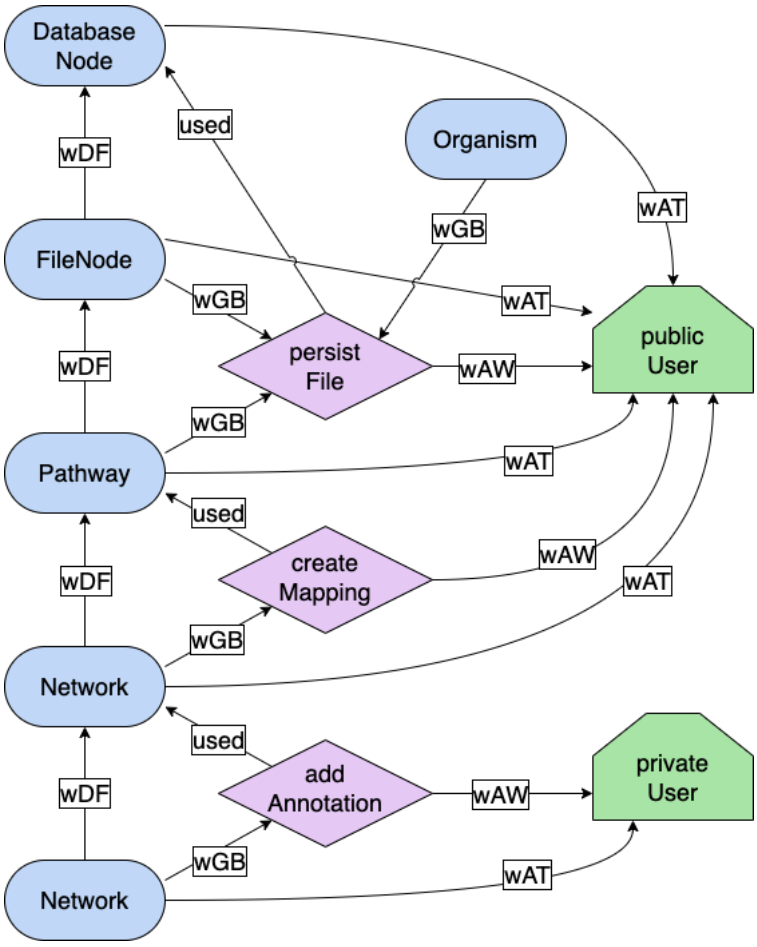


Figure 3.5: Provenance graph for one pathway and two derived networks. Colors and shapes of nodes correspond to the PROV data model (see Fig. 2.5). An uploaded SBML file results in the creation of a FileNode, a Pathway and an Activity "persistFile". They are connected to the Agent node, the database and the organism. A network mapping that is created from this pathway results in a Network node and an accompanying Activity "createMapping". Both are connected to the responsible Agent, in this case the "public user". Another Agent, a "private user", makes use of this network and adds an annotation to it. This is documented in the Activity "addAnnotation" and results in a new network node, which is attributed to this Agent.

be considered node entities, even if they do not use the `@NodeEntity` annotation themselves. If a custom label were to be assigned to a relationship entity, the OGM transforms it to a node entity instead and adds additional relationships that connect it to the intended source and target nodes. This would result in different query terms to extract this specific relationship compared to relationships that do not have custom labels added. To avoid potential loss of data on extraction, SBML4j restricts custom labels to the nodes in the graph.

Warehouse Layer

The *Warehouse layer* holds the metadata of the models that have been loaded. Each node in this layer is derived from the type `WarehouseGraphNode`, which itself is derived from the type `ProvenanceGraphEntityNode`. Figure 3.6 shows the entities in this layer and their connections with each other. The `WarehouseGraphNode` holds an additional `Map` for annotations concerning warehouse metadata, which are prefixed with *warehouse*. Each `WarehouseGraphNode` is linked to an instance of `Organism`, which has to be provided when a model is uploaded to SBML4j. An `Organism` is defined by the attribute `orgCode`, the three letter organism identifier defined by KEGG.

As each model has to be uploaded from a textual representation in a file, a `FileNode` is created for each model. The `FileNode` is defined by the attributes *filename*, *fileNodeType* and *md5sum*, providing the original filename from the uploaders file system, a filetype of the input file of the custom type *FileNodeType* and the *128-bit MD5 hash*[107] of the original file calculated on the server, respectively.

A `DatabaseNode` is created for every combination of source and version, which are provided when uploading SBML models. The unique key for the `DatabaseNode` entity consists of the attributes *source* and *sourceVersion*. If multiple models are uploaded for the same database source and version, they are connected to the same database-representing entity.

Each model that is uploaded to SBML4j results in the creation of a `PathwayNode` which represents the model in the database. Its attributes are *pathwayIdString* and *pathwayNameString* which are populated with the attributes `id` and `name` from the input model which is of type *org.sbml.jsbml.Model*. All entities that are created as part of this model in the SBML layer are connected to the PathwayNode with relationships of type `CONTAINS`. Multiple pathways can be grouped together in a collection, which results in the creation of a `PathwayCollectionNode` and an additional `PathwayNode`. Together these two nodes define the collection, while the first one is used to identify the pathway as being a collection, and the latter one holds the references to the entities in the collection. Each `PathwayNode` that is part of the group gets connected to the `PathwayCollectionNode` with a provenance relationship of type *hadMember*. Furthermore, the `PathwayCollectionNode` is attributed to the provided user and connected to the organism of the contained pathways. The `PathwayNode` that gets created is connected to the `Organism` which is associated with the `DatabaseNode` of the pathways in the collection.

For the creation of network mappings from a pathway (see Section 3.3.4), a `MappingNode` is created in the warehouse layer. The attributes of this node type are *mappingType*, *mappingName*, *mappingNodeTypes*, *mappingRelationTypes*, *mappingNodeSymbols* and *mappingRelationSymbols*. They describe the content of the mapping for inventory presentation via the REST interface. The `MappingNode` gets connected to its content in the mapping layer with relationships of type `CONTAINS`.

Figure 3.6: Classes of the warehouse layer. Classes that are used to store the metadata for the loaded models. The *ProvenanceGraphEntityNode* is derived from the *ProvenanceEntity*. The *WarehouseGraphNode* is derived from the *ProvenanceGraphEntityNode* and it provides the reference to an instance of the *Organism*. All warehouse-related nodes are derived from this type.

```
1       resource = "http://identifiers.org/kegg.genes/hsa:2064";
2       if (resource.contains("kegg.genes")) {
3           newExternalResourceEntity.setType(ExternalResourceType.KEGGGENES);
4           newExternalResourceEntity.setDatabaseFromUri("KEGG");
5       }
```

Listing 3.1: Java source code for database extraction from a given URI that points to `identifiers.org`.

The nodes in the Warehouse layer are used when creating responses to inventory-type requests (e.g., list all pathways) and when creating network mappings or user-derived networks. The relationships in this layer are used to identify the contents of individual pathways or networks.

SBML Layer

The *SBML* layer replicates the entities found in the SBML specification for the core and qualitative species package and adds additional entities for the expression of biomodels.net annotations on the entities (Fig. 3.7).

The entities in the SBML layer are derived from the two entities, which separates the content from the provenance and warehouse parts of the database graph, the `ContentGraphNode` and `ContentGraphEdge` entities. `SBMLSBase`, the `ExternalResourceEntity` and the `NameNode` entity are derived directly from `ContentGraphNode`. There is one relationship entity in the *SBML* layer that inherits from `ContentGraphEdge`, the `BiomodelsQualifier` entity.

The `SBMLSBase` entity is modeled after the `sBase` entity in the SBML core specification and holds the basic attributes *sBaseId*, *sBaseName*, and the *sBaseSboTerm*, which provides the identifier to the SBO ontology. In addition to these attributes, a list of `BiomodelsQualifier` entities represents the *biomodels.net* qualifier annotations that can be found on the corresponding entity in the SBML model. This `BiomodelsQualifier` entity is derived from the `ContentGraphEdge` entity and is the only explicit relationship entity in this layer. It is characterized by the *qualifier* and *type* attributes from the `CVTerm` class from the *org.sbml.jsbml* package. Each such entity connects one `SBMLSBaseEntity` with one `ExternalResourceEntity`. It defines the relationship type between this external resource and the biological entity it annotates. The `ExternalResourceEntity` is defined by the `URI` given in the attribute `uri`. It can be connected to different `BiomodelsQualifier` relations to multiple `SBMLSBaseEntity`-derived species. For URIs that point to the `identifiers.org` website, the corresponding source and type will be extracted and are stored in the `databaseFromUri` and `type` attributes, respectively (see Listing 3.1).

For external resources that point to the KEGG database, the REST endpoint given by the URI is queried and the information added to the knowledge graph. For genes, all names are

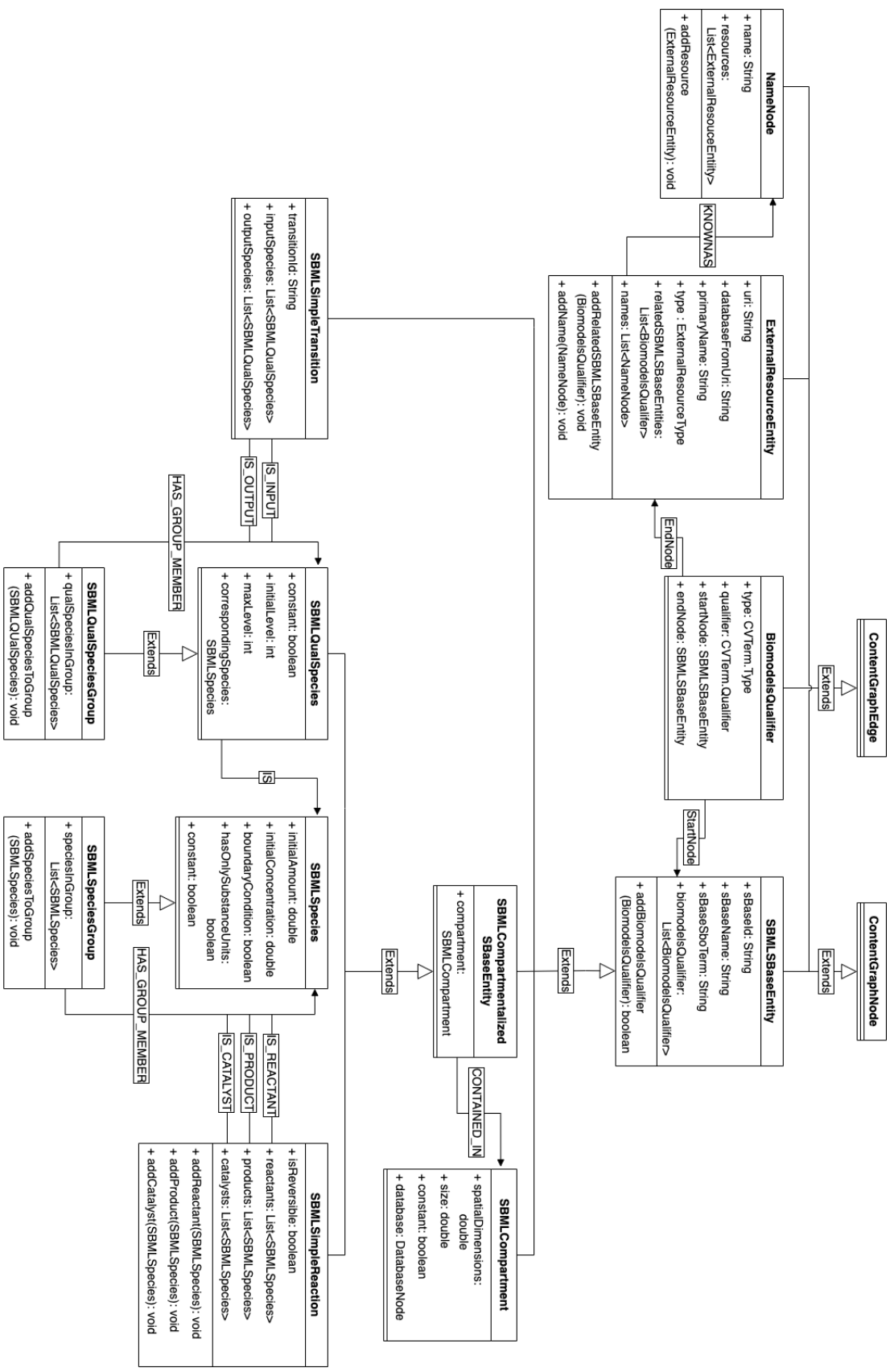


Figure 3.7: Classes of the SBML layer with base classes in the Content layer from which they inherit.

Homo sapiens (human): 2064

Help

Entry	2064 CDS T01001
Symbol	ERBB2, CD340, HER-2, HER-2/neu, HER2, MLN_19, MLN-19, NEU, NGL, TKR1, VSCN2, c-ERB-2, c-ERB2, p185(erbB2)
Name	(RefSeq) erb-b2 receptor tyrosine kinase 2
KO	K05083 receptor tyrosine-protein kinase erbB-2 [EC:2.7.10.1]
Organism	hsa Homo sapiens (human)

Figure 3.8: Top part of the KEGG genes web-response to the URI "http://identifiers.org/kegg.genes/hsa:2064". The entry number identifies this response element, the symbol field gives all known gene symbols for this entry, the name field provides a human-readable name derived from the NCBI Reference Sequence Database (RefSeq), the KO field links to the KEGG orthology, and the organism field provides the three-letter organism code that this gene specific gene entity is found in. Page last visited on Oct 20th, 2022

extracted from the list of known symbols (Fig. 3.8). The first symbol in this list is used as the *primaryName* attribute of the `ExternalResourceEntity`. For each additional symbol, a `NameNode` entity is created and added to the list of known names by relating it with a relationship of type `KNOWNAS`.

By linking the external resource directly to the `SBMLSBaseEntity`, it is ensured that all biological entities in the SBML layer can be enriched with *biomodels.net* qualified data.

The three entities `ExternalResourceEntity`, `BiomodelsQualfier` and `NameNode` have no direct counterpart in the SBML specification and are used for model-enrichment in the SBML4j database only. For that reason, those three entities are treated as their own data-model extension and can be found in the package `model.sbml.ext.sbml4j`.

Most biological entities in the SBML specification are bound to a compartment. These compartments are represented by the `SBMLCompartment` entity, which provides the same attributes as the SBML specification version, `spatialDimensions`, `size` and `constant`, which define the compartment in addition to the provided name that can be found in the `SBMLSBaseEntity` attributes. In addition, a compartment is bound to one database source, and has a relationship of type `OF` to the `DatabaseNode`, which represents this source.

The reference to any biological entity to such a compartment is stored in the `compartment` attribute of the `SBMLCompartmentalizedSBase` entity. The relationship that links the two is of type `CONTAINED_IN`.

The two biological entities from the SBML core specification that have representatives in the SBML4j data model are the `SBMLSpecies` and the `SBMLSimpleReaction` entities. They derive from `SBMLCompartmentalizedSBase`, as each such entity must be located in exactly one compartment. The `SBMLSpecies` entity is the main entity for representing biological entities, such as genes, proteins, and chemical compounds. They take part in biochemical

reactions, which are represented by the SBMLSimpleReaction entity. Multiple SBMLSpecies entities can form a group, and are then linked to a SBMLSpeciesGroup entity. As such groups can act as participants in biochemical reactions, they derive from SBMLSpecies. This way these groups can be treated equally to SBMLSpecies in any further computation.

The reaction entity which is defined in the SBML specification has a complex structure that serves the kinetic calculations, which are performed on these models. Since these types of calculations are not part of SBML4j, a simplified variant of the reaction element, the SBMLSimpleReaction has been implemented. It holds references to participants of the reaction and a Boolean flag for denoting if the reaction is reversible, but omits details about the kinetic properties of the reaction. With these elements from the core specification of SBML, the general biochemical processes that are described in the models can be stored in the database, while keeping unused elements for the kinetic calculations out of the data model for brevity.

For the representation of qualitative models, the two main elements from this extension are integrated into the SBML data model. The SBMLQualSpecies entities are the main elements from the qualitative models extension. They are connected to a corresponding SBMLSpecies to be able to link metabolic and non-metabolic parts of the knowledge graph. The remaining attributes on the SBMLQualSpecies entities are used for qualitative descriptions of the biological entity and are directly copied from the corresponding SBML-model elements. In the same way as SBMLSpecies, the SBMLQualSpecies entities can be grouped to SBMLQualSpeciesGroup entities, which are again modeled as SBMLQualSpecies themselves.

The SBMLSimpleTransition is derived from the SBMLSBaseEntity as it is not bound to a single compartment, but can describe processes that cross compartment boundaries as well. The SBMLSboTerm attribute from the SBMLSBaseEntity is used to encode the type of the transition. In addition, the SBMLSimpleTransition entity is given a string-based id, transitionId, which is built from the primary names of the participating entities and the name of the SBO-term, like "FGD1-stimulation->CDC42". The name of the SBOTerm is extracted from the SBO-class in *org.sbml.jsbml*, which defines short characterization of every SBO-term. The participating entities are referenced in the lists inputSpecies and outputSpecies for the input and output entities, respectively, and contain one or more SBMLQualSpecies entities.

Network Layer

The *Network* layer consists of the contents of the flattened network mappings and user-derived versions of these networks. Since we flatten the relationships and species details into simple network representations, the entities making up this layer are called FlatSpecies and FlatEdge (Fig. 3.9). The FlatSpecies entity is derived from the ContentGraphNode entity and is

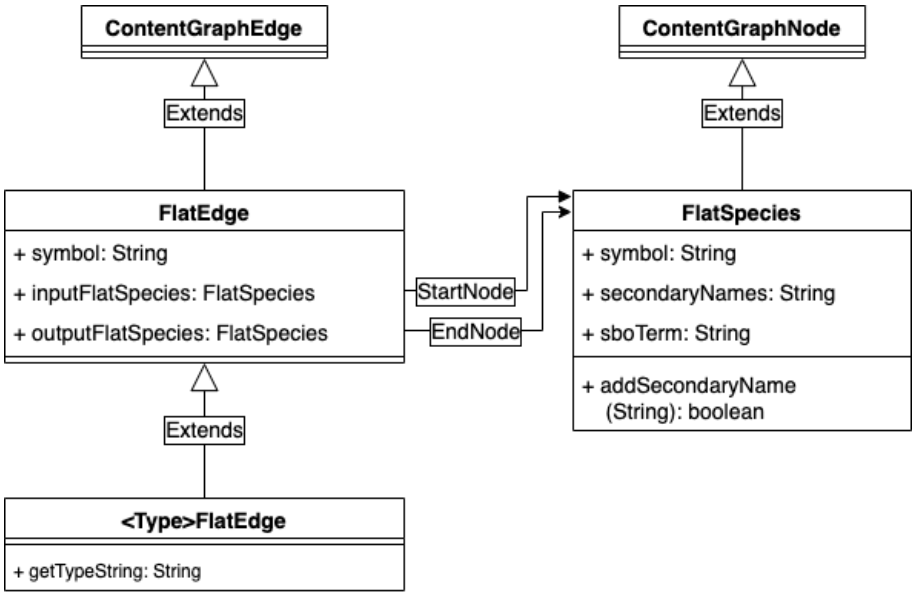


Figure 3.9: Classes of the Network layer with base classes in the Content layer from which they inherit.

used to represent `SBMLSpecies` or `SBMLQualSpecies` from the SBML layer in the network mappings. Multiple such species from the SBML layer can be collated to one `FlatSpecies`, if they are expressed by the same `ExternalResourceEntity`. The `primaryName` attribute of this external resource will be used as the `symbol` attribute of the `FlatSpecies` and all known secondary symbols from `NameNodes` are combined into a comma-separated string for the `secondaryNames` attribute. The `sboTerm` attribute contains the SBO term of the corresponding SBML-layer entity.

Biochemical reactions, which are modeled as `SBMLSimpleReaction` entities in the SBML layer, will also be mapped to a `FlatSpecies` entity. To mark them as reactions in the *network* layer, they get the additional label `FlatReaction` during the mapping process. Reaction participants in the form of other `FlatSpecies` are then connected to these reaction entities with the `FlatEdge` subtypes `REACTANTOF`, `PRODUCTOF`, and `CATALYZES`. By building reactions as nodes in the *network* layer, a metabolic network mapping forms a bipartite graph, where the reaction entities form one partition, and the participants form the other partition.

Non-metabolic mappings are composed of only one partition, which contains the relation partners as `FlatSpecies` entities. The `SBMLSimpleTransition` entities get mapped directly onto subtypes of the `FlatEdge` relationship entity. These relationships connect two `FlatSpecies` entities and assume the type of transition by the translated name of the SBO term. The only additional method `getTypeString()` on the `FlatEdge` entities returns the assigned label of the relationship entity. Since relationships in SBML4j can only have one label,

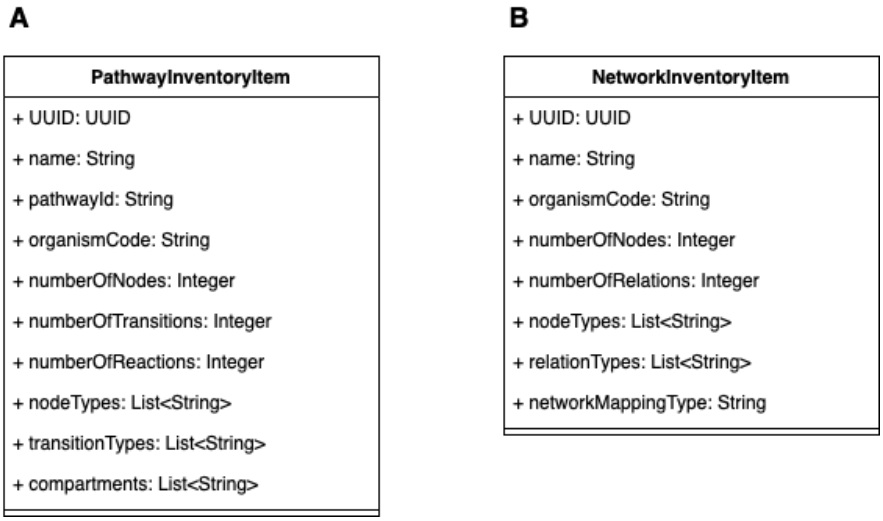


Figure 3.10: API response entity classes for inventory items. A. Inventory item for the `GET /pathways` endpoint, which lists the inventory of available pathways. B. Inventory item for the `GET /networks` endpoint, which lists the inventory of available networks.

they do not have the additional label of `FlatEdge`, but only the one, which denotes their subtype.

Table 3.1 gives the implemented relationship types of `FlatEdge`-derived entities and the mapping types in which these relationships are found. This list is based on the interactions that are present in the KEGG pathways.

API Request and Response Entities

API POST requests and responses use JSON-formatted content for transporting data between the client and SBML4j. To process and create those contents, custom classes are used, which derive from `ApiRequestItem` and `ApiResponseItem`.

The entities that extend `ApiResponseItem` provide inventory-type information to the client, with the two main inventory elements being for pathways (Fig. 3.10A) and networks (Fig. 3.10B). For reporting provenance three additional entities are defined in this category, the `PathwayCollectionItem` for listing the contents of a collection of pathways, and the `FileInventoryItem` and `DatabaseInventoryItem` for providing data on the source files and source database, respectively.

The other category of API entities extend the `ApiRequestItem` class and are used to provide structured data by the client. The `FilterOptions` class (Fig. 3.11A) lists all available types of nodes and edges, as well as all individual elements in a network. By providing subsets of these lists for a network, a filtered network is derived, which only contains the elements

Table 3.1: FlatEdge entity subtypes. Type of `Flat edge` in SBML4j with given label in the database, respective SBO Term and occurence in different network mapping types. Class name column gives the prefix to the full class name (i.e., "Catalyst" results in full class name "CatalystFlatEdge").

Class name <*>FlatEdge	**Edge Label**	**SBO Term SBO:<*>**	**Found in mapping**			
			met	ppi	reg	sig
Catalyst	CATALYSES	0000460	X	-	-	-
Control	CONTROL	0000168	-	-	X	-
Dephosphorylation	DEPHOSPHORYLATION	0000330	-	X	-	-
Dissociation	DISSOCIATION	0000180	-	X	-	-
Glycosylation	GLYCOSYLATION	0000217	-	X	-	-
Inhibition	INHIBITION	0000169	-	-	X	X
Methylation	METHYLATION	0000214	-	X	-	-
MolecularInteraction	MOLECULARINTERACTION	0000344	-	X	-	-
NonCovalentBinding	NONCOVALENTBINDING	0000177	-	X	-	-
Phosphorylation	PHOSPHORYLATION	0000216	-	X	-	-
Product	PRODUCTOF	0000011	X	-	-	-
ProteinComplexFormation	PROTEINCOMPLEXFORMATION	0000526	-	X	-	-
Reactant	REACTANTOF	0000015	X	-	-	-
Stimulation	STIMULATION	0000170	-	-	X	X
Targets	TARGETS	—	X	X	X	X
Ubiquitination	UBIQUITINATION	0000224	-	X	-	-
UncertainProcess	UNCERTAINPROCESS	0000396	-	X	-	-
UnknownFromSource	UNKNOWNFROMSOURCE	—	-	X	X	X

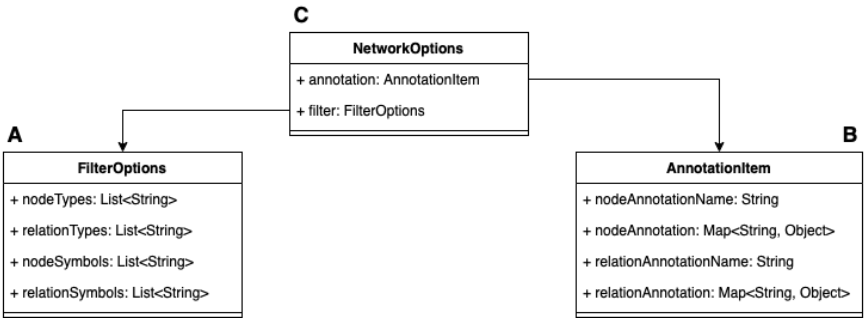


Figure 3.11: API response entity classes for filtering and annotation. A. Lists all available node and edge labels, as well as individual elements by name and can be provided with reduced content to filter accordingly. B. Provides maps for matching names of nodes and edges to potential annotation data, which a client can provide by filling in the map values. C. Bundles the two elements for annotation and filtering in one response element, which can be requested for each network.

present in the provided lists. If any list is omitted, no filter will be applied for it. To filter out all elements in a list, an empty list must be provided.

The `AnnotationItem` (Fig. 3.11B) provides two maps for defining annotations on nodes and edges, which are matched to `NULL` values initially. By providing subsets of these maps with values other than `NULL` for a network, the provided values will be added to the node or edge entities as annotations.

These options are bundled together in the `NetworkOptions` element (Fig. 3.11C), which can be requested for each network. The client can modify their contents and provide either of the two elements for annotation or filtering to the REST api to perform the requested actions on the network.

The `PathwayCollectionCreationItem` element is provided by the client to define the `Pathway` entities that are to be bundled to a pathway collection. Such a pathway collection can then be mapped to a single network mapping, which allows the creation of mappings of arbitrary subsets of pathways in the service. This element gives a name and description for the collections as well as a list of `entityUUIDs` of Pathways to collect.

The `NodeList` class consists of a List of Strings for providing symbols of nodes which should be used to calculate so-called context networks, which give the neighborhood of the provided nodes.

In addition, the `OverviewNetworkItem` extends the `NodeList` class with the elements `baseNetworkUUID`, `edgeweightproperty`, `annotationName`, and `networkName`. It is used in the convenience endpoint `/overview`, which generates a context network and adds a boolean annotation with the given name to the provided nodes in the resulting network.

For reporting the provenance information of a network, all relevant information is collected in the `ProvenanceInfoItem` (Fig. 3.12). It gives the type of entity and the content as the

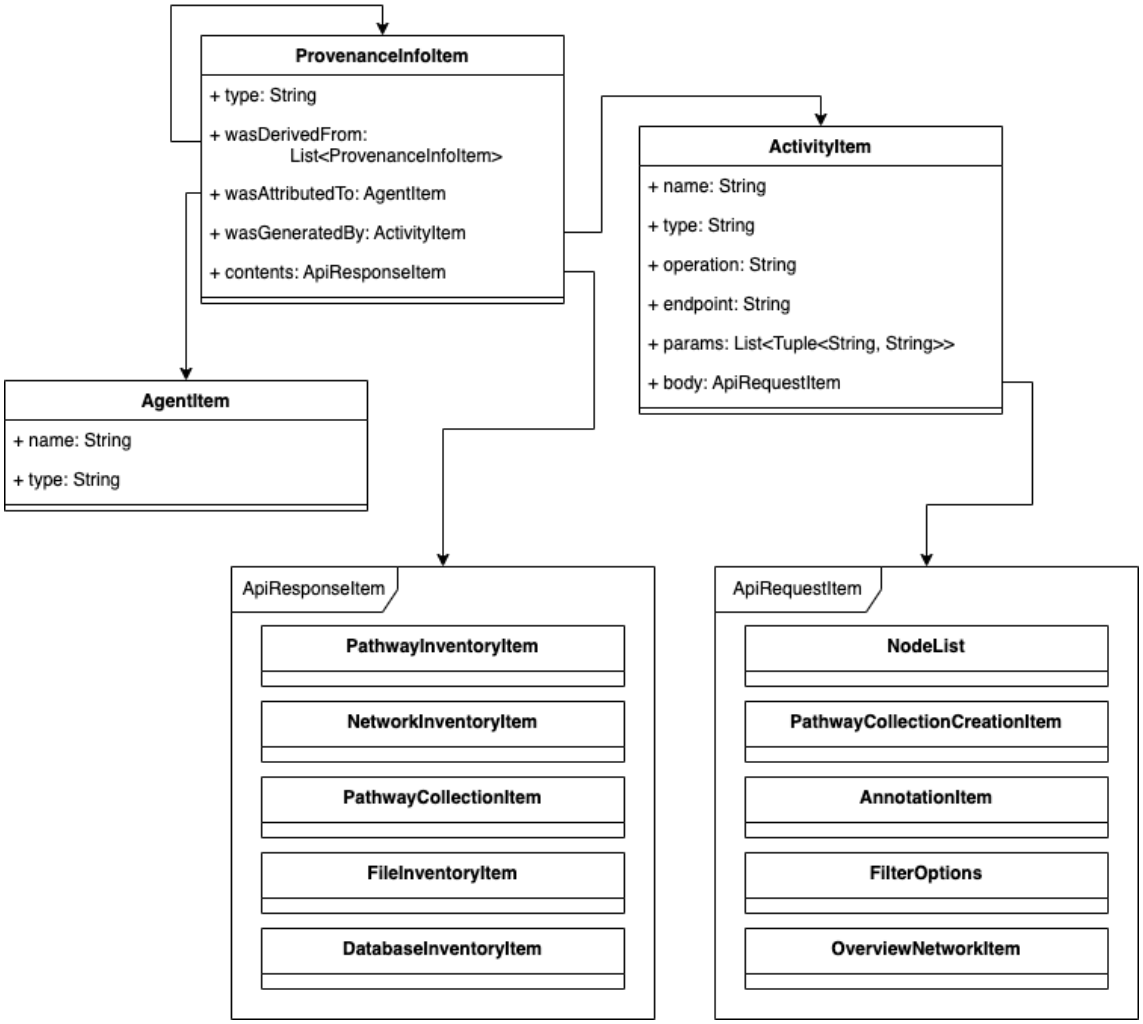


Figure 3.12: Provenance Report Item. Response Body for the `/prov` endoint.

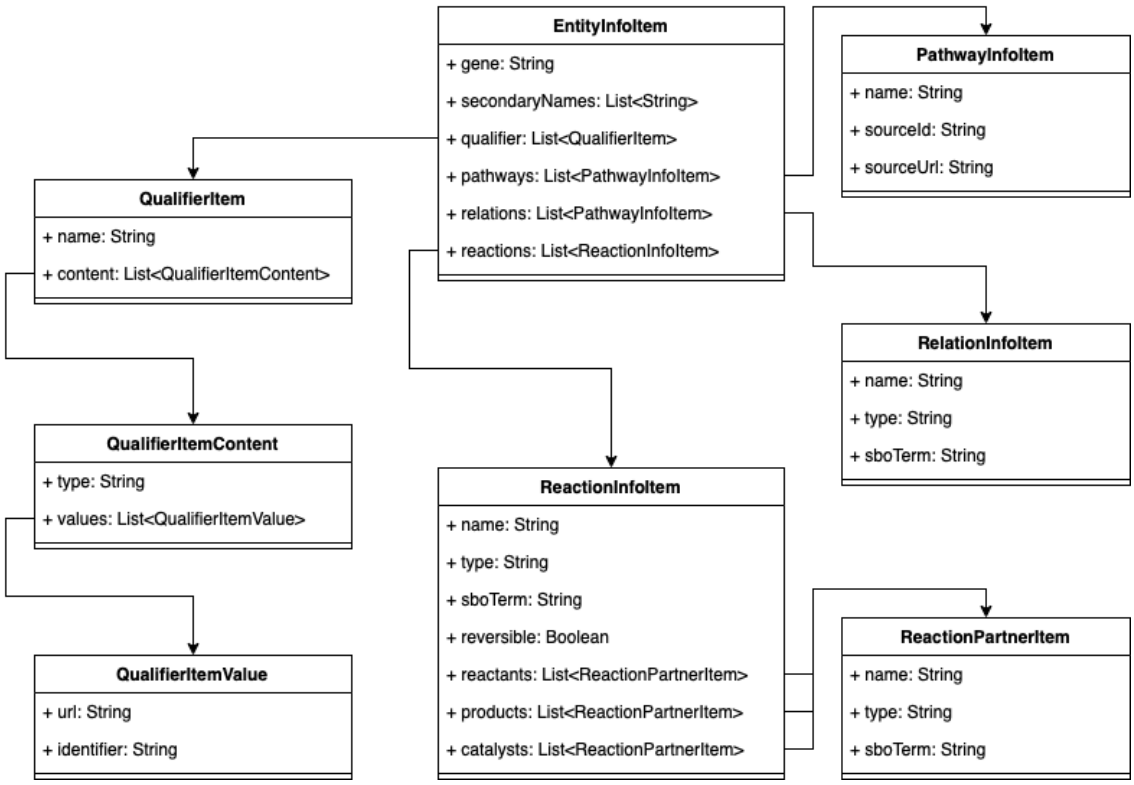


Figure 3.13: Entity Info Item. Response Body for the `/entityInfo` endpoints.

corresponding `ApiResponseItem`-derived inventory item. An `AgentItem` gives information on the user which requested the activity that generated this network. The `ActivityItem` contains the name and type of the activity, the type of REST request that was sent and the full name of the REST endpoint it was made to. Additionally, all data that was provided by the client to the endpoint is extracted. This includes the parameter names and values of the REST call in the list `params`, as well as potential body elements that were sent in `POST` and `PUT` requests. The body element consists of the `ApiRequestItem`-derived entity that was sent as JSON-payload in the request. The `wasDerivedFrom` attribute lists all entities from which this specific entity was derived from. Each such entity is itself of type `ProvenanceInfoItem` and contains the same information for the respective warehouse node at the end of the `wDF` relationship.

To inspect entities from the SBML knowledge graph, the `EntityInfoItem` (Fig. 3.13) is used. It contains the known secondary names of a given node, the reactions and transitions the entity takes part in, as well as the pathways it is found in and the Biomodels.net qualifiers which describe it.

Exceptions

To control error reporting for internal server errors, several custom exceptions are defined in SBML4j that extend the general `Exception` class. They do not implement special constructors, as they do not need to provide additional information to the general exception message. They control the correct handling and reporting of errors to the clients by the `controller` classes if an error occurred during the execution of the `service` class methods. When such an exception is thrown by a service method, the provided error message is relayed to the client in the `reason` header of the API response. It is therefore important that the provided error messages are human-readable and give enough information to the client for identifying the cause of the problem.

Enumerations

To uniformly define certain types of entities, enumerations are used. They are collected in the `GraphEnum` class. Noteworthy examples in this category include the available types of `Activity` entities and possible types of `ProvenanceGraphEdge` entities. Other enumerations define the type of network mapping, a standardized spelling for the network-manipulation steps, and the types of `ExternalResourceEntity` that are implemented, to name a few.

3.3.3 Repository Layer

The `repository` layer contains the database interfaces for retrieving and writing objects of the graph to and from the database. As with the data model, this layer is divided into several organizational units, which are represented by the package structure of the repository package. In the beginning of this section, the basic concept of the repositories will be laid out. The `Neo4jRepository` will be introduced, which serves as the basis for data extraction and persistence methods. In the rest of the section, specific repository methods will be highlighted, which make use of custom cypher queries on interface methods to handle complex data access logic. These methods are chosen to illustrate important operations. These include the querying of the SBML knowledge graph, the creation of network mappings from the SBML knowledge graph, the execution of the APOC graph algorithms, and the retrieval of the contents of networks as well as reading and writing provenance information.

Neo4jRepository

The `Neo4jRepository` is provided by the `Spring Data` module for the Neo4j database and offers the basic `save`, `find/findAll` and `delete` methods for individual elements and collections of elements. The optional `depth` parameter of these methods is used to reduce the number of related entities that are stored in or retrieved from the database. The parameter

```
1   @Query("MATCH "
2         + "(s:SBMLSpecies)"
3         + "-[b:BQ]->"
4         + "(e:ExternalResourceEntity) "
5         + "WHERE b.type = \"BIOLOGICAL_QUALIFIER\" "
6         + "AND b.qualifier IN [\"BQB_HAS_VERSION\", \"BQB_IS\", \"BQB_IS_ENCODED_BY\"] "
7         + "AND e.primaryName = $name "
8         + "AND e.databaseFromUri = $databaseFromUri "
9         + "RETURN s")
10  public Iterable<SBMLSpecies> findByBQConnectionTo(String name, String databaseFromUri);
```

Listing 3.2: Custom query for traversing the biomodels.net relations.

defines the number of relationships that can be traversed to find nodes, which are related to the provided entities.

The repository provides support for parameter-name extraction from method names. An interface method named `findByName` will result in the creation of a search query that uses the "name" element for result filtering. The keyword `find` triggers the creation of a `MATCH` query, while the so-called predicate, which is delimited by the `By` keyword results in the addition of the `WHERE` clause of the query. Multiple such predicates can be concatenated using the `And` keyword in the method name. More details and further options of query creation by method name can be found in the official documentation of the `Spring Data Neo4j` module[108].

In addition to query building from method names, custom Cypher queries can be created inside the repository interfaces using the `@Query` annotation from the `Spring Data Neo4j` package. Method arguments can be passed into the cypher query to modify the query according to previously collected data. This feature is used in most of the repositories of SBML4j to retrieve collections of elements of the graph with specific relationships and properties.

Querying the SBML Knowledge Graph

To retrieve specific entities from the SBML knowledge graph, the `ExternalResourceEntity` and the biomodels.net qualifier relation `BQ` can be traversed (see Listing 3.2).

The interface method `findByBQConnectionTo` (Listing 3.2, line 10) takes two arguments. The name or symbol of the requested entity is given by the `name` argument. The assigned database of the `ExternalResourceEntity` that was extracted from the original URI is provided by `databaseFromUri`. The query matches all biomodels.net qualifier relations "b", which have the label `BQ` that connect an `SBMLSpecies` "s" and an `ExternalResourceEntity` "e". The `WHERE` clause limits the result set to these "e", which match the provided name and database, and the type and qualifier of "b" as defined by the query. The hard-coded limitation to the three qualifier `HAS_VERSION`, `IS`, and `IS_ENCODED_BY` (line 6) is used throughout SBML4j. On the one hand, this bridges the gap between the DNA and protein level by allowing encoding genes as source for proteins (i.e., `IS_ENCODED_BY`), on the other hand it enables

```
1   @Query("MATCH "
2       + " (p:PathwayNode)"
3       + "<-[w:Warehouse]->"
4       + "(r:SBMLSimpleReaction) "
5       + "WHERE p.entityUUID = $pathwayUUID "
6       + "AND w.warehouseGraphEdgeType=\"CONTAINS\" "
7       + "WITH r "
8       + "MATCH "
9       + "(r)"
10      + "-[rel:IS_PRODUCT|IS_REACTANT|IS_CATALYST]->"
11      + "(s:SBMLSpecies) "
12      + "WHERE s.sBaseSboTerm IN $nodeSBOTerms "
13      + "WITH r, rel, s "
14      + "MATCH "
15      + "(s)"
16      + "<-[b:BQ]->"
17      + "(e:ExternalResourceEntity) "
18      + "WHERE b.type = \"BIOLOGICAL_QUALIFIER\" "
19      + "AND b.qualifier IN [\"BQB_HAS_VERSION\", \"BQB_IS\", \"BQB_IS_ENCODED_BY\"] "
20      + "RETURN e, "
21      + "b, "
22      + "s as species, "
23      + "type(rel) as typeOfRelation, "
24      + "r as reaction")
25  Iterable<MetabolicPathwayReturnType> getAllMetabolicPathwayReturnTypes(
26                                          String pathwayUUID,
27                                          List<String> nodeSBOTerms);
```

Listing 3.3: Custom Cypher query for extracting metabolic interactions in a pathway.

hits for species that are described through multiple variants (i.e., `HAS_VERSION`) or exact matches (i.e., `IS`). The query returns a collection of matching `SBMLSpecies` entities, which are returned by the repository method via an `Iterable` collection in Java.

Mapping Creation

When network mappings are created from the SBML knowledge graph, either the metabolic parts or a subset of the non-metabolic parts of a pathway have to be extracted. To extract the metabolic parts, all reactions in a pathway are first matched by traversing the warehouse relations of type `CONTAINS` (Listing 3.3, lines 1–6) that connect all entities to the warehouse node of type `PathwayNode`. With these reactions, all reaction partners that are connected via the relations `IS_PRODUCT`, `IS_REACTANT`, and `IS_CATALYST` are matched (Listing 3.3, lines 7–12). For all such partners, the external resources are queried that describe them (Listing 3.3, lines 13–19). Again, the same type and qualifier values are used as when querying the SBML knowledge graph for `SBMLSpecies` based on their symbol. This query returns multiple entities (i.e., e, b, s, r) and the type of the relationship "rel" named "typeOfRelation" (Listing 3.3, lines 20–24). The entities "s" and "r" as well as the string "typeOfRelation" are found as class variables in the `MetabolicPathwayReturnType`, while the describing entities "e" and "b" are populated in the `SBMLSpecies` entities, because they are part of the result set of the query.

```
1    @Query(value = "MATCH "
2          + "(pw:PathwayNode)"
3          + "<-[w:Warehouse]->"
4          + "(t:SBMLSimpleTransition) "
5          + "WHERE pw.entityUUID = $pathwayEntityUUID "
6          + "AND w.warehouseGraphEdgeType = \"CONTAINS\" "
7          + "AND t.sBaseSboTerm in $transitionSBOTerms "
8          + "WITH t "
9          + "MATCH p="
10         + "(t)"
11         + "<-[tr:IS_OUTPUT|IS_INPUT]->"
12         + "(q:SBMLQualSpecies)"
13         + "<-[bq:BQ]->"
14         + "(e:ExternalResourceEntity) "
15         + "WHERE bq.type = \"BIOLOGICAL_QUALIFIER\" "
16         + "AND bq.qualifier IN [\"BQB_HAS_VERSION\", \"BQB_IS\", \"BQB_IS_ENCODED_BY\"] "
17         + "AND q.sBaseSboTerm in $nodeSBOTerms "
18         + "RETURN p")
19     Iterable<SBMLSimpleTransition> findMatchingPathsInPathway(
20                                         String pathwayEntityUUID,
21                                         List<String> transitionSBOTerms,
22                                         List<String> nodeSBOTerms);
```

Listing 3.4: Custom query for extraction of non-metabolic transitions of a pathway.

To retrieve non-metabolic interactions in a pathway, all `SBMLSimpleTransition` entities have to be matched that are contained in the pathway (Listing 3.4, lines 1–6). Additionally, they must have a matching SBO term found in the `transitionSBOTerms` list (Listing 3.4, line 7). With these transitions, a path "p" is declared, which consists of the match of the transition entity "t" from the first part of the query, the relationship of either `IS_INPUT` or `IS_OUTPUT`, the corresponding `SBMLQualSpecies` entity, and the external resource "e" that is connected through the biomodel.net qualifier "bq" (Listing 3.4, lines 8–17). Here, the SBO-term of the `SBMLQualSpecies` entity has to match one of the provided terms in the list `nodeSBOTerms`. In the end the path "p" is returned by the query (Listing 3.4, line 18). As all entities in the path are connected either directly to the `SBMLSimpleTransition` or indirectly through the `SBMLQualSpecies`, all elements of the path get populated in the `SBMLSimpleTransition` entity. The repository method (Listing 3.4, lines 19–22) can therefore return an iterable of `SBMLSimpleTransition`, in which each element holds references to the other parts of the corresponding path on the object level in the Java code.

Network Traversal and Extraction

The path expander function from the APOC procedures is the main method for finding network neighborhoods in the Neo4j database (see Listing 3.5). It performs a breadth-first search (bfs) and returns all traversed relationships of successful path expansion that satisfy the given criteria. The function call (Listing 3.5, line 5) is performed inside a regular Cypher query. As it needs a reference to a node from which the expansion should start, the function call needs to be preceded by a `MATCH` clause that uses a `WITH` statement in place of the regular `RETURN`

```
1   @Query("MATCH "
2       + "(start:FlatSpecies) "
3       + "WHERE start.entityUUID = $startNodeEntityUUID "
4       + "WITH start "
5       + "CALL apoc.path.expand("
6       + "start, "
7       + "$apocRelationshipString, "
8       + "$apocNodeString, "
9       + "$minDepth, "
10      + "$maxDepth "
11      + ") "
12      + "YIELD "
13      + "path as apocPath "
14      + "RETURN relationships(apocPath) as pathEdges;"
15      )
16  List<FlatEdge> runApocPathExpandFor(String startNodeEntityUUID,
17                                      String apocRelationshipString,
18                                      String apocNodeString,
19                                      int minDepth,
20                                      int maxDepth);
```

Listing 3.5: Cypher query and interface method for the path.expand APOC function.

statement (Listing 3.5, lines 1–4). The start node is identified by the `entityUUID`, which is provided as the first method argument of the interface. The path.expand function takes four additional arguments, which configure how the expansion is performed on the graph. The `apocRelationshipString` defines which relationships can be traversed during expansion, including their direction. Multiple relationships can be provided by separating them with the "|"-character. The direction in which the algorithm may traverse a given relationship can be set using the "<" and ">" before or after the name of the relationship. To allow both directions, these symbols have to be omitted. The `apocNodeString` gives a list of node labels for determining the nodes which the algorithm may include in the search. Again, the "|"-character can be used to provide multiple node labels. Using the "+" or "-"-character, nodes can be explicitly included or excluded, respectively. Of further importance is the "/"-character in front of a node label, which determines that the expansion should stop at the specified label, and only paths that end with a node of this label are included in the result set. The last two parameters, `minDepth` and `maxDepth` determine the minimum and maximum number of relationships that need to be traversed for a path to be eligible for returning.

After the APOC-method call, the query defines which parts of the return value of the algorithm should be yielded for building the result set of the query, in this case the whole `path` (Listing 3.5, lines 12–13). In the final part of the query the return values are defined (Listing 3.5, line 14). Only the relationships need to be returned, since they contain explicit references to the nodes they connect and therefore will be part of the returned relationships. All relationships are returned as `FlatEdge` entities in a `List` by the interface method.

The second query that is used from the APOC package is the shortest path search, which is an implementation of Dijkstra's shortest path algorithm[109] (Listing 3.6, line 7). This function

```
1   @Query(value="MATCH (fs1:FlatSpecies) "
2         + "WHERE fs1.entityUUID = $startNodeEntityUUID "
3         + "WITH fs1 "
4         + "MATCH (fs2:FlatSpecies) "
5         + "WHERE fs2.entityUUID = $endNodeEntityUUID "
6         + "WITH fs1, fs2 "
7         + "CALL apoc.algo.dijkstraWithDefaultWeight("
8         + "fs1, "
9         + "fs2, "
10        + "$relationTypesApocString, "
11        + "$propertyName, "
12        + "$defaultWeight) "
13        + "YIELD "
14        + "path as apocPath, "
15        + "weight as w "
16        + "RETURN relationships(apocPath) as pathEdges;"
17        )
18  List<FlatEdge> apocDijkstraWithDefaultWeight(
19                              String startNodeEntityUUID,
20                              String endNodeEntityUUID,
21                              String relationTypesApocString,
22                              String propertyName,
23                              float defaultWeight);
```

Listing 3.6: Cypher query and interface method for the dijkstraWithDefaultWeight APOC function.

takes two nodes as start and endpoints, which are matched by their `entityUUID` with the first two `MATCH` parts of the query (Listing 3.6, lines 1–6). It also requires a string defining the relationships that are to be traversed during the shortest path search, which has the same syntax as in the `path.expand` function. In addition, `propertyName` gives the name of an annotation key that contains a numeric value that acts as weight for the path-length calculation in Dijkstra's algorithm. If any or all relationships are missing this annotation, the provided value in the `defaultWeight` attribute will be used. Again, the relationships of the path are returned as `FlatEdge` entities in a list.

For extracting all contents of a network, usually to return it to the requesting client, the `FlatEdge` entities of a network are matched and populated with the `FlatSpecies` they connect (see Listing 3.7). This query matches these relationships, without specifying their individual label, by querying for direct relationships of two `FlatSpecies` entities that are contained in the same network. First, the `MappingNode` "m" is matched according to its `entityUUID` (Listing 3.7, lines 1–2). This enables its use in multiple positions of the `MATCH` query that follows. Then, the `FlatEdge`-derived relationships are matched (Listing 3.7, line 6), by providing it without a label (i.e., -[r]->). The query returns these found relationships and populates it with the matched `FlatSpecies` entities "s" and "s2" from the query (Listing 3.7, line 11). The repository method ((Listing 3.7, line 12) can then return a list of `FlatEdge` entities, which keep their individual derived type during return value generation.

```
1   @Query("match (m:MappingNode) "
2         + "WHERE m.entityUUID = $entityUUID "
3         + "WITH m MATCH "
4         + "(m)"
5         + "<-[w1:Warehouse]->"
6         + "(s:FlatSpecies)-[r]->(s2:FlatSpecies)"
7         + "<-[w2:Warehouse]-"
8         + "(m) "
9         + "WHERE w1.warehouseGraphEdgeType = \"CONTAINS\" "
10        + "AND w2.warehouseGraphEdgeType = \"CONTAINS\" "
11        + "RETURN s, r, s2")
12    List<FlatEdge> getNetworkContentsFromUUID(String entityUUID);
```

Listing 3.7: Custom query for network contents extraction.

```
1   @Query(value = "MATCH "
2         + "(start:ProvenanceEntity)"
3         + "-[edge:PROV]->"
4         + "(end:ProvenanceEntity) "
5         + "WHERE edge.provenanceGraphEdgeType = $edgetype "
6         + "AND start.entityUUID = $startNodeEntityUUID "
7         + "RETURN end")
8     Iterable<ProvenanceEntity> findAllByProvenanceGraphEdgeTypeAndStartNode(
9                 ProvenanceGraphEdgeType edgetype,
10                String startNodeEntityUUID);
```

Listing 3.8: Cypher query and interface method for finding the end node of a provenance relationship with a given start node.

Provenance-graph Traversal

The provenance graph is traversed by matching all provenance relationships of a given type and providing either the start or the end node in a query. The nodes on the other side of the relationships are returned. This class of queries makes use of two facts. First, every node entity in the data model is derived from `ProvenanceEntity`, which makes every node selectable by this node label. Second, all relationships are directed. The direction is indicated by the “>” symbol in line three of Listing 3.8. This way results can be limited to nodes on specific ends of relationships.

```
1   @Query(value = "MATCH "
2         + "(start:ProvenanceEntity)"
3         + "-[edge:PROV]->"
4         + "(end:ProvenanceEntity) "
5         + "WHERE edge.provenanceGraphEdgeType = $edgetype "
6         + "AND end.entityUUID = $endNodeEntityUUID "
7         + "RETURN start")
8     Iterable<ProvenanceEntity> findAllByProvenanceGraphEdgeTypeAndEndNode(
9                 ProvenanceGraphEdgeType edgetype,
10                String endNodeEntityUUID);
```

Listing 3.9: Cypher query and interface method for finding the start node of a provenance relationship with a given end node.

```
1  import java.util.UUID;
2
3  @Service
4  public class GraphBaseEntityService {
5    private String getNewUUID() {
6      return UUID.randomUUID().toString();
7    }
8  }
```

Listing 3.10: EntityUUID generation in the GraphBaseEntityService class.

The query in Listing 3.8 traverses the provenance relationship in the direction of the relationship by providing the `entityUUID` of the start node and returning all found end nodes. The opposite query in Listing 3.9 traverses the relationship against the direction by providing the `entityUUID` of the end node and returning all found start nodes.

3.3.4 Service Layer

SBML4j consists, at its core, of a multitude of small services that interact with each other through dependency injection to fulfill requests that are sent to the REST API by clients.

The first category of services handle the connections to the database through the repositories and create or manipulate graph entities. One service of interest in this category is the `GraphBaseEntityService`, which will be introduced in this section.

The other category of services is responsible for receiving data from the `controller` layer and processing client input from the API. The main tasks that will be covered in this section are building the SBML knowledge graph from SBML models, creating network mappings from this knowledge graph, the copying step of network contents for various network-related activities as well as the tracking and extraction of provenance information for a network.

GraphBaseEntity Service

The `GraphBaseEntity` service handles the core attributes `id` and `version` of every entity that is created in SBML4j. It is also responsible for generating `entityUUIDs` for new entities. The private class method `getNewUUID` generates a pseudo-randomly generated UUID using the static factory from `java.util.UUID` (Listing 3.10). The string representation is used on the entity, as the OGM cannot natively map UUIDs to attributes on nodes or relationships.

Since the `GraphBaseEntity` class holds the map for client-provided annotation data, this service is responsible for adding that data to the entity. The method `addAnnotation` (Listing 3.11) receives the value of the annotation as generic `java.lang.Object` type and needs to be provided with a string representation of the type of data that is added. This representation is later used to cast the object to the appropriate type. A service that uses this method must therefore determine the type of the object before the call. The last argument `appendExisting`

```
1    public GraphBaseEntity addAnnotation(
2            GraphBaseEntity entity,
3            String annotationName,
4            String annotationType,
5            Object annotationValue,
6            boolean appendExisting
7            );
```

Listing 3.11: GraphBaseEntityService class-method signature for adding annotations to entities.

defines whether an existing annotation with that name should be replaced or the new data appended at the end of it. The default separator for appending string-type annotations is the comma character and is always followed by a whitespace. Custom separators for a variety of standard annotation names can be configured in the main application configuration. A full list of available names can be found in Table D.1. The method returns the modified entity, but does not persist the changes in the database. The calling service has to take care of persisting the entity after adding the annotation.

Building the SBML Knowledge Graph

The `SBMLSimpleModelService` class is responsible for extracting the entities from the SBML model. In a first step the compartments are extracted from the model and created in the database. Existing `SBMLCompartment` nodes are reused if they match in all attributes. For each model, the containing `SBMLSpecies`, `SBMLSimpleReaction`, `SBMLQualSpecies`, and `SBMLSimpleTransition` are always created as new entities. While this may lead to redundant node entities if two models use the same biological entities with the same properties, it ensures that each model can be viewed isolated from the rest of the knowledge graph.

`ExternalResourceEntities` and accompanying `NameNodes` will be reused if they are encountered in multiple models. If an `ExternalResourceEntity` already exists on model persistence, the entity is retrieved from the database and a new `BiomodelsQualifier` relationship is built, connecting it to the new model entity that it annotates.

Network Mapping

The `NetworkMappingService` service interface is responsible for creating network mappings and provides one public method named `createMappingFromPathway` (Listing 3.12). Based on the contents of a provided `pathway` and a type of network mapping, this method builds the mapping by flattening the hierarchies and relationships, connects the contents to the warehouse and the provenance layer and returns the warehouse entity, which represents the created mapping. The four available mapping types can be divided in `metabolic` and `non-metabolic` mappings. While the metabolic mapping contains only the biochemical reactions as a bipartite

```
1   public interface NetworkMappingService {
2
3     public MappingNode createMappingFromPathway(
4                          PathwayNode pathway,
5                          NetworkMappingType type,
6                          ProvenanceGraphActivityNode activityNode,
7                          ProvenanceGraphAgentNode agentNode,
8                          String newMappingName)
9                        throws NetworkAlreadyExistsException, Exception;
10  }
```

Listing 3.12: Service interface for creating network mappings from pathway data.

directed graph, the non-metabolic mappings are divided further by the node types and relationship types that are extracted from the SBML layer. Node types are restricted to polypeptide chains and non-covalent complexes for PPI and signaling mappings, while regulatory mappings can also contain simple chemicals. The relationship types for the mapping are filtered by SBO-terms and result in the available relationships per mapping as illustrated in Table 3.1. The question whether a certain mapping type is directed or undirected is not considered at creation time as this only influences traversal of the relationships, which can be controlled at access time.

One important aspect of the mapping-building process is the flattening of hierarchies in groups of biological entities. In the SBML knowledge graph, a `SBMLQualSpecies` group will be a direct partner in a transition, but not the individual `SBMLQualSpecies` that are part of this group. In the network mapping, this group is to be resolved in the following way. First, all group members are connected to the transition partner of the group by the appropriate `FlatEdge` that represents the SBO term of the `SBMLSimpleTransition`. Additionally, all group members are connected to each other to add the group information to the network mapping. Here, two possible relationships are built. If all group members are of type polypeptide, the connecting relationship is of type `protein complex formation` and it gets assigned the SBO-term *SBO:0000526*. If at least one member is of a different type (e.g., simple chemical), the relationship between the partners is of type `molecular interaction` and instead gets assigned the SBO-term *SBO:0000344*.

The four different network mappings that can be built in SBML4j are illustrated with a theoretical example in Figure 3.14. It shows which elements of the SBML knowledge graph are considered for each type of network mapping.

Copying Network Contents

Unless otherwise configured or requested specifically by a client, networks get duplicated when any action is performed on them. This way, all intermediary steps of a network analysis pipeline will be available at all times and provenance of all these steps can be tracked independently

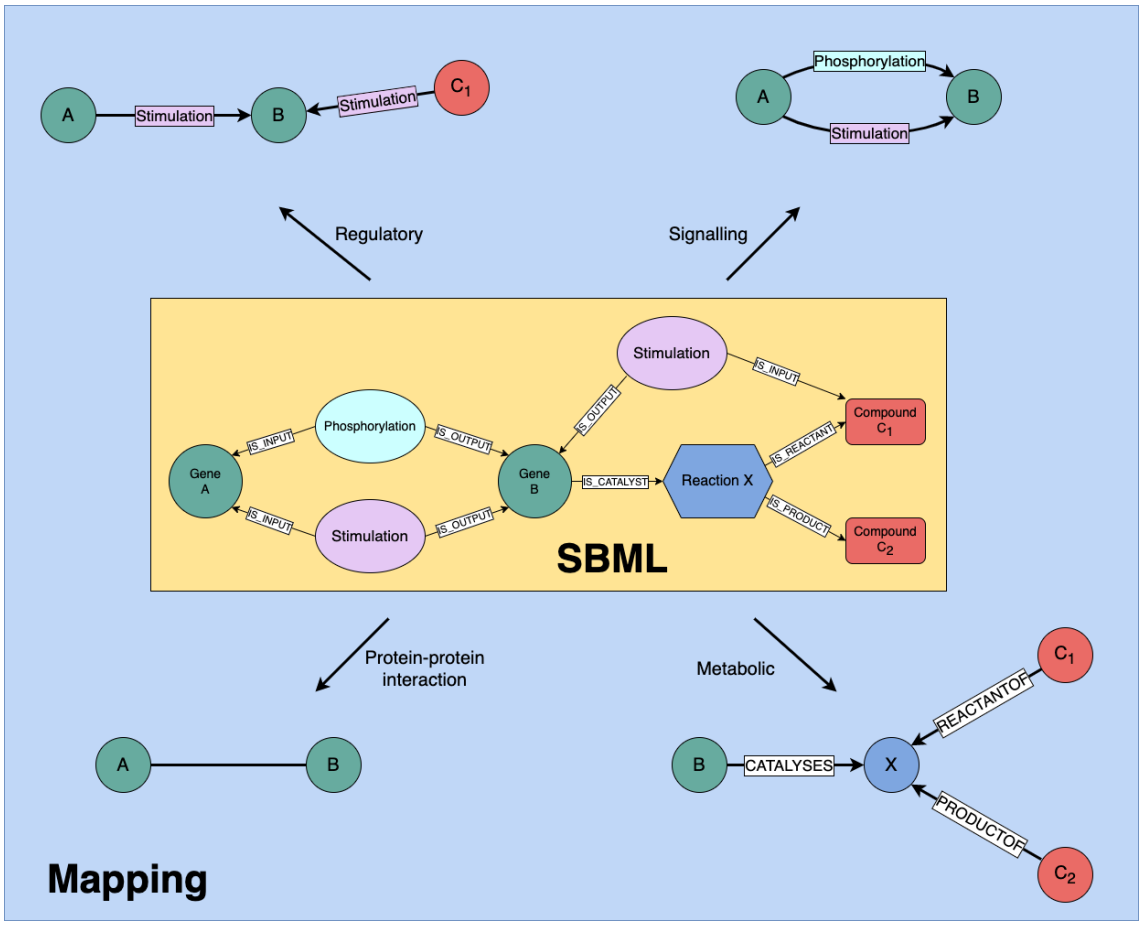


Figure 3.14: Differences in network-mapping creation from the same SBML knowledge graph. Regulatory and signaling mappings are directed graphs containing different subsets of transition types and the same set of node types. In the SBML area (yellow) a synthetic SBML knowledge graph is depicted with `Gene A` and `Gene B` connected to two shared `Transitions` of different types (phosphorylation, stimulation). Additionally, `Gene B` acts as a catalyst of `reaction X` with reaction partners `Compound C1` and `Compound C2`. In the Mapping area (blue) the four different mapping types are shown. A regulatory network mapping consists of transitions of regulatory nature (Stimulation, Inhibition) that connect the FlatSpecies node entities derived from the SBML entities *Gene A* and *Gene B* as *A* and *B* respectively. A signaling network mapping contains additional transition types like phosphorylation events but does not include node entities other than polypeptide chains, groups of polypeptide chains, and any transitions connecting those omitted entities. A protein-protein interaction mapping is an undirected graph with only `polypeptide-chain` nodes. The undirected edges signify that there is at least one interaction between the two, but the type is omitted. The nodes labeled "A" and "B" correspond here to the proteins that are encoded by *Gene A* and *Gene B*, respectively. A metabolic network mapping ignores transitions from the SBML knowledge graph and focuses on the reactions present. The result is a bipartite graph with reaction partners (reactants, products, catalysts) in one node partition and the reactions in the other.

```
1  public void resetGraphBaseEntityProperties(GraphBaseEntity target) {
2    target.setEntityUUID(this.getNewUUID());
3    target.setId(null);
4    target.setVersion(null);
5  }
```

Listing 3.13: GraphBaseEntityService-class method for resetting basic attributes on an entity.

```
1  import org.neo4j.ogm.session.Session;
2
3  @Service
4  public class NetworkService {
5    Session session;
6    ...
7    public MappingNode copyNetwork(...) {
8      ...
9      // reset all objects
10     this.session.clear();
11     //persist reset objects as new database entities
12   }
13 }
```

Listing 3.14: NetworkService-class method excerpt illustrating the "clear" operation of the OGM session during network duplication.

of one another. To copy the contents of a network, the following procedure is used: First, the network is extracted and all nodes and relationships are instantiated by the OGM. Instead of issuing copy procedures in code, the duplication process is relayed to the database, where the new network contents need to be created in any way after applying the requested operations of the current activity. To relay the duplication process to the database, the basic attributes of all entities are reset (Listing 3.13). A new UUID is generated for the entity and the `id` and `version` attributes are set to `null`. All other attributes of the `target` entity are left as they are, to create a duplicate entity that differs only in the basic attributes that are responsible for entity matching by the OGM. After resetting all objects in the network in this way, the OGM session is cleared of all known mappings between database entities and class objects (Listing 3.14, line 11). When the reset objects are then persisted in the database, they will be assigned new `id` and `version` values by the OGM and created as new entities in the database.

Provenance

The `Provenance` services are central services in SBML4j. The `ProvenanceGraph` service is used to build the provenance relationships, find nodes based on their connectedness with provenance relationships and is also responsible for creating `Agent` and `Activity` nodes as well as adding provenance annotation to these nodes. It is used by most controllers and services in SBML4j that create or manipulate data. The most frequently used provenance task is to connect two entities with a `ProvenanceGraphEdge`. Since all entities derive from

```
1   void connect( ProvenanceEntity source,
2                 ProvenanceEntity target,
3                 ProvenanceGraphEdgeType edgetype);
4
5   void connect( ProvenanceEntity source,
6                 String targetEntityUUID,
7                 ProvenanceGraphEdgeType edgetype);
8
9   void connect( String sourceEntityUUID,
10                ProvenanceEntity target,
11                ProvenanceGraphEdgeType edgetype);
12
13  void connect( String sourceEntityUUID,
14                String targetEntityUUID,
15                ProvenanceGraphEdgeType edgetype);
```

Listing 3.15: ProvenanceGraphService method signatures for connecting two entities with a ProvenanceGraphEdge.

`ProvenanceEntity`, the `connect` methods work with these types of objects and can thus connect any two entities in the database (Listing 3.15).

In addition to methods that take two such entities as arguments, oftentimes during processing one entity is instantiated, while the other entity is referenced by the `entityUUID`, or just two `entityUUID`s are known. For this reason, the method is overloaded with different combinations of arguments.

The `ProvenanceReport` service gathers the provenance information that is stored for a specific network and all its predecessors including the initial SBML file information. The main traverse follows the `wDF` relationships from the requested network up to the database node, which marks the origin of the data (see Fig. 3.5). For each warehouse node on this path, the inventory information about the contents are extracted. Then all activities that were involved in generating this node and its contents are visited by following the `wGB` relationship. The activity nodes contain the provenance information about the modification step that was performed, including the request parameters that were sent to the API endpoint as well as the request body of `POST` and `PUT` requests.

In case that no `used` relationships are found for an activity (e.g., see the activity "persistFile" in Fig. 3.5), incoming `wGB` relationships are evaluated instead. `PathwayCollectionNodes` on the `wDF` path follow the `hM` relationships to the individual `PathwayNode` entities, whose inventory items are included in the report. Finally, when the `DatabaseNode` is reached, which has no outgoing `wDF` relationship, the connected organism is collected via the warehouse graph connection. All collected data is returned to the controller, which builds the JSON-formatted response for the client.

Input and Output (I/O)

This domain is responsible for receiving and sending files from and to the clients, processing the contents of the files and requesting data from external resources via calls to RESTful interfaces. Input from files is processed in the `SBML`, `GraphML` and `CSV` services when clients provide files of one of these types. For SBML files, the contained SBML model is extracted with the JSBML library and returned to the calling controller. The GraphML service handles both input and output of GraphML files. When it receives a collection of `FlatEdges` with referenced `FlatSpecies` entities, it creates the corresponding GraphML representation that is sent to the client. On receiving the contents of a GraphML file that was uploaded to SBML4j, it creates a collection of `FlatEdges` and `FlatSpecies` and returns them to the calling controller or service.

Files with comma-separated values are processed by the `CSV` service. It uses the `univocity` parser[110] to process the rows of the input file. To define which column in the file matches the `FlatSpecies` symbol attribute in the network two options are available. Either the client provides a column name in the REST request, or the column name is preconfigured in the SBML4j configuration using the option *sbml4j.csv.matching-column-name*.

The `KEGG REST` service provides methods for requesting data entries from the KEGG API. It gathers secondary names for genes and chemical properties for compounds and drugs during the creation of the SBML knowledge graph for an SBML model that contains *biomodels.net* qualifiers to KEGG identifiers.

3.3.5 Controller Layer

The `Controller` classes implement the API interfaces, which define the REST endpoints of the application. The first step for each endpoint is to handle the user, which is given in the header field of the request. Then the execution order of service methods is controlled, provenance connections are established and provenance metadata is added to the activity node responsible for the current request. Lastly, exceptions are handled and appropriate responses are built and returned to the client.

User Handling

SBML4j defines a configurable *public user*, whose data is available for all users of the service. This user owns all publicly available networks from which any user can derive their own private network instances. If a request omits the user header, this *public user* will be used for this request. Any network created in this way will therefore also be owned by this *public user* and will be available for all users of the service. It is important to note that these public networks can only be deleted if the configuration is set to allow it. More information about

this and other configuration options can be found in Section 3.3.7. If a client requests to use a network, which is attributed to another non-public user, the access is denied and a `badRequest` response is returned. The `reason` header then contains the message "User xy is not authorized to access network", while keeping the identity of the owning user disclosed.

Provenance Tracking

During the execution of processing steps for a request, the controllers are responsible for tracking the provenance of the entities that are created.

This involves logging of the request parameters and the provided data by the client. This data is added as provenance annotation to the `Activity` node, which represents the current task in the data model. The structure of the annotation is illustrated in Listing 3.16 on the example of creating a neighborhood network, also called a context network. All provenance information is collected in a `java.util.HashMap`. The body of the request, the name and UUID of the newly created network are stored directly as string variables. The map keys for the parameters of the request are prefixed with "params" to indicate their nature. This notation will result in the creation of an additional Map named "params", when the provenance information is extracted again from the Activity node. This way, the conversion to a JSON-formatted object is straightforward when building the provenance report of the network later on.

As the last step, the map is provided as an argument to the `addProvenanceAnnotation` method alongside the `Activity`-node instance, which is to be annotated. This adds each element in the map to the `provenance` map of the receiving node.

Exception Handling

All exceptions thrown by the services are caught in the controller methods. The controller then returns an `Error 400 Bad Request` to the client. The exception message gets relayed to the client in the `reason` header of the response. Details on the possible response types of the different endpoints can be found in the OpenAPI documentation of SBML4j[111].

3.3.6 REST API

SBML4j as a service-oriented application offers a diverse set of REST endpoints to provide access to the application functionality. Together, these endpoints form the API of SBML4j. The API has been specified according to the OpenAPI Specification version 3.0.3. It has been designed using the toolset available at swagger.io[112] and stored in a text-based definition file. The complete definition file can be found in the `sbml4j-compose` GitHub repository[113]. Additionally, the API documentation of SBML4j can be accessed online at swaggerhub[111]. A list of available API endpoints can be found in appendix in Table D.2. This section is going to focus on the important endpoints and highlight relevant aspects of the API.

```
1   Map<String, Object> provenanceAnnotation = new HashMap<>();
2
3   //   body
4   provenanceAnnotation.put("body", bodyString);
5   //   networkname
6   provenanceAnnotation.put("networkname", newNetworkMappingName);
7   //   uuid
8   provenanceAnnotation.put("networkUUID", newNetworkEntityUUID);
9   // request parameters
10  //   base uuid
11  provenanceAnnotation.put("params.UUID", uuid);
12  // minSize
13  provenanceAnnotation.put("params.minSize", minDepth);
14  // maxSize
15  provenanceAnnotation.put("params.maxSize", maxDepth);
16  // terminateAt
17  provenanceAnnotation.put("params.terminateAt", terminateAtString);
18  // direction
19  provenanceAnnotation.put("params.direction", directionString);
20  // weightproperty
21  if (weightproperty != null)
22    provenanceAnnotation.put("params.weightproperty", weightproperty);
23  //   prefixName
24  if (prefixName != null)
25    provenanceAnnotation.put("params.prefixName", prefixName);
26
27  this.provenanceGraphService.addProvenanceAnnotation(activity, provenanceAnnotation);
```

Listing 3.16: Sequence of steps for tracking the provenance of the creation of a context network.

The User Header

Most API endpoints have the optional header field named `user`. If any string is provided, it will be used as the name for the provenance `Agent`, to which the created networks will be attributed. Any subsequent request that aims to use this network will have to provide the same `user` header value. In case this header value is omitted any network created will belong to the public user and is accessible to every other user that accesses the service.

SBML API

The SBML API is dedicated to processing SBML files for creating the knowledge graph in the database. As each SBML file creates new `Pathway` entities, the request is modeled as a `POST`-request. The important aspect of the `/sbml` endpoint is the necessity of transporting multiple xml files in the content of the request (see Listing 3.17). In the `OpenAPI Spec 3`, file transfers are realized with the content type `multipart/form-data`. To be able to upload multiple files with one request, a property of type `array` has to be provided in the specification. The framework component for the RESTful interface in Java Spring translates this content to a list of elements of type `MultipartFile`, where each such element represents one provided SBML file. In addition to the SBML files and the `user` header field, this API endpoint requires the parameters `source` and `version` for the name and version of the file source respectively,

```
1   requestBody:
2     description: The xml files containing the sbml models
3     required: true
4     content:
5       multipart/form-data:
6         schema:
7           properties:
8           # The property name 'files' will be used for all files.
9             files:
10              type: array
11              items:
12                type: string
13                format: binary
```

Listing 3.17: OpenAPI 3.0 definition of the request body of the `POST` `/sbml` request.

as well as a three-letter organism code in the `organism` parameter. These parameters for defining the file source and organism are used to identify the `DatabaseNode` and `Organism` node to which the provenance information of this file is linked.

For retrieving information on the stored SBML model components, two sets of endpoints are included in the SBML API. The first set consists of a `GET` and a `POST` request to the URI `/entityInfo`. Both endpoints take individual or multiple `symbols`, either in the query part of the request or in the body of the request, respectively. They provide a JSON-formatted information object, which gives detailed information about the known properties of SBML knowledge-graph elements that are found for the provided symbols. The information includes reactions and transitions the entity takes part in as well as the biomodels.net qualifiers that are associated with this entity. While the `GET` request is provided for retrieving information about individual entities or a small number of entities, the `POST` request can be used for `batch` processing of a large number of symbols.

The second set consists of a `GET` and a `POST` request to the `/idMap` endpoint. They can be used to retrieve source-specific `IDs` to given symbols that are available in the SBML knowledge graph. If an entity is associated with the provided symbol, the biomodels.net qualifiers which are connected to this entity are searched for database IDs. The tuple of qualifier and ID is then returned. If the request limits the requested IDs to certain systems, only those IDs will be returned, otherwise all found IDs are returned. Again, the `GET` endpoint is provided for individual requests, while the `POST` endpoint is for batch processing.

Pathway API

The Pathway API offers a `GET` request for listing all available pathways and collections of pathways which can be used to create network mappings. The creation of a mapping can be triggered with a `POST` request to the endpoint `/mapping/{UUID}`, where the `UUID` path parameter must be the `entityUUID` of a pathway. The required query parameter `mappingType` denotes which type of network mapping should be created from the given pathway. Further

optional parameters for defining the name of the resulting network can be provided. The reader is referred to the REST API documentation[111] for details.

Network API

The Network API offers a variety of different endpoints for listing available networks, applying analysis steps to networks, providing and retrieving provenance information to networks and retrieving the contents of networks. An overview of available networks can be requested with a `GET` request to `/networks` without any query parameters.

If the user header is omitted, only networks owned by the public user will be listed. When the user header is provided, the response includes networks that are attributed to the provided user and are not available to other users of the service. With a `POST` request to the endpoint `/networks?parentUUID={UUID}`, a copy of the network provided by the `parentUUID` can be created.

All other requests in this API use the `{UUID}` as path parameter and operate with or on the network given by this UUID. SBML4j defines an API endpoint for each operation that can be performed on a network. A `GET` request to `/networks/{UUID}` will download the network denoted by this `UUID` in the GraphML format, if the network belongs to the user that is provided in the appropriate header of the request or is owned by the public user. A network can be deleted with a DELETE request to the same endpoint. Only networks owned by the provided user can be deleted with that request unless SBML4j is configured to allow deleting of public networks. A `GET` request to the `/networks/{UUID}/options` endpoint will send the available filter and annotation options for the network that is denoted by the provided `UUID`. The filter options can be sent in a `POST` request to the `/networks/{UUID}/filter` endpoint to create a filtered subnetwork of the network with given `UUID`. In the same manner, the annotation options can be used in a `POST` request to the `/networks/{UUID}/annotation` endpoint to add arbitrary data onto the nodes and edges of the network with given `UUID`. Unless otherwise requested, both the `filter` and the `annotation` requests result in a new network resource being created that receives a new `UUID` to be used in follow up requests. Annotation data can also be added to a network from a CSV file by issuing a `POST` request to the `/networks/{UUID}/csv` endpoint, using the given or configured column in the CSV file to match data entries to network nodes. Lastly, the `/networks/{UUID}/context` endpoint can receive both a `GET` and a `POST` request. For a `GET` request, a context network for the provided request parameters is calculated on the network with given `UUID` and immediately return to the client as GraphML content. No new network resource is created in SBML4j. In contrast, a `POST` request triggers the same calculation of a context network for the same parameters, but instead of returning the created resource, it creates a new network instance in the database that contains the context network and returns the `NetworkInventoryItem` of that new network.

This new context network can then be fetched repeatedly with GET requests using the new UUID that is found in the NetworkInventoryItem without having to recalculate its contents. For more details on these requests, the reader is referred to the official documentation of the API[111].

Overview API

For workflow convenience, the endpoint /overview with a GET and POST method bundles the extraction and creation of neighborhood networks with the annotation of provided nodes by a named Boolean annotation. It can be used in conjunction with dedicated configuration properties in workflows and applications that specifically aim at generating neighborhood networks for defined sets of genes (e.g., Drivergene networks) with a single REST call.

GraphML API

The GraphML API is dedicated to receiving GraphML files through a POST request to /graphml. For each file uploaded to this endpoint, a network is created that contains the nodes and edges that are defined in the xml document. If the parentUUID parameter is set, SBML4j attempts to connect the newly created network to the network with the given UUID with a *wasDerivedFrom* relationship to indicate provenance. This parent network has to be accessible by the provided user, which means it must either be owned by that user or the public user.

Warehouse API

The Warehouse API allows the retrieval of UUIDs for two types of warehouse nodes. First, the UUID of a database node with given name and version can be extracted with a GET request to /databaseUUID. Second, the UUID of a FileNode that is the parent entity of the network or pathway with the provided UUID can be requested by using GET /fileorigin. With the received UUIDs from each of the endpoints, provenance information can be added to these entities when used in the Provenance API.

Provenance API

The Provenance API features the /prov/{UUID} endpoint, which accepts GET and PUT requests. With a GET request the provenance report for a given Warehouse entity (i.e., Pathway, Network) can be downloaded in a JSON-formatted response. A PUT request will add the provided JSON-formatted request body to the provenance information of the entity with given UUID.

Table 3.2: Important Configuration properties in SBML4j.

prefix	property	description
context	minSize	Minimum number of relations to traverse for the neighborhood path search
context	maxSize	Maximum number of relations to traverse for the neighborhood path search
context	terminateAt	Nodelabel which terminates the paths in the neighborhood path search
context	direction	Direction in which relations can be traversed in the neighborhood path search
annotation	append	Boolean to control whether existing annotations should be appended (TRUE) or replaced (FALSE)
network	public-user	Name of the public user to use for this instance
csv	matching-column-name	Name of column in csv files, that will be used to match gene symbols for annotation (can have multiple entries)

3.3.7 Configuration

The configuration is handled via the `Spring` configuration properties. These properties are defined using the `@ConfigurationProperties` annotation on specific classes. The configuration values are accessible through the `Sbml4jConfig` class, which gets injected into services and controllers that need to access them.

Table 3.2 lists some important configuration properties with the prefix they use and a short description of their meaning. The values for these properties can be supplied by an *application.properties* file or through environment variables. The environment variables will override previously defined properties in the properties files. SBML4j supports different configurations for `development`, `test`, and `production` environment profiles. By adding a suffix with the respective short handle string to the filename of the application.properties file, each profile can be configured separately, for instance `application-dev.properties`.

In addition to the application-specific configuration properties, SBML4j makes use of built-in properties for defining the application context path and the connection details for the Neo4j database (see Tab. 3.3). A full properties file can be found in the appendix as Listing E.2. Detailed descriptions of the properties not mentioned here can be found in the documentation of SBML4j[114].

Table 3.3: Spring Framework and server configuration properties used in SBML4j.

prefix	property	description
spring.data.neo4j	uri	URI of the Neo4j database to use in the format protocol://host:port (example: bolt://localhost:7687)
spring.data.neo4j	username	The username to use for the Neo4j database
spring.data.neo4j	password	The password to use for the Neo4j database
server.servlet	contextPath	The contextPath of the tomcat server that handles the HTTP requests
server.servlet.session	timeout	Define a timeout for client sessions (use -1 for no timeout)
server	port	The port to be used by the tomcat server for accepting requests)

3.3.8 The Python Client Library `pysbml4j`

In addition to the SBML4j server application, the Python client module `pysbml4j` has been developed as part of this software suite. This section gives some insights into the design considerations and introduces the three components that make up the client.

Design Considerations

The main focus for the Python client library design was the usability aspect. A potential user will usually create a network from some pathway or collection of pathways, or use a pre-built network that is already available. With this, the most prominent tasks will be filtering, annotation and context generation. All these activities should be achievable with just a few lines of code and without having to refetch newly created networks between individual steps. The main return type for network data from SBML4j is the `GraphML` format, which can be used in existing network libraries that are available in Python, like networkx[115]. This eliminates the necessity to reimplement the features that are offered by these libraries, which makes it possible to keep the SBML4j Python client library compact. This improves maintainability and reduces the number of bugs that could potentially be introduced in an implementation. For ease of use, the configuration of the client instance should be straightforward and the default option should cover the most frequently used setup.

Table 3.4: REST endpoints of SBML4j and the respective pysbml4j method in the `Sbml4j` class. The request type and the endpoint URI with the Python method that targets the endpoint and the type of the returned value.

Request type	Endpoint URI	Return type	Python method
GET	/**networks**	json	listNetworks
POST	/**sbml**	json	uploadSBML
POST	/**graphml**	json	uploadGraphML
GET	/**pathways**	json	pathwayList
POST	/**pathwayCollection**	json	createPathwayCollection
POST	/**mapping**/{UUID}	json	mapPathway

The Configuration Class

Each instance of the SBML4j client has to be instantiated with a `Configuration` object, where a default one is created if it is omitted. The default configuration will resolve to a server exposed at the URL `http://localhost:8080/sbml4j`, which is also the default configuration of the SBML4j server application. The URL is composed of the server address (i.e., http://localhost), the port 8080 and the application context (i.e., /sbml4j). Each of these can be passed to the constructor of the Configuration class to change the server details. In addition to the *URL*, the `Configuration` class allows changes to the headers of the requests that are sent to the server. This is used internally for requests that send JSON-content to the server which sets the *Content-type*-header to *application/json*. The header value for providing a `user` to SBML4j endpoints can be set directly using the provided setter method.

Whenever a change to the server parameters occurs, the internal Boolean `_isInSync` is set to `False`. All requests made to the server check the status of this flag. If the value of the flag is `False`, the list of networks is refreshed before the request is issued. This ensures that the client has an up-to-date view of the networks that are available on the server for the current user.

The Sbml4j Class

The `Sbml4j` class keeps a list of available networks, executes the http-calls to the server instance and processes the responses. It uses the `urllib3` library[116] for making the REST calls and implements dedicated methods for the RESTful endpoints of the SBML4j server.

The methods of the `Sbml4j` class and their respective REST endpoints are listed in Table 3.4. All `Pathway`-related tasks are handled directly by this class, as well as listing available networks and submitting GraphML and SBML files to the server. The `Sbml4j` object keeps a list of networks that are available for the currently configured user. To ensure that the information is

```
1    def checkSyncStatus(self):
2          if not self._configuration.isInSync:
3              self.refreshNetworkList()
4              self._configuration.isInSync = True
```

Listing 3.18: Python method `checkSyncStatus` of the SBML4j class in pysbml4j to keep the internal network list in sync with the server data.

```
1    def getNetworkByName(self, name):
2          self.checkSyncStatus()
3          networkInfoDict = self._nameToNetworkMap[name].getInfoDict()
4          return Network(networkInfoDict, self)
```

Listing 3.19: Python method `getNetworkByName` of the SBML4j class in pysbml4j to select networks by their name.

up-to-date, each method that uses or manipulates this list, has to check whether the list is still in sync to the server data (Listing 3.18). The Boolean `isInSync` is kept in the configuration and gets set to `False` whenever the configuration changes, for example when the user-header value is changed.

The method `Sbml4j.getNetwork(UUID)` can be used to retrive a `Network` object from the SBML4j server. It requires a network UUID as an argument. Since UUIDs are inconvenient to use for the end user, `pysbml4j` instead operates on the names of the networks. For that, the Sbml4j class keeps a dictionary matching the names of networks to the network-objects. This allows a user to select networks based on their name instead of using the `UUID` using the method `Sbml4j.getNetworkByName(name)` (see Listing 3.19).

```
1    class Network(object):
2        uuid: str
3        name: str
4        organismCode: str
5        numberOfNodes: int
6        numberOfRelations: int
7        numberOfReactions: int
8        nodeTypes: list
9        relationTypes: list
10       networkMappingType: str
11
12       sbml4jApi = None
13
14        def __init__(self, dict_from_api, api):
15            self.sbml4jApi = api
16            self.updateInfo(dict_from_api)
```

Listing 3.20: Network object of pysbml4j with available fields and constructor.

```
1    def updateInfo(self, dict_with_info):
2        self.uuid = dict_with_info['uuid']
3        self.name = dict_with_info['name']
4        self.organism_code = dict_with_info['organismCode']
5        if dict_with_info['numberOfNodes'] != None:
6            self.numberOfNodes  = dict_with_info['numberOfNodes']
7        else:
8            self.numberOfNodes = 0
9        try:
10           self.numberOfRelations = dict_with_info['numberOfRelations']
11       except:
12           self.numberOfRelations = 0
13       try:
14           self.numberOfReactions = dict_with_info['numberOfReactions']
15       except:
16           self.numberOfReactions = 0
17
18       self.nodeTypes = dict_with_info['nodeTypes']
19       self.relationTypes = dict_with_info['relationTypes']
20       self.networkMappingType = dict_with_info['networkMappingType']
```

Listing 3.21: Python code of the method for updating the contents of a Network object.

Network Object

The `Network` object (Listing 3.20) is the main element of interaction for a client. It contains all fields that are found in the `NetworkInventoryItem` and a reference to the SBML4j instance it originated from. The constructor requires such an instance in the `api` parameter and a dictionary element containing the initial values for the inventory values.

Although many operations create new network instances in the database, the user can continue to work with the original `Network` object, as it gets updated with the new information. This way, multiple network-manipulation steps can be performed on a single Python object, without the need to introduce a new object for the result of each step. This is done internally by calling the method `updateInfo` (see Listing 3.21). The argument needs to be a dictionary object, which contains the elements from the `data` element of the response of the initial request. All endpoints that modify or create a network in SBML4j return the `NetworkInventoryItem`, which contains all the necessary elements for this update method. Table 3.5 lists the methods on the `Network` object that directly target specific endpoints of the SBML4j REST API. In addition, the convenience method `shortestPath` is available that allows direct calculation of a shortest path on a given network.

3.3.9 Deployment

The SBML4j server application is maintained by the "kohlbacherlab" organization on GitHub[114]. It is provided as an open-source application with an `MIT license` from the "Open Source Initiative"[117].

Table 3.5: REST endpoints of SBML4j and the respective pysbml4j method in the `Network` class. The request type and the endpoint URI with the Python method that targets the endpoint and type of the returned value. Return types with value *Network* update the current Network object to contain the newly created network in SBML4j.

Request type	Endpoint URI	Return type	Python method
POST	/**networks**	Network	copy
GET	/networks/**{UUID}**	GraphML String	graphML
POST	/networks/{UUID}/**csv**	Network	addCsvData
GET	/networks/{UUID}/**context**	GraphML String	getContext
POST	/networks/{UUID}/**context**	Network	postContext
GET	/networks/{UUID}/**options**	json	getOptions
POST	/networks/{UUID}/**filter**	json	filter
POST	/networks/{UUID}/**annotation**	Network	annotate
PUT	/**prov**/{UUID}/	json	addProvenance
GET	/**prov**/{UUID}/	json	getProvenance

The java application is managed by apache maven[118] and can be built into a Java ARchive (JAR file), which can be executed with locally installed Java runtime environments. SBML4j is also available as pre-built docker image, which can be found at DockerHub[119].

The necessary Neo4j database instance is not part of SBML4j and a running instance needs to be provided before the start of the application.

The recommended deployment method is via `docker compose`. The project on GitHub, `sbm4j-compose`[113], bundles the configuration of SBML4j, the startup and controlled termination of the SBML4j container as well as a Neo4j container, and additionally provides a shell script for downloading the docker images, setting up the volumes and managing the database dumps. An example `docker-compose.yml` file can be found as supporting Listing E.1 in the appendix. As a further benefit from the compose setup, it can initialize and start a `swagger-api` container[120] that provides the `Swagger UI`[121], an interactive API documentation. The OpenAPI definition file that is part of the docker-compose GitHub repository can be customized to fit the server address of an individual setup. This enables the real-time use of the REST API through the interactive API documentation.

The Python client library `pysbml4j`[101] is available on the Python Package Index[122] and can be installed via the package installer for Python[123]. The source code is maintained as a project on GitHub[124].

3.4 Results

To demonstrate the feasibility of the SBML4j software three use-cases are presented: Finding upstream driver genes for a given input gene, revealing a gene neighborhood network for multiple genes and finding a shortest path between two input genes. The use-cases include simple python code blocks that demonstrate the use of these methods along with different visualization options for the networks using the packages networkx and ipycytoscape, the Neo4j Browser that comes with the database itself and the graph drawing tool BiographVisArt[125]. At the end of this section, the provenance report layout is presented.

3.4.1 KEGG Knowledge Graph

For these examples the list of KEGG pathway maps shown in tables 3.6 and 3.7 were used. They were downloaded in the `KGML` format from the KEGG web service. The KEGG pathway database release was: Release 97.0+/02-16, Feb 21. Those KGML files were translated to SBML using KEGGtranslator version 2.5[126,127]. The command line arguments for translation with KEGGtranslator are listed in the appendix under Listing E.3. Those translated SBML files were then uploaded to an empty SBML4j instance using Python and pysbml4j (see Listing E.5 in the appendix for a source code example). The entities that were created from the SBML knowledge graph serve as the basis for the network mappings used in the examples below.

3.4.2 KEGG Network Mappings

From these 61 pathways a collection pathway was created using the `/pathwayCollection` endpoint. Then regulatory, signaling, metabolic and protein-protein interaction network mappings were created from this collection, named REG-KEGG61-97.0, SIG-KEGG61-97.0, MET-KEGG61-97.0 and PPI-KEGG61-97.0 respectively. Table 3.8 lists the number of nodes and relationships in the different mapping types created in that manner. These network mappings were used in the following examples.

3.4.3 Shortest-path

Figure 3.15 (a) shows the shortest path between the two genes CCND2 and PTEN in a protein-protein interaction network visualized using the ipycytoscape widget. The Python code used to generate the network (excluding the visualization) can be found in Figure 3.15 (b).

3.4.4 Gene Neighborhood

Given a set of gene symbols, a context network was calculated. It contains all genes matching the given symbols, all connecting paths between those genes and all direct neighborhoods of

Table 3.6: Non-disease related pathways used in the results section. Listed are the entry identifiers and associated definitions from the KEGG pathway database – Release 97.0+/02-16, Feb 21.

identifier	definition
hsa03320	PPAR signaling pathway
hsa04010	MAPK signaling pathway
hsa04012	ErbB signaling pathway
hsa04014	Ras signaling pathway
hsa04015	Rap1 signaling pathway
hsa04020	Calcium signaling pathway
hsa04022	cGMP-PKG signaling pathway
hsa04024	cAMP signaling pathway
hsa04060	Cytokine-cytokine receptor interaction
hsa04064	NF-kappa B signaling pathway
hsa04066	HIF-1 signaling pathway
hsa04068	FoxO signaling pathway
hsa04070	Phosphatidylinositol signaling system
hsa04071	Sphingolipid signaling pathway
hsa04072	Phospholipase D signaling pathway
hsa04080	Neuroactive ligand-receptor interaction
hsa04110	Cell cycle
hsa04115	p53 signaling pathway
hsa04150	mTOR signaling pathway
hsa04151	PI3K-Akt signaling pathway
hsa04152	AMPK signaling pathway
hsa04210	Apoptosis
hsa04218	Cellular senescence
hsa04310	Wnt signaling pathway
hsa04330	Notch signaling pathway
hsa04340	Hedgehog signaling pathway
hsa04350	TGF-beta signaling pathway
hsa04370	VEGF signaling pathway
hsa04371	Apelin signaling pathway
hsa04390	Hippo signaling pathway
hsa04510	Focal adhesion
hsa04512	ECM-receptor interaction
hsa04520	Adherens junction
hsa04630	JAK-STAT signaling pathway
hsa04915	Estrogen signaling pathway

the given genes. First the input genes are sorted alphabetically and the first two genes are connected using a shortest path search. The sorting step is necessary to have a determinism regarding the choice of the first nodes to connect. All found genes or nodes and the relationships on this path are added to the result set. For each remaining gene, the shortest path to any

Table 3.7: Human Disease pathways related to cancer used in the results section. Listed are the entry identifiers and associated definitions from the KEGG pathway database – Release 97.0+/02-16, Feb 21.

identifier	definition
hsa05200	Pathways in cancer
hsa05202	Transcriptional misregulation in cancer
hsa05203	Viral carcinogenesis
hsa05204	Chemical carcinogenesis
hsa05205	Proteoglycans in cancer
hsa05206	MicroRNAs in cancer
hsa05210	Colorectal cancer
hsa05211	Renal cell carcinoma
hsa05212	Pancreatic cancer
hsa05213	Endometrial cancer
hsa05214	Glioma
hsa05215	Prostate cancer
hsa05216	Thyroid cancer
hsa05217	Basal cell carcinoma
hsa05218	Melanoma
hsa05219	Bladder cancer
hsa05220	Chronic myeloid leukemia
hsa05221	Acute myeloid leukemia
hsa05222	Small cell lung cancer
hsa05223	Non-small cell lung cancer
hsa05224	Breast cancer
hsa05225	Hepatocellular carcinoma
hsa05226	Gastric cancer
hsa05230	Central carbon metabolism in cancer
hsa05231	Choline metabolism in cancer
hsa05235	PD-L1 expression and PD-1 checkpoint pathway in cancer

Table 3.8: Number of nodes and relationships for KEGG Mappings derived from 61 KEGG pathways used in this work.

Mapping Type	Number of nodes	Number of relations
PROTEIN-PROTEIN INTERACTION	671	1311
SIGNALING	1369	2457
REGULATORY	1521	2773
METABOLIC	107	221

gene in the result set is searched and the nodes and relationships of that path are added to the resulting network. This might not ensure the smallest possible network to connect all given input genes, but it guarantees to find one if all input genes are on one connected graph. If a gene cannot be connected to the network, it is dropped from the result set. Figure 3.16 (a)

b

```
1 from pysbml4j import Sbml4j
2 client = Sbml4j()
3 network = client.getNetworkByName("PPI-KEGG61-97.0")
4 spNetGraphML = network.shortestPath("CCND2", "PTEN")
```

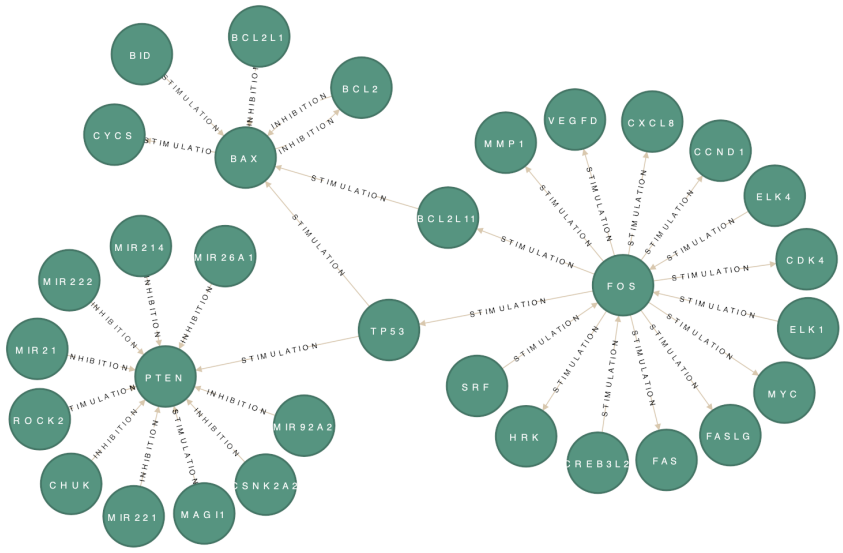
Figure 3.15: Shortest-path network between the genes CCND2 and PTEN in a PPI network with respective python code. (a) Shortest-path network between the genes CCND2 and PTEN in a protein-protein-interaction network visualized using the ipycytoscape widget in an IPython notebook. (b) Complete code example for retrieving the shortest path between the two genes in a protein-protein-interaction network. Line 1 imports the `Sbml4j` client connector from the pysbml4j package, line 2 establishes a connection to a local instance of SBML4j running at the default address of *http://localhost:8080/sbml4j*. In line 3 a network is selected by its given network name and stored in the `network` variable. The `shortestPath` method on the `network` that starts in line 4 queries for a context network connecting the two input genes with the shortest path assuming a default weight of 1 on each relationship that is traversed. The resulting graph is reported in the GraphML format.

shows the gene neighborhood network for three genes FOS, PTEN and BAX in the signaling network created above. The Python source code used to generate the network can be seen in Figure 3.16 (b).

3.4.5 Upstream Cancer Driver Genes

To elucidate disease driving mechanisms in rare case cancer patients it is often useful to find cancer driver genes that are located upstream in the regulatory network of an affected gene or genes that are specific to this patient. SBML4j enables the exploration of regulatory networks in that manner in a few simple steps. After choosing the regulatory network of interest as a base network, driver gene information can be added to it using a simple `csv` file linking the gene symbol to the driver type and further optional information. One such source can be the driver gene atlas by Vogelstein et.al[128]. By providing this list of genes in the `csv` file format to SBML4j, the driver gene information can be added on the network nodes, marking them as *Driver*. This keyword can then be used as termination criterion for an expanding path search in the upstream-regulation direction, which results in a network tree rooted in the given genes with the leaves being the driver genes. Figure 3.17 (a) shows the upstream driver gene network for the Gene MMP9 visualized by BiographVisArt. The respective Python code for uploading the csv file and retrieving the network is shown in Figure 3.17 (b).

ⓐ

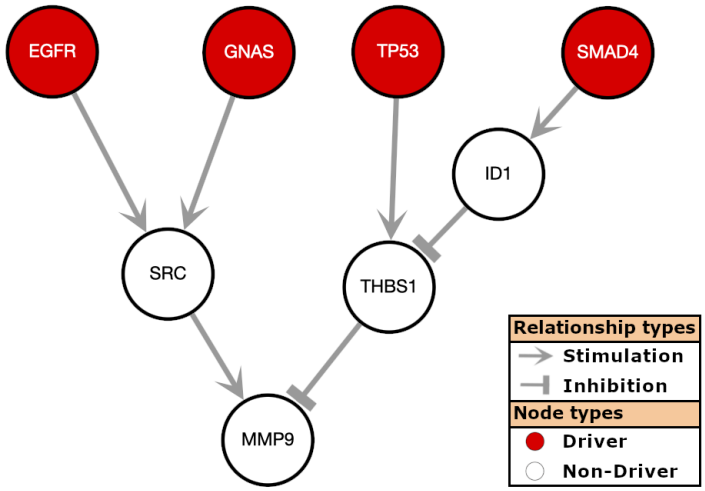


ⓑ

```
1 from pysbml4j import Sbml4j
2 client = Sbml4j()
3 network = client.getNetworkByName("SIG-KEGG61-97.0")
4 network.createContext(geneList=["FOS", "PTEN", "BAX"],
5                       direction="both",
6                       maxSize=1,
7                       newNetworkName="MyGeneNeighbourhood")
8 nhNet = client.getNetworkByName("MyGeneNeighbourhood")
9 nhNetGraphML = nhNet.graphML()
```

Figure 3.16: Multi-gene neighborhood network visualization from the Neo4j Browser and python code for generating the network. (a) Multi-gene neighborhood network for the genes FOS, PTEN and BAX on a signaling network with specific interactions on the relationships visualized in the Neo4j Browser. (b) Complete code example for generating the shown gene neighborhood network from a KEGG-based signaling network. Line 1 imports the `Sbml4j` client connector from the pysbml4j package, line 2 establishes a connection to a local instance of SBML4j running at the default address of *http://localhost:8080/sbml4j*. In line 3 a network is selected by its given network name and stored in the `network` variable. The `createContext` method on the *network* that starts in line 4 triggers the creation of a new network instance in the database, but the response does not include the generated network itself. The provided `geneList` specifies the genes that are to be connected to a gene neighborhood. The `direction` parameter (line 5) denotes that relationships should be traversed in any direction (upstream and downstream), the `maxSize` parameter in line 6 limits the number of relations to traverse before stopping the context expansion when searching the direct neighborhood of the input genes. The optional parameter `newNetworkName` in line 7 will be set as the name for the newly created network. Line 8 receives the newly created network from the SBML4j service by its given name and in line 8 the graphML representation of the calculated network is requested from the service which is then saved in the variable *nhNetGraphML* for further use, i.e. visualization.

a



b

```
1  from pysbml4j import Sbml4j
2  client = Sbml4j()
3  network = client.getNetworkByName("REG-KEGG61-97.0")
4  network.addCsvData(csvFile="drivergenes.csv",
5                     dataName="Driver",
6                     newNetworkName="MyDrivergeneNetwork")
7  driverNet = client.getNetworkByName("MyDrivergeneNetwork")
8  upstreamNetGraphML = driverNet.getContext(geneList=["MMP9"],
9                                            terminateAt="Driver",
10                                           direction="upstream",
11                                           maxSize=3 )
```

Figure 3.17: Upload drivergene data from a csv file to a network and retrieve upstream hits for a given input gene. (a) Upstream driver gene network for the MMP9 gene on a regulatory network visualized using BioGraphVisArt. Driver genes are shown in red, non-driver genes in white. The arrows with pointed heads denote stimulation events while the ones with flat heads indicate inhibition events. (b) Complete code example for adding annotation from a csv file on a network and using this annotation to search for hits of the annotation type upstream of a given gene. Line 1 imports the `Sbml4j` client connector from the pysbml4j package, line 2 establishes a connection to a local instance of SBML4j running at the default address of *http://localhost:8080/sbml4j*. In line 3 a network is selected by its given network name and stored in the `network` variable. The `addCsvData` method on the *network* that starts in line 4 adds the annotation contained in the given file *drivergenes.csv*. The `dataName` parameter in line 5 gives the name for the label a node will receive when annotation is added to it. The optional parameter `newNetworkName` in line 6 will be set as the name for the newly created network. Line 7 receives the newly created network from the SBML4j service by its given name. The method `getContext` starting in line 8 takes the arguments `geneList` for providing one or more node symbols as starting points for context generation, `terminateAt` for providing the generated node label from the csv data to serve as stop sign for the context extension, `direction` for limiting the context extension to *upstream* relations and the `maxSize` parameter to limit the number of relations to traverse before stopping the context expansion. All paths that end in a node with the given node label that are `maxSize` steps away from any input gene in *geneList* are reported in a connected graph which is reported in the GraphML format.

3.4.6 Provenance Report

We downloaded the Notch Signalling Pathway from the KEGG pathway database (compare Fig. 2.2 in Chapter 2) in the KGML format, translated it to the SBML format using KEGGtranslator and uploaded the translated file to an emtpy SBML4j instance. We added provenance data to the resulting pathway and created a signalling network mapping from it. Here, we present the provenance report of this process.

The provenance report can be generated by sending a `GET` request to the `/prov/{UUID}` endpoint or by calling the Python method `getProvenance` while providing the `UUID` of the signalling network in question. The content of the report is the serialized version of the `ProvenanceInfoItem` (see Fig. 3.12) and is reported in JSON format.

For brevity, key elements of the report are presented here, while the full report can be found in the appendix as Listing E.4.

```
1   {
2       "type": "Network",
3       "contents": {
4           "uuid": "b5f3b7c9-820b-4e8e-8549-4d30eff67127",
5           "UUID": "b5f3b7c9-820b-4e8e-8549-4d30eff67127",
6           "name": "NOTCH_SIGNALLING_MAPPING",
7           "organismCode": "hsa",
8           "numberOfNodes": 22,
9           "numberOfRelations": 44,
10          "numberOfReactions": 0,
11          "nodeTypes": [
12              "polypeptide chain"
13          ],
14          "relationTypes": [
15              "INHIBITION",
16              "STIMULATION",
17              "PROTEINCOMPLEXFORMATION"
18          ],
19          "networkMappingType": "SIGNALLING"
20      },
```

Listing 3.22: Top Element of the Provenance report that describes the network's contents and its type

The top element of the report describes the network's contents and its type as shown in Listing 3.22. It also gives information about the user that is attributed to this network as well as the activity that generated the network. The information about the activity includes the type of REST call that has been made, the endpoint it was made to as well as the parameter values that were provided in the call (see Listing E.4 lines 25–58).

The next section of the report advances one level down the provenance hierarchy along the `wasDerivedFrom` provenance relation. It provides details about the warehouse entity that served as basis for the generation activity that has been described above and contains the same sections as the top level element, namely the content and type of the entity, which in this case

is a `Pathway` (see Listing E.4 lines 60–86). Again, this is followed by information about the user and the activity that created this entity, where in this case the latter is the `POST` request to the `/sbml` endoint, since the `Pathway` is created when the model in the uploaded file is persisted (see Listing E.4 lines 87–90 and 93–119).

```
1                       "provenance": [
2                           {
3                               "translation": {
4                                   "name": "KEGGtranslator",
5                                   "arguments": {
6                                       "gene-names": "FIRST_NAME",
7                                       "format": "SBML_CORE_AND_QUAL",
8                                       "add-layout-extension": "FALSE",
9                                       "use-groups-extension": "FALSE",
10                                      "remove-pathway-references": "TRUE",
11                                      "remove-white-gene-nodes": "TRUE",
12                                      "autocomplete-reactions": "TRUE"
13                                  },
14                                  "version": "2.5"
15                              }
16                          },
17                          {
18                              "origin": {
19                                  "Download date": "Oct 18, 2022 19:15:42 +0000 (UTC)",
20                                  "Creation date": "Oct 3, 2022 16:25:37 +0900 (GMT+9)",
21                                  "filename": "hsa04330.xml",
22                                  "source": "KEGG Pathways database",
23                                  "version": "104.0"
24                              }
25                          },
26                          {
27                              "SBML4j": {
28                                  "version": "1.2.2"
29                              }
30                          }
```

Listing 3.23: Provenance annotation data on the `File` entity in the Provenance report

The next level along the `wasDerivedFrom` provenance relation is dedicated to the SBML file. It starts with the provenance information that has been added after the uploading process and can be seen in Listing 3.23. It contains three subsections: `translation`, `origin`, and `SBML4j`. The `translation` section gives details about the tool that has been used to translate the input file and contains the name, the version and the runtime arguments that were provided to that tool (see Listing 3.23 lines 3–15). Then, the origin of the translation process is given in the `origin` section (see Listing 3.23 lines 18–25). It contains the name and version of the file source, the filename, as well as the original creation date of the file and the date of the download. Lastly, provenance information about `SBML4j` itself has been added to the report.

In this case it consists of the software version that has been used to create this `File` in the database (see Listing 3.23 lines 27–29).

Following this provenance information is the metadata about the SBML file itself (see Listing E.4 lines 154–159), which contains the name of the file and its computed MD5 sum. The file has been created by the same activity as the pathway entity, which is repeated at this point in the report for completeness (see Listing E.4 lines 164–193).

The final level of the JSON hierarchy describes the `Database` entity that is at the basis of this provenance and is described by the `source` and `version` that was provided in the `POST` request when uploading the SBML file. Again, it was created by the same activity, whose info is repeated on this level for completeness.

This report provides all available provenance information on the network resource, which traces back the creation steps to the original source and gives details about transformation steps and data origin where available. This type of report is available for every network in SBML4j. The comprehensiveness of the provenance information in such reports depends on the amount of data that is provided for the indivdual steps. The sections `wasDerivedFrom`, `wasAttributedTo` and `wasGeneratedBy` are automatically populated when working with the data in SBML4j. The additional data in the `provenance` subsections have to be provided by external clients that manipulate or describe the data within SBML4j.

3.5 Discussion and Conclusion

We presented SBML4j, a service-oriented software for storing, manipulating, exploring and retrieving of biological networks and biomedical data. By offering full functionality through a RESTful interface and Python client library no individual database queries are necessary to populate the database, annotate and manipulate the networks or retrieve network resources. This provides a standardized method to interact with biological networks and gives the ability to add reproducible biological network knowledge to research projects and clinical decision support systems.

A graph database for storing biological-network data enables the native representation of the data by the database model. Further, it enables the application of graph algorithms on the database level. For the traversal of the networks and calculation of neighborhoods, SBML4j uses the APOC extension to Neo4j. More recently the `Graph Data Science`[129] library is offering comparable algorithms on in-memory representations of the graph, which can potentially speed up these calculations.

This thesis puts a focus on the KEGG pathway database as a source for the biological models. Given adequate SBML models of other data sources, the software presented here can be used to build network mappings for those. The mapping process in SBML4j relies on the biomodels.net qualifier annotations and SBO terms. These elements are often missing in models from other

sources, like PathwayCommons, Reactome or WikiPathways. With a growing interest in systems biology, the amount of well annotated models will grow and more input sources will become available in the future.

With improvements in the frameworks and tools that SBML4j relies on, many enhancements can be implemented in the future. A next-generation update of SBML4j can, for instance, move the UUID management into the database, since newer versions of the Neo4j database now support these types. Additionally, a change to the reactive query layout, with implications for all layers that make use of query results as well as the API request and response mechanics. Due to the graph structure of the database, as well as the stored data, it is also recommended to investigate whether a GraphQL[130]-based layout of the API is to be preferred over the REST layout that SBML4j currently uses. The newly introduced *auditing* features of the Spring modules are also a good extension of the provenance-tracking mechanisms, which make the recording of changes more reliable and easier to track.

For the purposes of this thesis, the aspect of user authentication, which is critical for production use of SBML4j, has been kept aside. While keeping user-generated networks separated by attributing them to the provided username in the header of the request is working for prototyping use, in a production environment this needs to be combined with proper user authentication. Any properly authenticated user, for instance through OAuth2[131] can then be attributed to their networks and the data kept truly secure.

To speed up database queries, advanced indices are needed on combinations of labels and properties. Unfortunately, these types of indices are not supported by the free community edition of the Neo4j database, but are only available via a paid subscription to the enterprise version. Here, an evaluation of a potential speedup from these composite indices and keys warrants the costs of an enterprise license. Alternatively, the underlying graph database could be changed to an unrestricted open-source one. In the past years significant advancements have been made to several open-source graph database solutions, for instance DGraph, ArangoDB or MongoDB, which need to be evaluated towards their viability for this application.

SBML4j creates a simplified SBML knowledge graph through its data model and the database entities that are persisted from it by the OGM. For the current use cases, generating mostly non-metabolic network mappings for investigating qualitative features of the biological processes, this simplified model is sufficient. For future expansions it should be investigated whether more features of the SBML core model, as well as more SBML extensions, should be incorporated into the SBML4j data model. One example is the `groups` extensions, which encodes the connections between a group and its member entities in a structured manner, compared to the current encoding in annotation objects of the groups. Furthermore, it is possible to make use of the data model from jSBML directly by extending the jSBML classes as `Node` or `Relationship` entities and providing conversion classes for data types that are not supported by Neo4j. Additional investigations should be made to evaluate if this approach leads to a

more detailed reflection of the SBML data in the graph database, without the need to manually model all parts and translate between the two data models.

Moreover, the creation of physical copies or derivatives of network mappings for each modification step is beneficial to recording the provenance of the networks. However, this leads to the growth of the database over time and a potential decline in query performance. A detailed comparison about the alternative use of labels for denoting network membership and additional properties for keeping track of the provenance needs to be performed. Advancements in the database driver and OGM could make this approach more feasible now.

To enable performant graph traversals and effective use in graph algorithms, the `FlatEdge` entities are modeled as Java Classes and persisted with the respective relationship type as label. This necessitates the implementation of more classes to be able to reflect further relationship types. An investigation into alternative means of representing the relationship types for network mappings should be carried out. Possible scenarios include the definition of the relationship types using external XML files, which can be dynamically loaded, or the further incorporation of the SBO terms directly as relationship types as they are provided by jSBML.

The biomodels.net qualifier types that are used for matching entities are hard coded in the respective database queries. This made sense at the time, as the three used types `IS`, `HAS_VERSION`, and `ENCODED_BY` were the only sensible choices. It might however be possible, that future SBML models make use of different types of biomodels.net qualifier, which would need to be reflected in these queries. Incorporating these types into the configuration of SBML4j can help make this future proof.

The provenance tracking mechanics in SBML4j allow the tracking of the origin of a network through all steps of the network manipulation process. The available provenance report gives all necessary details to recreate a network from the sources. However, it cannot be directly used in a single request to reproduce the results. A good future extension might therefore be a dedicated endpoint for uploading the provenance report alongside the needed input data (i.e., SBML source files, potential CSV files for annotation), which automatically applies all described steps with the given parameters to recreate the network. Furthermore, the inclusion of external processing steps into the provenance tracking is limited to the manual addition of results and provenance information about the external tool. Possible improvements here might include the integration of automated processing steps with REST calls to these external tools.

Giving access to network analysis methods through a standardized REST interface with accompanying Python client libraries enables the integration of biological-network knowledge in existing bioinformatics and medical informatics pipelines and workflows without the need for hand-crafted database queries. By providing a locally executable database application, sensitive data can be analyzed in the context of biological networks without the need for data transfers to external servers or cloud infrastructure. With SBML4j biological-network creation, manipulation and extraction is reproducible and the standardized output format GraphML

makes it easy to share generated networks. The accompanying provenance report enables the reliable reproduction of the networks from the same input sources. By using SBML input files to initially construct the network database, no hand-crafted input scripts are required and setup can be scripted in Python. This also adds to the reproducibility of any results produced with SBML4j, as SBML models follow a strict specification, can be kept up to date using biomodels.net qualifier information and can be easily shared among research projects.

Chapter 4

Application: Network provenance in research pipelines

4.1 Introduction

Like many other disciplines, medicine and related fields are in danger of falling victim to the replication crisis, often also termed reproducibility crisis[132]. In 2016, Baker et al.[133] published the results of their survey concerning the state of reproducible science. They found that "[m]ore than 70% of researchers tried and failed to reproduce another scientist's experiments"[133]. Conversely, 66% of the participants in the survey stated that their lab had procedures in place for reproducibility, where the most frequently given answer was that a lab member is instructed to redo the experiment in question. Allison et al.[134] paint a similar picture in their report on reproducibility for their respective fields, obesity, nutrition, and energetics. They found a lot of mistakes in published articles and encountered many hurdles when they tried to contact the authors or the journals about these issues. One particularly striking point they make is that many authors refused to provide raw data for their analysis, which hinders other researchers from attempting to reproduce the results[134].

4.2 Background

In cancer research only 11% of scientific findings regarding new approaches for targeting tumors could be confirmed in replication studies, according to a survey performed by Begley and Ellis[135]. Similar inquiries in systems-biology modeling reveals that 37% of 455 investigated mathematical models could not be reproduced, even with empirical correction or with the support of the original authors[136]. Even more striking are the numbers stated by Kuenzi et al. in their groundbreaking analysis "A census of pathway maps in cancer systems biology"[80], where they assessed to what degree publications of systems-biology approaches, which map

and model cancer pathways, capture and extend current knowledge. They found that for pathway maps used in cancer research, 96% of the publications they surveyed did not provide enough information on their biological networks for automated processing. Many studies only provided their networks in publication figures or in a way that omits vital information, for instance details about which reported entities interact with each other[80].

Reproducibility of research is significantly impaired when scientists do not disclose the full information needed to recreate an experiment or investigate the correctness of published results. Timothy M. Errington, a lead investigator of the "Reproducibility project: Cancer biology"[137] stated: "Perhaps the clearest finding from the project is that many papers include too few details about their methods."[138]. The project draws a "muddy" picture of the state of reproducible science. This is especially true when publications show or use biological networks whose origins are unknown and which cannot be accessed programmatically or at least be recreated from the information provided[80].

In the clinical context, where therapeutic decisions are more and more frequently based on recent research results and clinical studies, the process of decision making has to be documented in full. Keeping track of the steps taken to create biological network depictions that are used in MTB meetings is challenging because of the wealth of online resources and the manual processing steps applied to a chosen base network. In addition, such a network is often only the starting point of an investigation into cellular processes that can be exploited for patient treatment or therapeutic intervention. In these cases, the representation of the network needs to encompass all necessary information for conducting further analysis. This so-called `provenance` of a network needs to be recorded in such a way that it is accessible at any point in time.

We use public datasets from `The Cancer Genome Atlas` (TCGA)[139] to demonstrate how SBML4j can act as data and metadata storage at the center of existing research efforts and clinical decision processes. The TCGA project collected molecular data from 20,000 primary cancer and corresponding control groups. Most of the data has access restrictions for certified research consortia, while some datasets were put in the public domain to be used by anybody. We downloaded a public dataset and preprocessed it using Python scripts that fetch them from the TCGA web API[139].

Then, they are used as scores for the software DeRegNet[105], which calculates maximally deregulated subnetworks in a provided source network. This source network is defined to be a directed graph $G = (V, E)$ with V denoting the nodes, or vertices, of the graph, and $E \subset V \times V$ denoting the edges of the graph that connect two vertices. Each edge is represented by the two vertices it connects as defined in Eq. 4.1. The direction of the edge is given by the ordering of

the nodes, where the first node is the source and the second node in the notation denotes the target of the edge.

$$(u, v) \in E \iff \exists \text{ edge } e \text{ that connects node } u \in V \text{ and node } v \in V \tag{4.1}$$

The result network was uploaded to SBML4j and connected to the provided base network. Then, we added the provenance information regarding the settings and parameters of DeRegNet to that newly created network.

Both the publication effort as well as the clinical decision system can then extract all relevant data and metadata including the provenance of input data and network data from a single source, namely the SBML4j service. This enables the peer review process to retrace the experimental steps and evaluate the submitted work. At the same time, clinicians and medical experts can verify the source and provenance of the provided candidate genes and base their decisions for therapeutic interventions on solid evidence.

4.3 Materials and Methods

The research setting that has been set up for this project can be seen in Fig. 4.1. It consists of the independent graph and data pipelines as well as the two standalone applications SBML4j and DeRegNet[105]. The flow of data and metadata is shown by the green and red arrows, respectively. The graph pipeline collects pathway maps from the KEGG database, translates them to the SBML format and uploads these files to SBML4j to create a network resource. Provenance information about the source of the network resource are stored alongside the network. This network resource is preprocessed for the use with DeRegNet by filtering incompatible elements from the graph. The data pipeline downloads biological data from the TCGA web service, preprocesses the files to be used as scoring input to DeRegNet and annotates the network resource with these data and available metadata. The network and the score data are provided to DeRegNet, which calculates the maximally deregulated networks. We then uploaded these results to SBML4j and annotated them with relevant provenance information about the DeRegNet application. A clinical process like the MTB preparation phase or a publication effort in a research lab can access the network resource as well as the provenance report directly from SBML4j.

4.3.1 Graph Pipeline

The graph pipeline (see Fig. 4.1) consists of multiple Python programs and makes use of KEGGtranslator[126] for translating between the KGML[49] and the SBML format. Additionally, a Python program is dedicated to preparing a network instance in SBML4j for the use with DeRegnet. The sequence of requests and responses between these parts of the pipeline can be seen in Fig. 4.2.

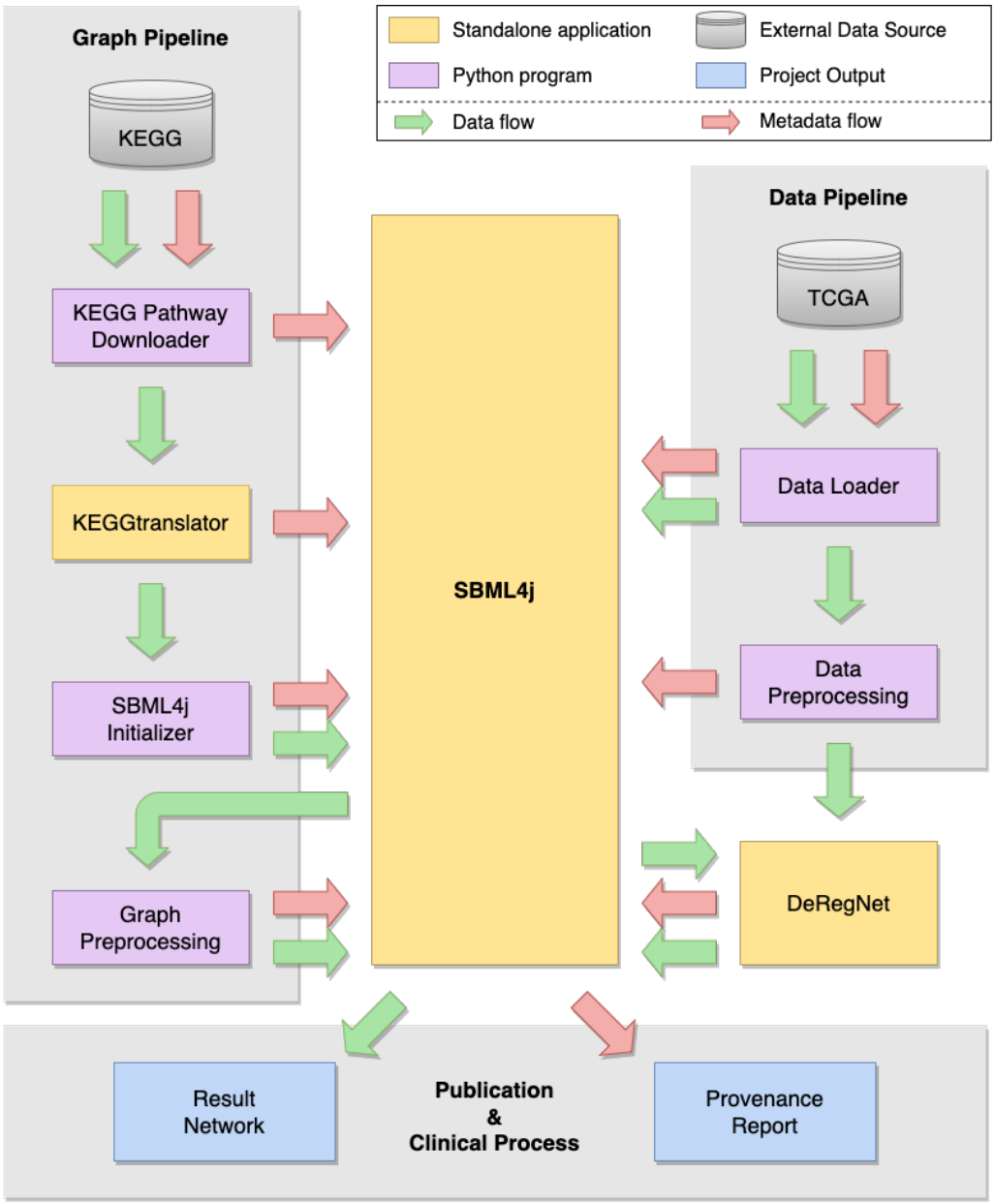


Figure 4.1: Research pipeline for generating deregulated subnetworks from TCGA data with SBML4j provenance tracking. The Graph Pipeline downloads pathway maps from KEGG, translates and preprocesses them for SBML4j and combines them into a single network resource. This network is further preprocessed for DeRegNet and made available via the GraphML export. In the Data Pipeline biological data is collected from the TCGA database, preprocessed and provided to DeRegNet. The result of DeRegNet is uploaded to SBML4j and appropriate metadata added to the created network resource. Publication and clinical processes can access the result networks and provenance reports directly from SBML4j.

KEGG Pathway Downloader

The KEGG pathway downloader (KPWD) is a Python program dedicated to searching and downloading KEGG pathway maps in the KGML format from the KEGG API[140], a RESTful application interface to the KEGG data. To download a specific pathway map from the service, a GET request to the URL `http://rest.kegg.jp/get/<pathway-identifier>/kgml` (see Fig. 4.2A) can be sent, where `<pathway-identifier>` has to be replaced with the exact identifier of a pathway map, including the organism code (e.g., hsa05200). KPWD allows to specify a set of such pathway identifiers or search terms in a configuration file. Each pathway in the provided set will be downloaded and stored on the local harddrive. If a search term is provided, an additional call to the `/find` endpoint is sent, which returns a list of pathway entries that contain the search term. Each of those pathways is then downloaded to the local harddrive. In addition, KPWD creates a metadata file for each downloaded pathway file that contains the reported creation date from KEGG, which is extracted from the KGML file as well as the date of the download. These files are provided to a later stage of the pipeline (see Fig. 4.2B), where their content is added to the corresponding objects in SBML4j as provenance information to document the file origin.

Table 4.1: Pathway ids used for the creation of the base network and downloaded by KPWD.

identifier	definition
05210	Colorectal cancer
05212	Pancreatic cancer
05225	Hepatocellular carcinoma
05226	Gastric cancer
05214	Gslioma
05216	Thyroid cancer
05221	Acute myeloid leukemia
05220	Chronic myeloid leukemia
05217	Basal cell carcinoma
05218	Melanoma
05211	Renal cell carcinoma
05219	Bladder cancer
05215	Prostate cancer
05213	Endometrial cancer
05224	Breast cancer
05222	Small cell lung cancer
05223	Non-small cell lung cancer

The source code for KPWD is available in the GitHub repository `https://github.com/thortiede/KPWD`, which gives instructions on how to configure and build the docker image

```
1  docker run -it --volume=${PWD}/data:/data \
2                 --volume=${PWD}/config:/config \
3                 kpwd examples/deregnet.ids
```

Listing 4.1: Command to run the docker container KPWD. Runs the docker container of name kpwd and passes the given file in the examples folder to the Python script. The data folder will contain the downloaded pathway maps, the config folder contains the config files and the example file listing the pathway ids.

that is being used here. The configuration file that is used with KPWD can be found in the appendix as Listing E.6.

For this application, we configured KPWD to download all the pathways under section `6.2 Cancer: specific types` of the KEGG pathway database[141] (see Tab. 4.1). The run command for KPWD is shown in Listing 4.1, while the provided file that contains the pathway identifiers can be found on GitHub at `https://github.com/thortiede/KPWD/config/examples/deregnet.ids`.

KEGGtranslator

Before the KEGG pathway maps can be uploaded to SBML4j they need to be translated into the SBML format. KEGGtranslator[126] is a command line tool to translate the KGML files from KEGG into various other formats, including SBML. Here, we have used version 2.5 of KEGGtranslator. The previously downloaded KGML files are provided to KEGGtranslator and the translation process (Fig. 4.2C) is started. The command-line arguments to the KEGGtranslator application can be found in the appendix in Listing E.3. A corresponding SBML file is created for each KGML file in the input set.

SBML4j Initializer

The SBML4j Initializer is a Python program to populate an SBML4j database with a given set of SBML models, combine those models in a `PathwayCollection` and create one or more network mappings from this collection. For each SBML model a POST request is made to SBML4j (see Fig. 4.2D) which creates a Pathway object in its database. The request includes version information for the KEGG database from which this model was originally downloaded. Additionally, metadata information about KEGGtranslator is collected (see Fig. 4.2E) and added alongside the KGML Provenance information (see Fig. 4.2B) to each Pathway to ensure the complete provenance of the data (see Fig. 4.2F). The returned UUIDs for the pathways are collected and used in a POST request to create a `PathwayCollection` (see Fig. 4.2G). This results in a new pathway object with a unique UUID, which gets returned to the caller. This UUID is then used in another POST request (Fig. 4.2H) to create a signalling mapping from this collection, whose UUID is then provided to the next step.

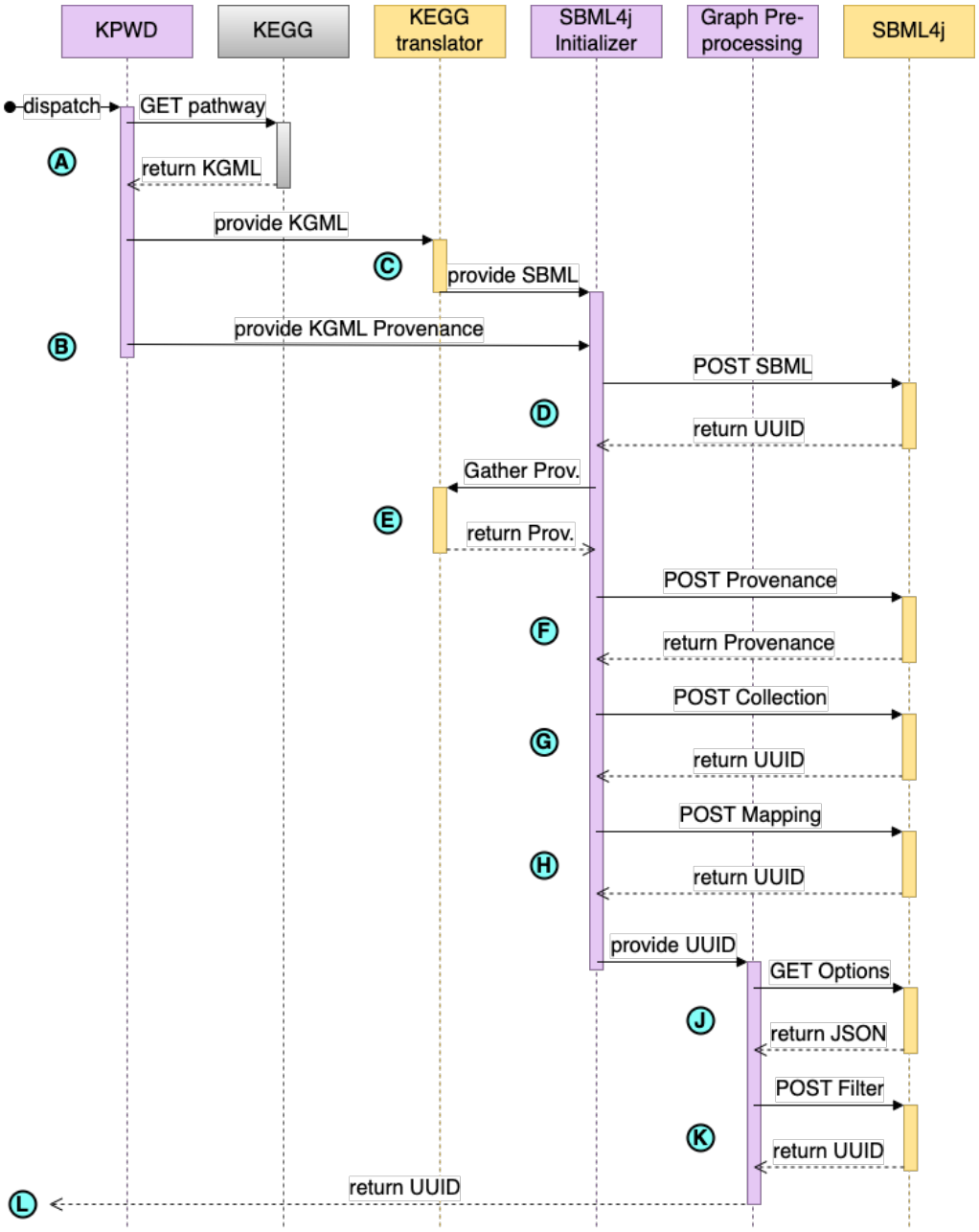


Figure 4.2: Sequence diagram for the graph pipeline. A dispatch signal triggers the download of KEGG pathway maps (A) which are translated to SBML by KEGGtranslator (B). Provenance metadata from KEGG is provided to the initializer script (C) and combined with metadata from KEGGtranslator (D). The SBML files are uploaded to SBML4j (E), enriched with the collected metadata (F), combined to a PathwayCollection (G) and a network mapping is created from it (H). With the UUID of that network, available filter options can be requested from SBML4j and the network subsequently filtered (J). The UUID of the finished network is reported to the original caller (K).

The source code of the SBML4j Initializer is available in the GitHub repository `https://github.com/thortiede/S4IWP`, while the configuration file for this project can be found in the appendix as Listing E.7.

Graph Preprocessing

The graph that is used for the calculation of subnetworks is assumed to have no self-loops, so in the notation of Eq. 4.1 the edgeset E of the graph G has to satisfy the condition in Eq. 4.2[105].

$$(v, v) \notin E \forall v \in V \tag{4.2}$$

Any source network that is provided to DeRegNet must adhere to this restriction. As networks in SBML4j can contain self-loops, a preprocessing step that eliminates them has to be performed. This is achieved by filtering out those relationships in a network for which source and target nodes are equal. First, we request the network options of a base network from SBML4j via the `networks/<UUID>/options` endpoint or the Python method `network.getOptions()` (see Fig. 4.2J). From the resulting JSON response, the `filter` object contains a list element named "relationSymbols" that contains all relation symbols in the network. We scanned these symbols for relationships that have the same start and end node (see Listing 4.2). To accomplish this, the symbol is split at the "->" substring (line 2). The last element of the resulting array contains the target node symbol (line 3). The substring of the symbol up to the "->" string contains the source node symbol and the relation name (lines 4–5). Since the target node symbol can be a substring of the source node symbol, a substring match for the target node symbol on the `source_and_type` string does not suffice for self-loop detection. To identify the full name of the source node of the relation, the `source_and_type` string is split by the "-" character, which is used to separate the source node symbol from the relation name (line 6). The last element in the resulting array is the name of the relation, since the relation name does not contain dashes (line 6). The substring of `source_and_type` string that spans from its start to the position that is one character before the beginning of the relation name is then identified as the full source node symbol (line 7). If the source and target strings are identical, the relation symbol denotes a self loop and the return value of `True` is returned, `False` otherwise. We removed all relation symbols that are detected this way from the list of relation symbols and used this updated list in the filter-options payload to a POST `filter` call (see Fig. 4.2K) of the base network. The response contains the UUID of the final network that will be used with DeRegNet and is returned to the calling agent of the graph pipeline (see Fig. 4.2L).

The source code for the graph preprocessing can be found in the Python subfolder of the DeRegNet Pipeline GitHub repository[142] in the file `DeRegNet_Graph_Preprocessing.py`.

```
1       def is_self_loop(relationSymbol):
2           parts = relationSymbol.split('->')
3           target = parts[len(parts)-1]
4           end = len(relationSymbol)-len(target)-2
5           source_and_type = relationSymbol[0: end]
6           relation_name = source_and_type.split("-")[-1]
7           source = source_and_type[:len(source_and_type) - len(relation_name) - 1]
8           if source == target:
9               return True
10          else:
11              return False
```

Listing 4.2: Python method to identify self-loops from a relation symbol obtained from the filter options of an SBML4j network.

4.3.2 Data Pipeline

The data pipeline (see Fig. 4.1) starts with the Data Loader Python program that fetches data and metadata from the TCGA API and stores both alongside the base network in SBML4j. The data file is then passed on to the preprocessing script that extracts the relevant data points for use with DeRegNet and creates a score file. The metadata about the preprocessing script is added to the Provenance data of the base network and the processed score file is returned to the calling agent. The sequence of requests and responses between these parts of the pipeline can be seen in Fig. 4.3 and will be described in more detail in the following sections.

TCGA Datasets

In this project, publicly available research data from TCGA is used in a research pipeline, specifically differential gene expression (DGE) data from the Clinical Proteomic Tumor Analysis Consortium (CPTAC)[143,144]. The experimental strategy used for data generation was single-cell RNA Sequencing and post-processing using the Seurat R package[145]. The datafiles are tabular separated values (tsv) files, where the columns `gene_names` and `avg_log2FC` were of interest for this work. A complete documentation of the file format can be found at the official GitHub repository[146]. This dataset has been analysed by the consortium using the Seurat[146] R toolkit and the differentially expressed genes were reported as logarithmic (base 2) fold changes (Log2FC). The Log2FC value provides a measure for the differentially expressed genes in tumor tissue compared to healthy tissue from control groups. Details on the analysis and the specific patient cohort are outside of the scope of this work and the reader is referred to the consortium[143,144] for an in-depth analysis and report.

The `gene_names` column contains the HGNC symbol of the coding gene for the measured mRNA and is used to match the data to the appropriate node in the SBML4j network upon uploading the file for annotation. The `avg_log2FC` column holds the averaged Log_2 Fold Change value as calculated from the data by Seurat. It provides the fold change of gene expression on a logarithmic scale. We used this value as score for the DeRegNet algorithm

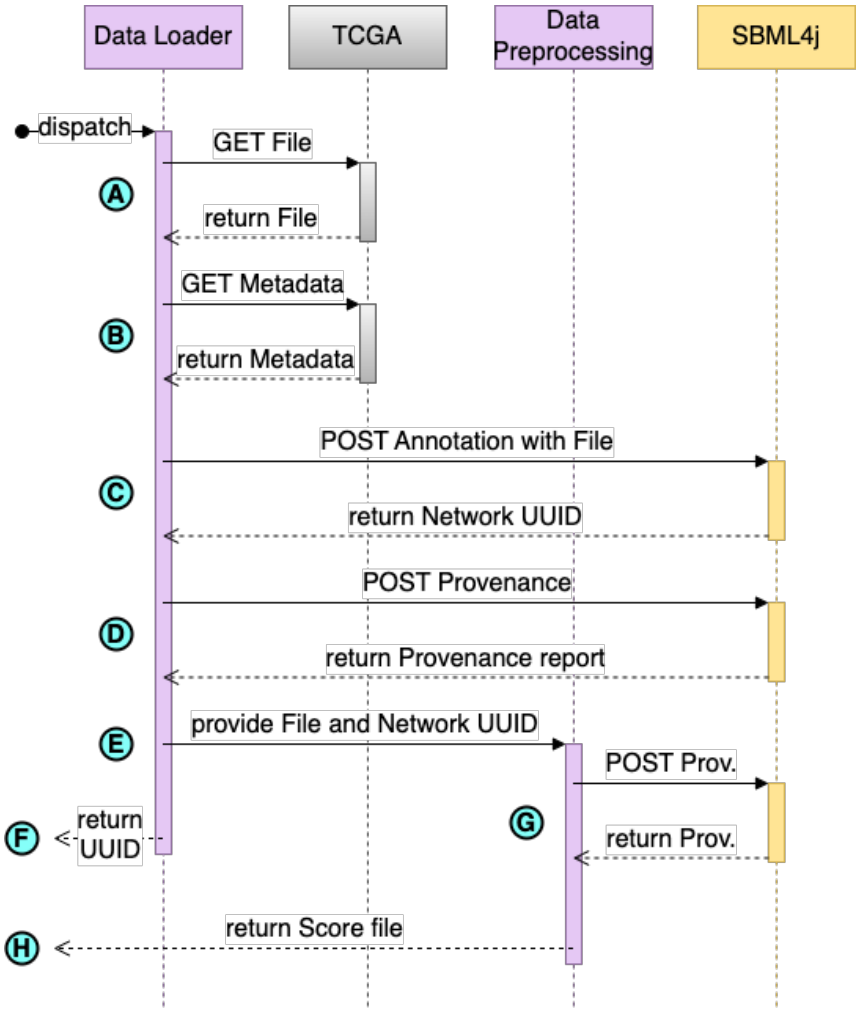


Figure 4.3: Sequence diagram for the data pipeline. (A) A dispatch signal to the Data Loader triggers the download of a TCGA data file. (B) Metadata for the file and the TCGA project is fetched from the TCGA API. (C) The file data is annotated on the prepared network in SBML4j. (D) Collected Metadata from TCGA is added as Provenance information to the network. (E) The downloaded file is provided to the preprocessing program via the filesystem. (F) After preprocessing and creating the score file for DeRegNet, metadata about this step is added to the network in SBML4j. (G) Score file is returned to the initial caller.

for determining the deregulation of genes. To apply these scores to nodes in the network in DeRegNet, we matched the entry in the column `gene_names` to the `name` attribute on the nodes of the base network. For this application, we used the file with UUID 69a92ea6-7d1e-4cac-a65f-b8afa2380e97 from this dataset. It can be accessed via the download URL `https://api.gdc.cancer.gov/data/69a92ea6-7d1e-4cac-a65f-b8afa2380e97`.

Data Loader

The Data Loader is a Python script that uses the package `urllib3`[116] to make requests to the TCGA API. For downloading individual files from this API, the TCGA UUID has to be known and provided to the endpoint `https://api.gdc.cancer.gov/data/<UUID>` (see Fig. 4.3A). The file is stored on the local harddrive, where the follow-up preprocessing script can access it (see Fig. 4.3E).

In addition to the raw file, we fetched some metadata about the TCGA project and the file itself from the TCGA API. Metadata about the file were extracted from the header fields of the file download response. The filename is extracted from the `Content-Disposition` header, while the `Date` of the download, the `Content-Length` and `Content-MD5` headers are stored in their entirety. Furthermore, the download URL and the UUID are stored alongside this file metadata. To get the project metadata, we issue a GET request to the URL `https://api.gdc.cancer.gov/projects/CPTAC-3` (see Fig. 4.3B). The request includes the parameter `expand`, which can be used to request details about sections of the project data. We provided `expand` keywords for the `summary`, the `experimental strategies` and the `data categories` of the project. The complete response of this request is stored in a Python dictionary alongside the file metadata.

Then the raw data file is uploaded to SBML4j in a POST request to the endpoint `/network/<UUID>/csv` (see Fig. 4.3C), which annotates all matching nodes with the provided data columns in the file, where `<UUID>` is replaced by the UUID of the base network that was created by the graph pipeline. We use the column `gene_names` to match the rows of the CSV file to the graph nodes. This creates a new network instance in SBML4j in order to keep track of the provenance steps that were performed on the data. The UUID of this new network is returned at the end of the script to the calling module, which will provide the respective network to DeRegNet for subnetwork calculation. Lastly, we upload the metadata dictionary in a POST request to the SBML4j API endpoint `/prov/<UUID>` (see Fig. 4.3D). This adds all entries in the dictionary to the Provenance Report of that network under the provenance item name that corresponds to the UUID of the TCGA file. The `<UUID>` in this request corresponds to the annotated base network that will be provided to DeRegNet and which is returned to the calling agent upon completion of the pipeline (see Fig. 4.3F).

The data loader is part of the pipeline project on GitHub[142] and can be found in the source file `TCGA_Data_Loader.py` in the Python subfolder.

Data Preprocessing

A score file for DeRegNet needs to be a CSV file with two columns, where the first column is named `id` and the second column is named `score`. The `id` column is used to match the `score` value to the nodes in the graph within DeRegNet. Here, the `gene_names` column matches with

the `name` attribute on the network graph. The values in the `gene_names` and the `avg_log2FC` columns are written to the output score file.

Details about this preprocessing, including the name and version of the script, the names of the matching colum and the data column in the original file as well as the name of the original file are provided as provenance information to the base network for DeRegNet in SBML4j (see Fig. 4.3G). Finally, the score file is stored on the harddrive and made available to the dispatching agent to be used in the call of DeRegNet (see Fig. 4.3H).

The source code for the data preprocessing can be found in the Python subfolder of the DeRegNet Pipeline GitHub repository[142] under the name `TCGA_Data_Preprocessing.py`.

4.3.3 DeRegNet

The software DeRegNet[105] calculates deregulated subnetworks from a base network and provided `score` values that get interpreted as a deregulation measure. DeRegNet performs maximum likelihood estimation on a defined probalistic model that takes network connectedness and topolgy into account to identify subgraphs in the given network. This optimization problem is solved with fractional integer programming and uses the software Gurobi[147] for this process. We used the latest version of the Gurobi solver, which at the time of writing is version *9.5.2*. It supports containerized environments using individual license tokens for each container. For details on the algorithm and the various runtime options the reader is referred to the publication[105] and the offical GitHub repository[148].

Here, we used the latest version "grb9.5.2" of DeRegNet, which is available from the first author's DockerHub repository[149]. For this application demonstration, all options except the graph input were left at their default setting. The command that was used to run the DeRegNet docker container can be seen in Listing 4.3. The input network to DeRegNet is a directed graph without self-loops in the GraphML format, which is what the Graph Pipeline produced earlier. The graph file to use is provided to DeRegNet on the command-line with the *--graph* option (see Listing 4.3line4). Likewise, the score file that was created by the Data Pipeline is passed to DeRegNet with the *--scores* option (see Listing 4.3line5). In addition, the character that separates the columns in the scores file is provided to the *--sep* option and the name of the node attribute that is to be used for matching the score provided to the graph is given in the *--graph-id-attr* option (see Listing 4.3 lines 5&6).

The local directory `io/` is passed to the container environment for facilitating data input and output from and to the host machine (see Listing 4.3line1). The input folder must contain the result of the Graph Pipeline in the file `network.graphml` and the result of the Data Pipeline in the file `scores.csv`. The result of DeRegNet is in our case a single graphml file that contains the optimal solution produced by the algorithm and will then be placed in the

```
1  docker run -it --rm -v $(pwd}/io:/io \
2                      -v $/path/to/gurobi/license:/gurobi/lic \
3  sebwink/deregnet:grb9.5.2 avgdrgnt.py \
4    --graph /io/in/network.graphml \
5    --scores /io/in/score.csv \
6    --sep ';' \
7    --graph-id-attr name \
8    --output-path /io/out
```

Listing 4.3: Command-line execution of the DeRegNet docker container with the graph and scores produced in this work.

`out/` subdirectory. The license file for the Gurobi solver was made available to the container environment in the folder `/gurobi/lic` (see Listing 4.3 line 2).

4.3.4 SBML4j

In this work, SBML4j version `1.2.2` has been used. It is executed through the docker-compose setup found on GitHub[113] and uses version `v1.0.0` of that repository. The setup configures and starts SBML4j in the required version as a docker container. In addition, the official Neo4j docker image[150] version `4.1.6` is started and configured according to the configuration file in the repository.

After initialization as described in the repository, no further action is required. We use an empty database and start the compose setup with *docker-compose up –attach-dependencies sbml4j*. This ensures that the console output of all containers is displayed on the terminal.

4.3.5 Pipeline Orchestration

The modules of this research pipeline need to be executed in the correct order. With an instance of SBML4j running with an empty database, we execute KPWD, KEGGtranslator and the SBML4j Initializer as described in this work.

As these three steps are the most time consuming, we used the `./sbml4j.sh` script from the sbml4j-compose repository to store the database state in a backup for later use. We provided the script with the option `-b deregnet_base_mapping_prov`, which creates two `.dump` files in the subfolder `db_backups`. This backup can be restored at a later time with the command `./sbml4j.sh -r deregnet_base_mapping_prov` in the sbml4j-compose main directory.

The rest of the Research Pipeline is executed using a set of shell scripts that can be found in the GitHub repository[142]. In this work we used version `v1.0.0` of this repository. The main script `run_pipeline.sh` runs the Python modules `Data Loader`, `Data Preprocessing` and `Graph Preprocessing` sequentially. Then it calls the script that executes the previously configured DeRegNet docker container. After DeRegNet has finished, two additional Python scripts are executed that upload the result network and annotate it with metadata

information about the tool. After the pipeline finishes successfully, the `result` folder contains the three result files `optimal.graphml`, `optimal_uuid`, and `optimal_prov.json`. The file `optimal.graphml` contains the solution network that was created by DeRegNet. The file `optimal_uuid` contains the UUID of that same solution network in SBML4j and `optimal_prov.json` contains the provenance report of the solution network in JSON format.

4.4 Results

The three result files `optimal.graphml`, `optimal_uuid`, and `optimal_prov.json` that are produced by the pipeline in this work contain the main results of the research setting presented here. The optimal solution network that has been calculated by the DeRegNet algorithm is available as GraphML output in the file `optimal.graphml`. It can also be retrieved from the SBML4j service using the UUID that can be found in the file `optimal_uuid`. A detailed provenance report that traces the origins of the result network all the way back to the original input files from KEGG and TCGA is available in the file `optimal_prov.json`. It can also be downloaded from SBML4j using the same UUID as before in a GET request to the `/prov/<UUID>` endpoint of the running instance.

4.4.1 Result Network

The result network from the DeRegNet calculation can be seen in Fig. 4.4. It has been visualized using BioGraphVisart[125].

The color of the nodes show the `deregnet_score`, which corresponds to the reported avg_Log_2FC fold change value in the data from TCGA. It shows deregulated cascades starting at the gene with the HGNC symbol `FGFR2` and ending in various upregulated target genes. The biological interpretation of this network is outside the scope of this work and serves purely a demonstration purpose.

4.4.2 Provenance Report

The Provenance report for this network contains all added metadata throughout the pipelines and gives a detailed listing of all source files, transformations and analysis steps that have been performed to reach this network. A visual representation of the contents of the report is shown in Fig. 4.5, while the report itself is in JSON format, with each layer corresponding to subelements of the previous element, thus giving the typical nested structure of the format. The JSON formatted Provenance report can be found in the appendix as Listing E.8. For brevity, only one `Pathway` section with nested `SBMLfile` section is included in the report (see Listing E.8 lines 437–614) and the entries for `nodeSymbols` and `relationSymbols` are trimmed to two

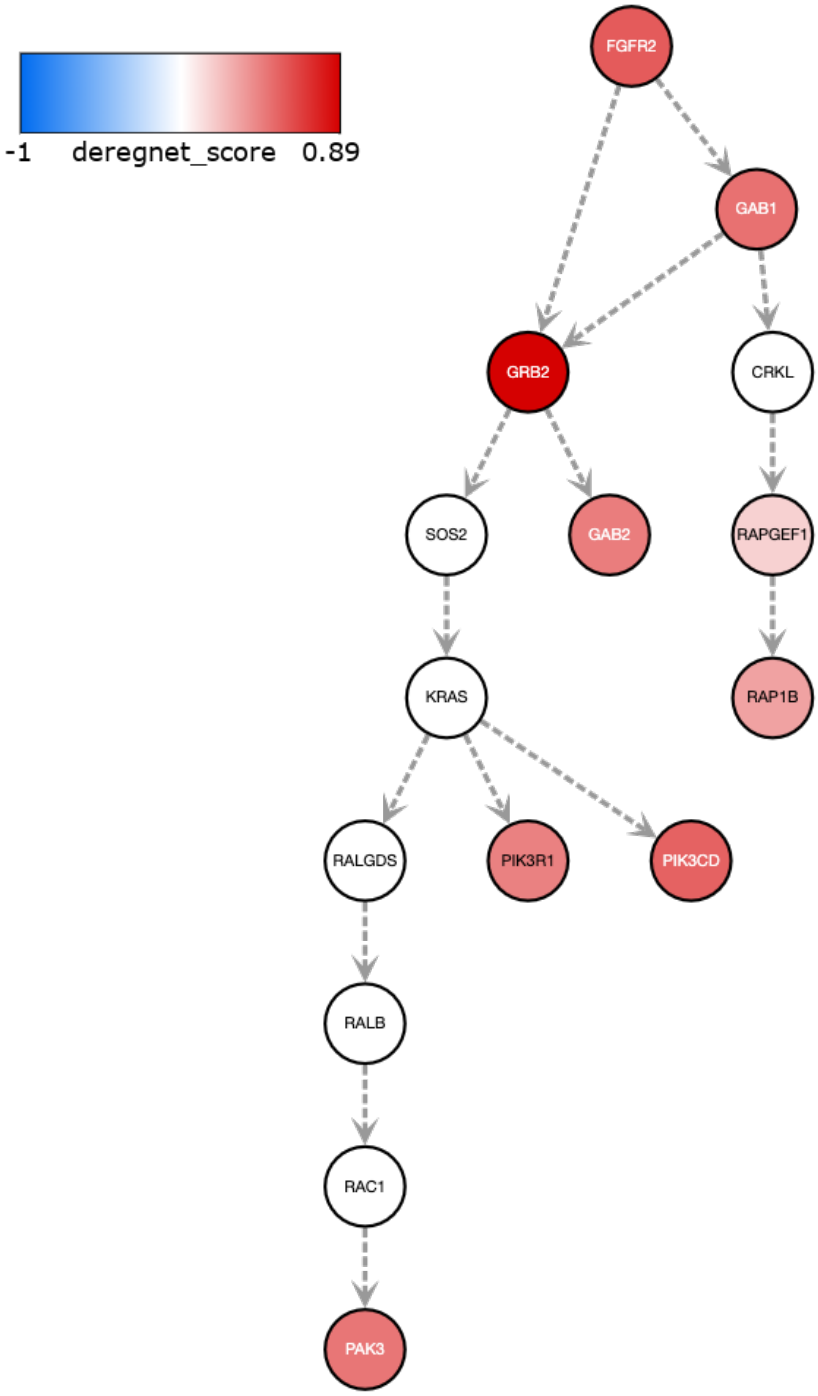


Figure 4.4: Result network from DeRegNet. Color of the nodes shows the deregulation score, arrows denote stimulation signals.

entries each (see Listing E.8 lines 120–129). The untrimmed provenance report can be found in the result folder of the Pipeline GitHub repository[142].

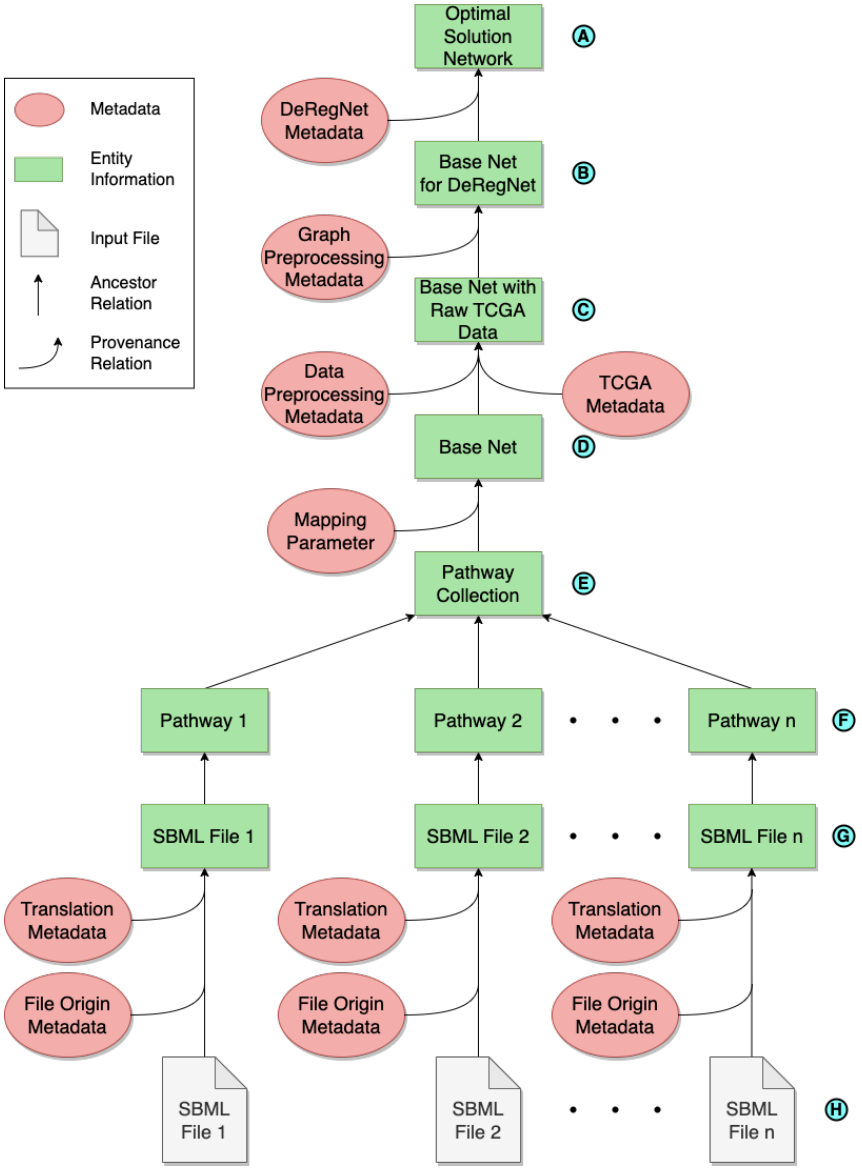


Figure 4.5: Visualization of the contents of the Provenance report. Green Boxes represent sections of the report that describe entities in the database. The vertical arrows indicate a nested relation of the elements with respect to the upper element being derived from the lower element. Red Circles indicate Metadata that are added to the elements via provenance upload, with the bend arrows indicating at which element they are attached. SBML input files are the initial entry and include provenance information about the original datasource.

The top most layer (see Fig. 4.5A) represents the network entity that has been created by uploading the DeRegNet optimal solution results file to SBML4j. It is accompanied by metadata information about the DeRegNet application and its configuration, if applicable. As DeRegNet has been run with default options, no additional metadata apart from the application info

is provided (see Listing E.8 lines 1–23). The next layer describes the base network that was created by the last step in the Graph Pipeline (see Fig. 4.5B). This network is the result of a standard filtering operation for which the request parameters as well as the complete body of the request are included in the report (see Listing E.8 lines 87–132).

The output of the data pipeline serves as parent network for this filtering operation, which is generated by the Data Preprocessing Python program (see Fig. 4.5C). It is annotated with provenance data concerning the preprocessing step (see Listing E.8 lines 184–191) as well as metadata about the file containing the annotation data and provenance information about the TCGA source (see Listing E.8 lines 135–183).

The network that served as basis for the Data Preprocessing Pipeline is depicted by the `Base Net` element (see Fig. 4.5D). It is the result of a mapping processing from a Pathway entity in SBML4j and its provenance information includes details about the request that triggered the mapping creation (see Listing E.8 lines 251–308).

The pathway from which the mapping is derived is a collection of smaller pathways (see Fig. 4.5E), whose creation is tracked by details about the request to the SBMl4j API for building this collection pathway (see Listing E.8 lines 311–375). It includes a list of all `UUID` of the pathways that are collected, and is internally derived from an entity of type `PathwayCollection` (see Listing E.8 lines 376–436).

For each of the pathways that were collected a list entry is present in the provenance report (see Fig. 4.5F). For illustration the entry for one pathway is included in the provenance report (see Listing E.8 lines 439–466), while the others are left out for brevity. Each pathway is generated by a POST request to the `/sbml` endpoint of SBML4j, whose provenance details are part of the report (see Listing E.8 lines 471–500).

The SBML file (see Fig. 4.5H) that was uploaded in this request is represented in the provenance report by a `File` type (see Fig. 4.5G), for which default metadata like the filename and MD5 sum are collected in SBML4j by handling the POST request itself (see Listing E.8 lines 529–534). Additional provenance information about the file's origin (see Listing E.8 lines 504–510) as well as the translation operation using KEGGtranslator (see Listing E.8 lines 512–525) is added to the report entry.

4.5 Discussion and Conclusion

The application DeRegNet calculates maximally deregulated subnetworks from a given base network and provided deregulation scores on the nodes of this network. DeRegNet provides a precalculated base network that was derived from the pathways of the KEGG pathways database. This provided base network is created with hand-crafted procedures and scripts, which are available at the GitHub repository of DeRegNet[148]. When DeRegNet is used by other researchers who want to calculate deregulated networks on the pathways of KEGG, they

can make use of this provided base network. However, when reporting their results in their publications, they can only refer back to this provided base network and the scripts that were used to create them as the source for their calculations. Tracing back the results to the exact version of the KEGG pathways database is cumbersome and makes evidence-based decisions rely on a single source of truth, in this case the pre-calculated base network that DeRegNet provides for KEGG.

In comparison, the base network provided by SBML4j comes with a complete provenance report that details the source files, their origin and KEGG version, the way the network has been created from these source files with SBML4j as well as the transformation steps that have been performed during this process. This allows for a more reliable recreation of the base network, which results in an increased reproducibility.

In the same manner, information about the biological data, which is used to create the deregulation scores that are used by the algorithm can be stored in SBML4j alongside the network data. CSV data from external sources can be added to any network in the service and provenance information about the source of the data files ensures reproducibility.

With the ability to upload not only SBML files but also GraphML-based networks, SBML4j can track the provenance of biological networks even when networks are consumed and created by external applications, as has been shown here with DeRegNet. The result network from DeRegNet can be provided to SBML4j and linked to the base network to allow a complete provenance history of the deregulated network.

With the support for the widely used GraphML format, many network-oriented research applications can already make use of this feature of SBMl4j. The support of more graph formats, like the `LEMON`[151] `Graph Format`[152] and the `Simple Interchange Format`[153] could open this up to even more applications in the bioinformatics and medical informatics disciplines and could be a viable direction for the development of SBML4j.

There is still manual work required to fully annotate the result network and all previous steps in the process. The scientist is responsible to add appropriate metadata as provenance information to individual steps. However, this manual work can be integrated in the workflow during creation of the base network and the calculation of the results. This enables the automation of these steps and helps to clearly define the necessary inputs for the achieved result. Once a deregulated network has been calculated and stored in SBML4j, its contents and the accompanying provenance report are available through the RESTful interface. This enhances the reproducibility of the results and makes scientific findings with DeRegNet more reliable for publications and clinical investigations. With the versatile graph interchange format GraphML, a network that is downloaded from SBML4j can be programmatically analysed, visualized and used in follow-up work.

The provenance report of such a network can be made available in publications, which enables other researchers to trace back the steps necessary for recreating the work. This report

can also serve as evidence for clinical decision systems, where therapeutic advice is based on biological network knowledge that has been provided by researchers using SBML4j.

With these mechanisms, SBML4j provides a central provenance repository for biological networks. It can provide networks and their detailed provenance to applications and publications. At the same time, it serves as central storage and provenance repository for networks that are created by any application and uploaded to SBML4j. This makes SBML4j and the accompanying pipeline tools presented here valuable software packages for any research lab that aims to improve their reproducibility.

Chapter 5

Application: Personalized Cancer and Network Explorer – PeCaX

The content of this chapter is an extended version of the article:

Figaschewski M., Sürün B., Tiede, T., Kohlbacher O.:
The Personalized Cancer Network Explorer (PeCaX) as a visual analytics tool to support molecular tumor boards[154]

5.1 Introduction

Cancer is typically caused by genomic alterations inducing unchecked cellular proliferation. In personalized oncology[155], molecular data (e.g., genomics) is used jointly with clinical data to stratify therapies and choose the therapy best-suited for a specific patient. Next Generation Sequencing (NGS) is widely used to find those genomic alterations, such as single-nucleotide variants (SNVs), copy number variations (CNVs), or gene fusions.

Based on this data, the typical analysis workflow is usually as follows:

1. The (cancer) genome of a patient is sequenced and the SNVs and CNVs are stored in a Variant Call Format (VCF) file.
2. Variants are annotated with their effect.
3. Usually only variants with a strong effect are considered.
4. The remaining variants are looked up in databases to identify driver genes and to find drugs associated with these potential targets.

5. If no drug can be found for this specific variant or it is not applicable for the patient, pathways containing the related gene are considered to find druggable targets up- or downstream of the actionable variant.

This process of revealing drug-gene information based on the variant annotations and incorporating pharmacogenomics information which is an indicator of the effect of the genes on a patient's drug response is called clinical annotation.

5.2 Background

Numerous tools exist for displaying and storing the information of a VCF file in the common tab separated values format (step 1) and to filter the variants for given annotations (step 3), e.g. VCF-Miner[156], BrowseVCF[157], VCF-Explorer[158]. But only few applications include the analysis of the SNVs and CNVs and the annotation of the variant effect (step 2), e.g. VCF-Server[159]. Perera-Bel et al. offer the additional option to find drugs targeting the variants (step 4) but their method does not perform variant effect prediction (step 2) and is limited to a specific data structure[160]. It also lacks a graphical user interface (GUI). OncoPDSS performs steps 1 to 4 but it is a web-server which can be a data security issue. Therefore, OncoPDSS does not store the input or results of the analysis[161]. These are displayed in unsearchable tables focusing on information about available pharmacotherapies which can be downloaded as TSV files. But it does not give information on the pathway context of a gene. So far, this information has to be collected manually.

We present PeCaX (Personalized Cancer Network Explorer), an integrated application for personalized oncology workflows. PeCaX performs clinical variant annotation by processing SNVs and CNVs and identifying clinically relevant variants and their targeting therapeutics using ClinVAP[162]. Networks containing the connections between the driver genes and the genes in their neighborhood as well as drugs targeting genes in this network are created through the novel SBML4j[114] and they are visualized with the use of BioGraphVisart[125], developed specifically for PeCaX. Our user-friendly, web-based graphical user interface does not only interactively display the report generated by ClinVAP and the networks with a few clicks, but adds web links to external gene and drug databases and gives the option to take notes which are stored in PeCaX along with the information presented in the tables and networks. This, for instance, allows the user to interactively work on and present the results in Molecular Tumor Board (MTB) meetings where oncologists and clinicians of different areas meet and discuss individual case files. The report can be downloaded as PDFs. In addition, the networks are available for download in publication-ready file formats (PNG, SVG and GraphML).

In contrast to many VCF-analysis and clinical decision support tools, PeCaX is a local application with a graphical user interface working on the user's local machine avoiding data privacy issues arising from the use of cloud-based services.

5.3 Implementation

Technical Overview

PeCaX is a service-oriented local application and has been built using the NuxtJS framework for its web-based front end and integrates several other local services developed by us via REST APIs (see Fig. 5.1).

It was developed in close interaction with persons responsible for MTB case management, scientific analysis, case preparation, and presentation at the University Hospital Tübingen to ensure a user-friendly user interface. It supports concurrent use and works on any modern web browser independent of the operating system. Its design as a web service allows access from browsers not running on the same machine as the service. Sensitive data is only processed on the machine PeCaX is installed on, not the machines that access it with the GUI. It is easy to deploy via pre-built docker containers and easily integrated using docker compose. The individual docker containers and the communication via REST APIs allow to update the services individually without the need to setup everything from the start.

5.3.1 Clinical Variant Annotation

The first major task that is performed by PeCaX is the annotation of clinical variants based on provided VCF files. These provide information on SNVs. Optionally, a TSV file can be provided that contains CNV information. The validity of files to be uploaded is checked by the filename extension (.vcf, .tsv). If an uploaded file contains semantic or syntactic errors, the analysis process is aborted and the user is notified that the input file is corrupt.

PeCaX integrates ClinVAP to create a case report by processing variants using functional and clinical annotations of the genomic aberrations observed in a patient. ClinVAP employs Ensemble Variant Effect Predictor (VEP) [163] to obtain functional effects of the observed variants and filters them based on the severity of the predictions.

It also performs clinical annotation which reveals the driver genes, actionable targets and enriches them with their known therapeutic associations using an integrated knowledge base from publicly available databases (e.g., COSMIC [164], CGI [165]). Moreover, it provides an option to filter the results based on the diagnosis type given as ICD10 code which was achieved by obtaining the gene-disease links from the background databases and mapping those diseases to their corresponding ICD10 codes. The mapping between the disease names from the databases and from ICD10 is done by matching their disease related features such as system, organ,

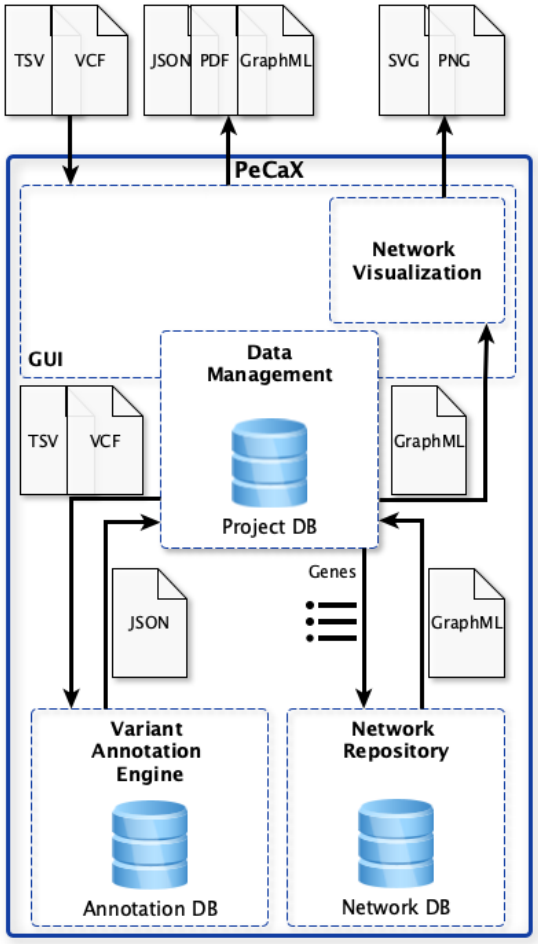


Figure 5.1: Architecture of PeCaX. PeCaX consists of five building blocks. The `GUI` is repsonsible for data input and out and presentation of the results to the user. The `Data Management` is responsible for keeping all relevant data available. The `Variant Annotation` processes the input files and generates the variant report. The `Network repository` creates and stores the neighborhood networks. The `Network Visualization` is responsible for providing interactive visualizations of the network within the GUI.

histology type. As soon as the annotation is finished the variant files are deleted and PeCaX receives the resulting report as JSON file with information structured into five categories: known driver genes, drugs targeting the variants, therapeutics targeting the affected genes, cancer drugs targeting the mutated genes, and drugs with known adverse effects.

5.3.2 Network Generation

Cancer is a complex and heterogeneous disease typically caused by genomic alterations. Even a single mutation can modulate the complex interaction network of genes to cause cancer

```
1  import pysbml4j
2  import os
3
4  client = pysbml4j.Sbml4j()
5  filePath = "/absolute/path/to/sbml/model/"
6
7  pathwayUUIDs = []
8  filelist = os.listdir(filePath)
9  for file in filelist:
10   fullfilename = os.path.join(filePath, file)
11   resp = client.uploadSBML([fullfilename], "hsa", "KEGG", "97.0")
12   pwuuids.append(resp.get(fullfilename).get("uuid"))
```

Listing 5.1: Upload translated KEGG pathway models to SBML4j in python using pysbml4j.

phenotypes. Since these mutations can occur in arbitrary genes, it is useful to understand the role of the altered genes in their physiological context, i.e., within the context of their regulatory networks. By examining the network neighborhood of an altered gene, potential new treatment approaches can be identified for patients without other treatment options (e.g., through targeted therapies). Examining the interplay of gene-drug interactions using networks gives insights into the effect of an intervention, for example, for a patient resistant to a drug. If the altered gene is not a drug target or cannot be targeted because of drug resistance or intolerance of the patient, the genes up- or downstream of it might be suitable drug targets.

Setting up SBML4j for PeCaX

The networks are provided by an instance of SBML4j. Before the first use of SBML4j, the required volumes for the database need to be set up. This can be done with the shell script `./sbml4j.sh -i` at the root folder of the pecax-compose project. It will create the three volumes `sbml4j_service_vol`, `sbml4j_neo4j_vol` and `sbml4j_documentation_vol`, which are used to store the SBML4j logfiles, the neo4j database files and the API-documentation data, respectively.

For security reasons, the SBML4j service is not exposed to the host machine by default when PeCaX is run via the docker-compose setup. A direct interaction with SBML4j is necessary to populate the database with data. This can be enabled by temporarily exposing the SBML4j port on the host machine via the appropriate setting in the *docker-compose.yml* file.

The database of SBML4j is populated with 61 cancer-related pathways from the KEGG pathway database (see Tables 3.6 and 3.7).

Those pathway files were downloaded in the KGML format from the KEGG website and translated to SBML using KEGGtranslator in version 2.5 [126,127]. The command, including all command-line arguments for the translation with KEGGtranslator are listed in the appendix under Listing E.3.

```
collUUID = client.createPathwayCollection(
             "KEGG61-97.0",
             "Collection pathway for all 61 KEGG pathways",
             pathwayUUIDs
           )
```

Listing 5.2: Python command for creating a collection for the previously uploaded pathways

```
resp = client.mapPathway(collUUID, "PATHWAYMAPPING", "PWM-KEGG-61PW")
```

Listing 5.3: Python command for creating a network mapping of the artifical type PATHWAYMAPPING from the previously created pathway collection.

After translation, the SBML models were uploaded to the SBML4j REST interface using python (see Listing 5.1). To be able to upload the models, the SBML4j service and the database container need to be running on the host. They will start up alongside PeCaX when `docker compose up pecax` is run, or can be started separately with the command `docker compose up sbml4j`.

These 61 pathways are collected in a PathwayCollection (see Listing 5.2) and a network mapping is created from this collection (see Listing 5.3). The mapping has the artificial type "PATHWAYMAPPING", which includes all available node types, relationship types as well as reactions and their reaction partners. This provides a comprehensive overview of many cellular processes that affect the development and sustainability of malignant carcinoma.

As a final preparation step, `Drug-Target` information was added to the created network mapping. This information was taken from DrugBank[166]. The specific data can be found in the "Drug target identifiers" file for all approved drug groups, which can be downloaded from the DrugBank website[167]. This will allow a broad view on available and possible drugs and the genes and gene products they target. This data was combined with the DrugBank vocabulary to map the DrugBankID of drugs to their name, which is available at `https://go.drugbank.com/releases/latest#open-data`.

An R script has been used to combine the two files and prepare the data for upload to SBML4j and can be found in the appendix under Listing E.9.

The resulting file "drug_genes_approved.csv" can then be provided as annotation data for the previously created network mapping. The file is uploaded to the `/networks/<UUID>/csv` endpoint, where `<UUID>` is to be replaced by the UUID of mapping that has been created earlier. With Python, this network can be selected by its name and the annotation file provided to the `addCsvData` method on this network. All network nodes, whose names match the values of the configured symbol-column in the CSV-file are annotated. A boolean annotation with the name "Drugtarget" indicates that a node has been matched. It receives additional annotation elements with the column data from the CSV-file. The "networkname"-parameter is used to

name the resulting network of this annotation operation. The network will be created as new instance and will be derived from the initial network that is used for annotation (i.e., PWM-KEGG-61PW).

After this operation finished successfully, the SBML4j database contains a network mapping named “PeCaX-Base”, which is enriched with drug-target information from Drugbank in the form of node annotations.

SBML4j is configured to use the network with this name as base network when using the `/overview`-endpoint. The neighborhood networks that are created during the analysis of PeCaX are derived from this network.

After the successful creation and configuration of the SBML4j database, the exposed port was closed again to prevent unauthorized access to the network data. Before starting the PeCaX application with the initialized network database, a backup of this state was created using the provided shell script and the command `./sbml4j.sh -b PeCaX-Base`.

Retrieving Networks from SBML4j

For each table in the ClinVAP report, a list of genes is assembled and provided to the `/overview` endpoint in a POST request.

A GET request to the same endpoint `/overview` with the parameter “name” matching the provided name during creation (i.e., “Drivergene_network”) will retrieve the network in the GraphML format. A HTTP response with code 204 is returned, if the creation of the overview network is not yet finished.

5.3.3 Interactive Graphical User Interface

PeCaX provides a simple graphical user interface to upload variants and display the results. The report generated by ClinVAP is displayed in an interactive tabular form next to the networks generated by SBML4j.

For an easier analysis of the networks, they are visualized as network graphs using BioGraphVisart[125]. The goal of the network analysis is to not only see the individual component but also the local neighborhood crosstalk with known pathways, and nearby options for therapeutic intervention (druggable genes). BioGraphVisart is a web-based tool written in Javascript. It automates the layout of the network graph, the labeling of nodes (genes, drugs) and edges (interactions), the edge style for different interaction types, the node coloring according to easily modifiable node attributes, and the generation of legends. In addition, human genes and proteins can be grouped with respect to predefined pathways from KEGG.

5.3.4 Data Management

The user has to give a project name before starting analysis. This name is used to create a collection in the local database (ArangoDB) used for data management. By that, the user can perform multiple analyses gathered in one collection, e.g., different patients presented in one MTB session. When an analysis is started, the uploaded data and set parameters are sent via REST API to the ClinVAP container and a unique job id is generated to store the parameters. The uploaded data is only stored during the clinical annotation and is removed once the process finishes. The results of the analysis as well as the ids of the networks generated with SBML4j are stored related to the job ID. The networks themselves are stored in the network database.

5.3.5 Deployment

PeCaX can be installed locally on a personal computer or for groups of users in an access controlled intranet. Containerization enables convenient deployment without complex software installation and configuration.

5.4 Results

Overview of PeCaX

PeCaX is a comprehensive GUI-based clinical decision support tool that requires no programming knowledge. Users can perform clinical annotation and gene drug interaction network analysis via the interactive graphical interface. PeCaX provides data security as it comes in Docker containers and all analyses are performed on local infrastructure. It is supported by all modern web browsers across platforms. Hence, it is easily integrated into diagnostic and MTB workflows to investigate the relevance of single variants, complete cases or cohorts, e.g. from GWAS.

We provide a web page with example data for demonstration purpose only at `https://pecax.informatik.uni-tuebingen.de`.

Data Upload

To annotate and analyze a data set, PeCaX requires (local) submission of the data. All data is assigned to a specific *project*, a generic way of grouping data sets and results (e.g., one project per tumor board meeting or for one tumor entity). PeCaX requires one VCF file as input containing information on SNVs and (optionally) a TSV file containing information on CNVs.

The specification of the VCF file has to adhere to the fileformat definition of `VCFv4.1` [168] or higher for PeCaX to be able to interpret the contained data. The following columns have to be present in tab-delimited format: `CHROM`, `POS`, `ID`, `REF`, `ALT`, `QUAL`, `FILTER`, `INFO`, `FORMAT`.

Mandatory data columns are `CHROM`, `POS`, `REF`, and `ALT`, while the other columns will be ignored by PeCaX. The optional file containing information about copy number variations has to be formatted as tab-delimited text file with file extension *.tsv* and contain the fields `size`, `type`, `copy_number`, `gene`, `exons`, `transcript`, `chr`, `start`, `end`, `effect`. The required columns for successful application of the contained CNV information are the `type` and `gene` columns, where the gene is to be provided in HGNC gene nomenclature. Example data for both formats can be found at the official GitHub repository for PeCaX[169].

In addition, the assembly of the human reference genome used in the mapping of the sequencing data is required (both GRCh37 and GRCh38 are supported). The clinical variant annotation can be filtered by a pre-selected cancer diagnosis given as ICD10 code. After uploading, the data is automatically submitted to a local instance of ClinVAP. In order to ensure data privacy, the VCF files are removed from the containers after processing by ClinVAP. The results of the variant annotation are stored in the project database in JSON format together with associated metadata (e.g., project name and the job ID). The user can also upload a JSON file from an earlier analysis which will skip ClinVAP and populate the data tables and networks from the data contained in the provided file and the linked networks. Alternatively, a job ID of an already executed analysis can be entered at the startpage. This reopens the associated session in the UI and loads the linked networks from the network database. These job IDs can be inspected on a dedicated subpage for each project where the user can select and delete them individually. Deletion of a job ID removes all information stored for this ID in the project database as well as the generated networks from the network database to ensure data privacy.

Interactive Visualizations

The results of the clinical variant annotation performed is structured into several sections, which are all rendered as interactive, responsive tables. The first section contains the list of known cancer driver genes along with the somatic mutations observed in the patient. The list of drugs with the evidence of targeting a specific variant of the mutated gene and the documented drug response for the given mutational profile are displayed in the second section. The third section contains information on somatic mutations in pharmaceutical target affected genes and consists of two tables: therapies that have evidence of targeting the affected gene and the list of cancer drugs targeting the mutated gene. The fifth section contains the list of drugs with known adverse effects. References supporting the results found are displayed in a sixth section and all the somatic variants of the patient with their dbSNP and COSMIC IDs are listed in the last section.

Figure 5.2 shows an example table of the results visualization. Each column of each table can be queried, filtered and sorted individually (Figure 5.2 (1)). The interactive table view supports a wide range of table operations in order to simplify navigation of the data,

Figure 5.2: Interface for the interactive visualization of a table. (1) Filter table. (2) Sort table or hide/show column. (3) Highlight row across all tables by gene. (4) Genes and mutations are linked to external databases. (5) User can take notes which are saved. (6) Download table as PDF. (7) References directly link to their PubMed or clinicaltrials.gov entry.

for example, hiding/showing columns (Figure 5.2 (2)), highlighting of rows across sections (Figure 5.2 (3)). For each gene listed, the tables contain links to various external data sources such as Uniprot[170], KEGG[44] or Ensembl[171]. The links can be accessed via the drop-down menu next to the gene symbol highlighting of rows across sections (Figure 5.2 (3)) and open in a separate browser tab or window. Likewise, the references given in the tables are directly linked to the web page of the related publication on PubMed or clinicaltrials.gov highlighting of rows across sections (Figure 5.2 (7)). At the end of each section, users can add notes to be stored along with the annotated data in the internal database highlighting of rows across sections (Figure 5.2 (5)). These notes can be used to record conclusions from the analysis of the data and can be downloaded together with the table as PDF highlighting of rows across sections (Figure 5.2 (6)).

When at least one gene symbol of a table can be associated with an entry in the SBML4j database, a network for this table will be generated and is displayed next to it (see Figure 5.3). The networks consist of nodes (genes, drugs) and edges (interactions). Genes found in the table are colored red and labelled with the gene symbol. Drugs associated with any of the genes in the network are represented as diamonds and are labelled with the drug name (Figure 5.3

Input file	Clinical Annotation	Network Generation	Overall
VCF	45.48	58.19	91.83
VCF + TSV	177.38	203.38	352.36
Overall	103.19	121.71	205.81

Table 5.1: Average processing time [s].

(1)). If multiple drugs have the same gene target, they are merged into one node which is expandable in order to make the network representation visually more concise. Different interaction types (e.g., signaling, regulation) are depicted by different edge styles (Figure 5.3 (2)). If two nodes have multiple interactions, their edges are merged into one, which can be deactivated by the user (Figure 5.3 (3)). Since drug and gene names may become very long, they are shortened and moving the mouse over a node reveals the full node name. Edge types are treated in the same manner (Figure 5.3 (4)). The layout can be changed by the user based on five different layout types or the user can drag the nodes or the whole network manually to arrange them in the most informative layout (Figure 5.3 (7)). The nodes are searchable by the node label (Figure 5.3 (9)) and deletable including connected edges. Gene nodes can be grouped and highlighted by associated KEGG pathways (Figure 5.3 (5)). A mouse click on a drug node links to an overview page with links to external databases containing information on this drug such as Drugbank[166], HGNC[172], and PDB[173].

Exporting Tables and Networks

The clinical variant annotation report including the stored user-created notes and manual annotations can be downloaded as a whole in PDF and JSON format or every table individually in PDF format (Figure 5.2 (6)).

The gene drug interaction networks are available for download individually in the formats PNG, SVG (Figure 5.3 (12)) and GraphML.

Performance

From submitting a VCF file until the full report is displayed, PeCaX takes about 92 s on average. Clinical annotation takes 45.48 s on average, while network generation is finished after 58.19 s on average. For a VCF-file in combination with a related TSV-file, average computation time increases to 352 s. Overall, PeCaX needs 205 s on average for the analysis of the data until the results are displayed. A detailed performance evaluation of the processing time for the example data can be found in the appendix in table D.3. A detailed performance evaluation of ClinVAP and the results of a stress-test on large-scale data have been published previously[162].

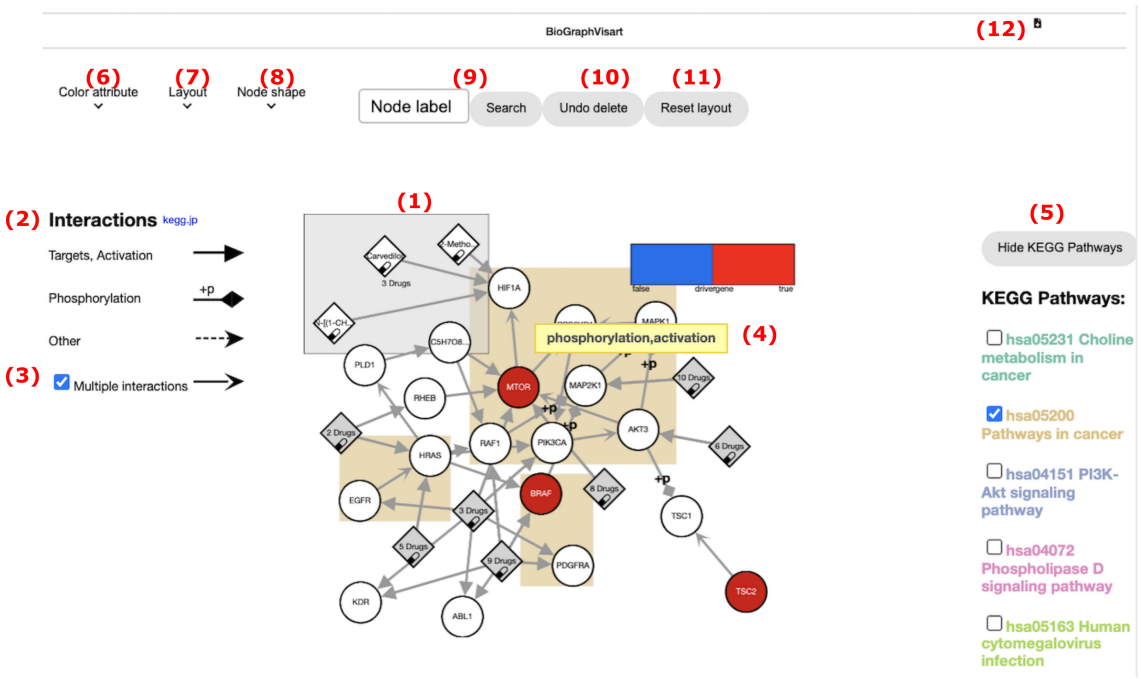


Figure 5.3: Interface for the visualization of a network. (1) Gene nodes have a circle shape, drug nodes a diamond shape. Multiple drugs targeting the same gene can be collapsed into one node. (2) Edge style legend. (3) Collapse/expand multiple edges between two nodes. (4) Moving the mouse over an edge displays the interactions. Moving it over a node displays the node label and secondary names. (5) Group nodes by KEGG pathways. (6) Selection of node color attribute. (7) Layout selection. (8) Highlight Boolean node attribute by node shape. (9) Search a node by its label and highlight it. (10) Withdrawal of node deletion achieved by right click on a node. (11) The network itself and each individual node can be dragged. The layout can be reset. (12) Download of the visualization in the formats PNG, SVG.

5.5 Conclusion

The individual nature of the genomic alterations causing cancer directly implies a personalized, or at least stratified approach to treating cancer if the underlying alterations are known. With the rapid drop in sequencing cost, sequencing has become routine for most cancers, but the interpretation of this data is still a major hurdle to clinical implementation of personalized oncology.

With PeCaX we present a novel tool for the exploration of the mutational landscape of a cancer patient and for treatment hypothesis generation. It is deployed in Docker containers guaranteeing full reproducibility independent of the operating system and as it is a local application it ensures data security and privacy. A local results database is used to keep track of the results and notes taken by the user. The combination of clinical variant annotation, gene drug interaction networks visualizing somatic variants in their pathway context and interactive web-based visualizations makes PeCaX unique and ensures ease of use for all users without the

requirement of programming experience. Its service-oriented architecture, the front end, the graph database in the back end, and the interactive graph visualization components constitute significant developments and in their combination in PeCaX a significant advancement over the mere tabular annotation of somatic variants. PeCaX supports the diagnostic workflow, e.g. in a Molecular Tumor Board, to reach transparent personalized therapeutic decisions in a shorter amount of time.

In the future, we plan to allow the upload of multiple VCF files at once for an easier comparison between patients. PeCaX might also include information on the patient's background provided by the user, e.g., gender, age, and previous therapies. Additionally, a quantitative usability study needs to be performed.

Chapter 6

Conclusion and Outlook

Elucidating the impact on the cellular machinery of one or multiple genes, of mutations in their sequence and aberrations in their expression is a non-trivial task. The interplay between the individual components of the cellular processes is complex and their consequences cannot be detected easily. To help scientists and clinicians to find these types of impacts and discover emerging properties, the processes as a whole have to be considered. A useful tool in this endeavor are biological networks that represent the molecular entities as nodes and the interactions as relationships or edges between them. Unfortunately, many research publications deem the reproducibility of their networks of lower importance than the findings they made using these networks. In the clinical setting, where complex diseases demand for innovative treatment plans and uncharted pharmaceutical interventions, these types of networks are becoming ever more important. Personalized medicine is looking at the disease mechanisms as a whole and searches for drug targets at one point in the cellular machinery that may have an effect at an entirely different location.

To tackle this twofold problem, biological networks must therefore, on the one hand, become more reproducible and, on the other hand, need to be easily accessible for clinicians and researchers. With SBML4j we provide a single network resource that addresses both these challenges. As a service-oriented application, SBML4j offers widely used and standardized access methods in the form of a RESTful interface and an intuitive custom python client library. With these, existing toolchains and research pipelines can use SBML4j as a central storage of biological network information with the capabilities for creating knowledge-driven neighborhood networks and shortest-path calculations. By providing means for annotating the networks with arbitrary data in a privacy preserving way, SBML4j can be used to map sensitive patient data for personalized exploratory analysis. We integrated SBML4j in PeCaX, the clinical variant exploration tool that enables visual exploration of the genetic neighborhoods of affected genes. With the integration of drug-target information, it is becoming a valuable tool in the prepa-

ration of Molecular Tumor Board Meetings. With presentation-ready network visualizations, PeCaX demonstrates the integration of SBML4j as a network resource for the clinical setting.

At the same time, the provenance of any network stored in SBML4j is fully preserved, even when external tools are used to manipulate the networks. The standardized network source of SBML models allows the construction of a comprehensive knowledge graph. This graph bridges traditional pathway boundaries by integrating individual entities through their common biological qualifier information. By encoding biological entities and relationships through well-known ontologies, SBML4j enables the creation of a variety of specialized network mappings. These mappings allow differentiated views on the cellular processes and their creation is fully tracked in the integrated provenance graph. The extensive possibilities for manipulation and annotation of these network mappings facilitate the creation of personalized network representations for various use cases. Each of the performed steps is recorded, the intermediate networks are preserved and the provenance of all networks is stored and retrievable. The detailed provenance report includes all performed steps, the mapping process and information about the SBML source models to enable full reproducibility. All parameters and options of requests to the REST API are tracked and included in the report. In addition, externally modified networks can be re-uploaded and details about this external provenance step added to the provenance graph.

With these features, SBML4j enables researchers to provide comprehensive provenance details for biological networks in their publications without much overhead. An SBML4j provenance report, along with the publicly available SBML source models, enables the reproduction of reported networks and actively fights back against the raging reproducibility crisis. In the clinical setting, SBML4j acts as a comprehensive source of biological networks. With its easy access methods, the containerized deployments scheme and the versatile manipulation and exploration features, SBML4j is a valuable resource for achieving truly personalized medicine.

The future development of SBML4j may increase the SBML feature set by integrating more of the already available SBML extensions. At the same time, additional biological qualifier systems can be included in the knowledge graph, as well as the connection to more ontologies for more widespread adoption. With the concentration on SBML as reliable source models, it can be beneficial to the systems biology community to implement an export mechanism for networks in the SBML format. To further the reproducibility of the created networks, a means of reading in a previously generated provenance report to recreate the described network in an unattended fashion will be advantageous. With the advancement of the graph-database technology, more options for graph-algorithmic calculations become available. For instance, the newly introduced Neo4j "`Graph Data Science`" package can speed up neighborhood network generation significantly and should be considered in future releases of SBML4j. Likewise, the reactive development paradigm can be implemented with newer releases of the Java Spring

framework and has the potential to dramatically increase productivity and reduce database load.

In conclusion, SBML4j delivers an extensive feature set for working with biological networks, enables the creation of reproducible network representations and can be seamlessly integrated in existing infrastructure and tool chains. It is a valuable tool for the elucidation of cellular processes, the discovery of novel therapeutic interventions and it furthers reproducible science. Since it is an open source project, all it needs is a continuously active community to drive its development forward.

Bibliography

[1] Nurk S., et al. (2022). The complete sequence of a human genome. *Science*, 376(6588):44–53. 1

[2] Jain K. K. *Personalized medicine: the impact of pharmacogenetics on drug development*. Decision Resources (1998). 5

[3] Jain K. K. (2002). Personalized medicine. *Current opinion in molecular therapeutics*, 4(6):548–558. 5

[4] Goldstein D. B., Tate S. K., and Sisodiya S. M. (2003). Pharmacogenetics goes genomic. *Nature Reviews Genetics*, 4(12):937–947. 5

[5] Matthews H., Hanison J., and Nirmalan N. (2016). "omics"-informed drug and biomarker discovery: opportunities, challenges and future perspectives. *Proteomes*, 4(3):28. 5

[6] Druker B. J., et al. (2001). Activity of a specific inhibitor of the bcr-abl tyrosine kinase in the blast crisis of chronic myeloid leukemia and acute lymphoblastic leukemia with the philadelphia chromosome. *New England Journal of Medicine*, 344(14):1038–1042. 5

[7] Deininger M. W. and Druker B. J. (2003). Specific targeted therapy of chronic myelogenous leukemia with imatinib. *Pharmacological reviews*, 55(3):401–423. 5

[8] Frueh F. W., et al. (2008). Pharmacogenomic biomarker information in drug labels approved by the united states food and drug administration: prevalence of related drug use. *Pharmacotherapy: The Journal of Human Pharmacology and Drug Therapy*, 28(8):992–998. 5

[9] Keller A., et al. (2009). A novel algorithm for detecting differentially regulated paths based on gene set enrichment analysis. *Bioinformatics*, 25(21):2787–2794. 5

[10] Schärfe C. P. I., Tremmel R., Schwab M., Kohlbacher O., and Marks D. S. (2017). Genetic variation in human drug-related genes. *Genome medicine*, 9(1):1–15. 5, 34

[11] Löffler M. W., et al. (2016). Personalized peptide vaccine-induced immune response associated with long-term survival of a metastatic cholangiocarcinoma patient. *Journal of hepatology*, 65(4):849–855. 5

[12] Schubert B. and Kohlbacher O. (2016). Designing string-of-beads vaccines with optimal spacers. *Genome medicine*, 8(1):1–10. 5

[13] Fröhlich H., et al. (2018). From hype to reality: data science enabling personalized medicine. *BMC medicine*, 16(1):1–15. 5

[14] Trusheim M. R., Berndt E. R., and Douglas F. L. (2007). Stratified medicine: strategic and economic implications of combining drugs and clinical biomarkers. *Nature reviews Drug discovery*, 6(4):287–293. 6

[15] Knepper T. C., et al. (2017). Key lessons learned from moffitt's molecular tumor board: The clinical genomics action committee experience. *The oncologist*, 22(2):144. 7

[16] A to Z List of Cancer Types of the National Cancer Institute. `https://www.cancer.gov/types`. Accessed: 2022-10-21. 7

[17] Stehelin D., Varmus H. E., Bishop J. M., and Vogt P. K. (1976). Dna related to the transforming gene (s) of avian sarcoma viruses is present in normal avian dna. *Nature*, 260(5547):170–173. 8

[18] Oppermann H., Levinson A. D., Varmus H. E., Levintow L., and Bishop J. M. (1979). Uninfected vertebrate cells contain a protein that is closely related to the product of the avian sarcoma virus transforming gene (src). *Proceedings of the National Academy of Sciences*, 76(4):1804–1808. 8

[19] Weinberg R. A. (1991). Tumor suppressor genes. *Science*, 254(5035):1138–1146. 9

[20] Fearon E. R. (1997). Human cancer syndromes: clues to the origin and nature of cancer. *Science*, 278(5340):1043–1050. 9

[21] Kinzler K. (1998). The genetics basis of human cancer. *Colorectal Tumors*, pages 565–587. 9

[22] Heinen C. D., Schmutte C., and Fishel R. (2002). Dna repair and tumorigenesis: lessons from hereditary cancer syndromes. *Cancer biology & therapy*, 1(5):477–485. 9

[23] Modrich P. and Lahue R. (1996). Mismatch repair in replication fidelity, genetic recombination, and cancer biology. *Annual review of biochemistry*, 65(1):101–133. 9

[24] Hoque M. O., et al. (2009). Changes in cpg islands promoter methylation patterns during ductal breast carcinoma progression. *Cancer Epidemiology and Prevention Biomarkers*, 18(10):2694–2700. 9

[25] Baylin S. B., et al. (2001). Aberrant patterns of dna methylation, chromatin formation and gene expression in cancer. *Human molecular genetics*, 10(7):687–692. 9

[26] Jones S., et al. (2010). Frequent mutations of chromatin remodeling gene arid1a in ovarian clear cell carcinoma. *Science*, 330(6001):228–231. 10

[27] Shay J. W. and Wright W. E. (2000). Hayflick, his limit, and cellular ageing. *Nature reviews Molecular cell biology*, 1(1):72–76. 10

[28] Shay J. W., Zou Y., Hiyama E., and Wright W. E. (2001). Telomerase and cancer. *Human molecular genetics*, 10(7):677–685. 10

[29] Ferrara N. (2002). Vegf and the quest for tumour angiogenesis factors. *Nature Reviews Cancer*, 2(10):795–803. 10

[30] Sridhar S. S. and Shepherd F. A. (2003). Targeting angiogenesis: a review of angiogenesis inhibitors in the treatment of lung cancer. *Lung cancer*, 42(2):81–91. 10

[31] Ng E. W. and Adamis A. P. (2005). Targeting angiogenesis, the underlying disorder in neovascular age-related macular degeneration. *Canadian Journal of Ophthalmology*, 40(3):352–368.

[32] Zhao Y. and Adjei A. A. (2015). Targeting angiogenesis in cancer therapy: moving beyond vascular endothelial growth factor. *The oncologist*, 20(6):660. 10

[33] Valastyan S. and Weinberg R. A. (2011). Tumor metastasis: molecular insights and evolving paradigms. *Cell*, 147(2):275–292. 10

[34] Thiery J. P. (2002). Epithelial–mesenchymal transitions in tumour progression. *Nature reviews cancer*, 2(6):442–454. 10

[35] Savagner P. (2001). Leaving the neighborhood: molecular mechanisms involved during epithelial-mesenchymal transition. *Bioessays*, 23(10):912–923. 10

[36] Thiery J. P., Acloque H., Huang R. Y., and Nieto M. A. (2009). Epithelial-mesenchymal transitions in development and disease. *cell*, 139(5):871–890. 10

[37] Kochetkova M., Kumar S., and McColl S. (2009). Chemokine receptors cxcr4 and ccr7 promote metastasis by preventing anoikis in cancer cells. *Cell Death & Differentiation*, 16(5):664–673. 10

[38] Massagué J., Batlle E., and Gomis R. R. (2017). Understanding the molecular mechanisms driving metastasis. *Molecular oncology*, 11(1):3–4. 10

[39] Tamborero D., et al. (2020). Support systems to guide clinical decision-making in precision oncology: The cancer core europe molecular tumor board portal. *Nature Medicine*, 26(7):992–994. 11

[40] Karolinska - MTB. `https://mtbp.org/`. Accessed: 2021-11-26. 11

[41] Halfmann M., Stenzhorn H., Gerjets P., Kohlbacher O., and Oestermeier U. User-driven development of a novel molecular tumor board support tool. In *International Conference on Data Integration in the Life Sciences*, pages 195–199. Springer (2018). 11

[42] Chakravarty D., et al. (2017). Oncokb: a precision oncology knowledge base. *JCO precision oncology*, 1:1–16. 11

[43] Szklarczyk D., et al. (2019). String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613. 11

[44] Kanehisa M. and Goto S. (2000). Kegg: kyoto encyclopedia of genes and genomes. *Nucleic acids research*, 28(1):27–30. 11, 12, 13, 124

[45] National Human Genome Research Institute: Biological Pathways Fact Sheet. https://www.genome.gov/about-genomics/fact-sheets/Biological-Pathways-Fact-Sheet. Accessed: 2022-10-21. 12

[46] Maier D. (2006). Hairless: the ignored antagonist of the notch signalling pathway. *Hereditas*, 143(2006):212–221. 13

[47] Yergeau F., Bray T., Paoli J., Sperberg-McQueen C. M., and Maler E. (2004). Extensible markup language (xml) 1.0. *W3C Recommendation*, 4:220. 14

[48] Le Novère N., et al. (2009). The Systems Biology Graphical Notation. *Nature Biotechnology*, 27(8):735–741. 14

[49] KEGG Markup Language. https://www.kegg.jp/kegg/xml/. Accessed: 2022-10-21. 14, 97

[50] The GraphML File Format. http://graphml.graphdrawing.org. Accessed: 2022-10-21. 14

[51] Demir E., et al. (2010). The biopax community standard for pathway data sharing. *Nature biotechnology*, 28(9):935–942. 14

[52] Hucka M., et al. (2019). The Systems Biology Markup Language (SBML): Language specification for Level 3 Version 2 Core Release 2. *Journal of Integrative Bioinformatics*, 16(2):20190021. 14

[53] Combine: Coordinating standards for modeling in Biology. http://www.co.mbine.org/. Accessed: 2022-10-21. 14

[54] Specification Document of SBML Level 3 version 2 core. https://github.com/sbmlteam/sbml-specifications/blob/release/sbml-level-3/version-2/core/spec/sbml-level-3-version-2-release-2-core.pdf. Accessed: 2022-10-21. 16

[55] SBML.org: SBML Specifications. https://synonym.caltech.edu/documents/specifications/. Accessed: 2022-10-21. 16

[56] Chaouiya C., et al. (2015). Sbml level 3 package: qualitative models, version 1, release 1. *Journal of integrative bioinformatics*, 12(2):691–730. 16

[57] SBML.org: SBML Level 3 Qualitative Models. https://synonym.caltech.edu/documents/specifications/level-3/version-1/qual/. Accessed: 2022-10-21. 16

[58] Systems Biology Ontology. https://www.ebi.ac.uk/ols/ontologies/sbo. Accessed: 2022-10-21. 17

[59] Courtot M., et al. (2011). Controlled vocabularies and semantics in systems biology. *Molecular systems biology*, 7(1):543. 19

[60] COMBINE: Biomodels.net qualifier website. https://co.mbine.org/standards/qualifiers. Accessed: 2022-10-15. 19, 20

[61] Juty N., et al. (2015). Biomodels: content, features, functionality, and use. *CPT: pharmacometrics & systems pharmacology*, 4(2):55–68. 19, 20

[62] Havugimana P. C., et al. (2012). A census of human soluble protein complexes. *Cell*, 150(5):1068–1081. 21

[63] Remy I. and Michnick S. W. (2007). Application of protein-fragment complementation assays in cell biology. *Biotechniques*, 42(2):137–145. 21

[64] Brückner A., Polge C., Lentze N., Auerbach D., and Schlattner U. (2009). Yeast two-hybrid, a powerful tool for systems biology. *International journal of molecular sciences*, 10(6):2763–2788. 21

[65] Orth J. D., Thiele I., and Palsson B. Ø. (2010). What is flux balance analysis? *Nature biotechnology*, 28(3):245–248. 21

[66] Bordbar A., Monk J. M., King Z. A., and Palsson B. O. (2014). Constraint-based models predict metabolic and associated cellular functions. *Nature Reviews Genetics*, 15(2):107–120. 21

[67] Alvarez M. J., et al. (2016). Functional characterization of somatic mutations in cancer using network-based inference of protein activity. *Nature genetics*, 48(8):838–847. 21

[68] Margolin A. A., et al. Aracne: an algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context. In *BMC bioinformatics*, volume 7, pages 1–15. Springer (2006). 21

[69] Fielding R. T. *Architectural styles and the design of network-based software architectures*. University of California, Irvine (2000). 21

[70] Nelson T. H. Complex information processing: a file structure for the complex, the changing and the indeterminate. In *Proceedings of the 1965 20th national conference*, pages 84–100 (1965). 21

[71] Fielding R., et al. Hypertext transfer protocol–http/1.1. 22

[72] OpenAPI-Specification/3.1.0.md at main · OAI/OpenAPI-Specification. `https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md`. Accessed: 2022-10-21. 23

[73] Francis N., et al. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445 (2018). 23

[74] Neo4j Graph Platform — The Leader in Graph Databases. `https://neo4j.com/`. Accessed: 2022-10-21. 23

[75] Awesome Procedures On Cypher (APOC) - Neo4j Labs. `https://neo4j.com/labs/apoc/`. Accessed: 2022-10-21. 26

[76] Pasquier T., et al. (2017). If these data could talk. *Scientific data*, 4(1):1–5. 26

[77] Di Tommaso P., et al. (2017). Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319. 26

[78] Berthold M. R., et al. (2009). Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, 11(1):26–31. 26

[79] Köster J. and Rahmann S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522. 26

[80] Kuenzi B. M. and Ideker T. (2020). A census of pathway maps in cancer systems biology. *Nature Reviews Cancer*, 20(4):233–246. 26, 95, 96

[81] Missier P., Belhajjame K., and Cheney J. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 773–776 (2013). 26

[82] The PROV data model. `https://www.w3.org/TR/2013/REC-prov-dm-20130430`. Accessed: October 2020. 26

[83] Li C., et al. (2010). Biomodels database: An enhanced, curated and annotated resource for published quantitative kinetic models. *BMC systems biology*, 4(1):1–14. 30

[84] Pico A. R., et al. (2008). Wikipathways: pathway editing for the people. *PLoS Biol*, 6(7):e184. 30

[85] Pratt D., et al. (2015). Ndex, the network data exchange. *Cell systems*, 1(4):302–305. 30

[86] NDEx WebApp. `https://www.ndexbio.org`. Accessed: 2022-10-21. 30

[87] Jassal B., et al. (2020). The reactome pathway knowledgebase. *Nucleic acids research*, 48(D1):D498–D503. 30

[88] Swainston N., et al. (2016). Recon 2.2: from reconstruction to model of human metabolism. *Metabolomics*, 12(7):1–7. 30

[89] SEMS | Model Management & Standards for Computational Biology. `https://sems.bio.informatik.uni-rostock.de`. Accessed: 2022-10-21. 30

[90] Henkel R. and Waltemath D. Masymos: Finding hidden treasures in model repositories. In *SWAT4LS*. Citeseer (2014). 30

[91] Touré V., et al. (2016). Ston: exploring biological pathways using the sbgn standard and graph databases. *BMC bioinformatics*, 17(1):1–9. 30

[92] Balaur I., et al. (2017). Recon2neo4j: applying graph database technologies for managing comprehensive genome-scale networks. *Bioinformatics*, 33(7):1096–1098. 30

[93] Lysenko A., et al. (2016). Representing and querying disease networks using graph databases. *BioData mining*, 9(1):23. 30

[94] Dubovenko A., Nikolsky Y., Rakhmatulin E., and Nikolskaya T. Functional analysis of omics data and small molecule compounds in an integrated "knowledge-based" platform. In *Biological Networks and Pathway Analysis*, pages 101–124. Springer (2017). 31

[95] Gerasch A., et al. (2014). Bina: a visual analytics tool for biological network data. *PloS one*, 9(2):e87397. 31

[96] Gerasch A., et al. (2015). Network-based interactive navigation and analysis of large biological datasets. *it-Information Technology*, 57(1):37–48. 31

[97] Shannon P., et al. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504. 31

[98] Wiese R., Eiglsperger M., and Kaufmann M. yfiles—visualization and automatic layout of graphs. In *Graph Drawing Software*, pages 173–191. Springer (2004). 31

[99] Project Jupyter. `https://jupyter.org/index.html`. Accessed: 2022-10-21. 31

[100] QuantStack/ipycytoscape: A cytoscape Jupyter widget. `https://github.com/quantstack/ipycytoscape`. Accessed: 2022-10-21. 31

[101] pysbml4j – PyPi. `https://pypi.org/project/pysbml4j/`. Accessed: 2022-10-21. 31, 81

[102] Sadowski G. and Rathle P. (2014). Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 13. 34

[103] Johnson R., et al. (2004). The spring framework–reference documentation. *interface*, 21:27. 35

[104] Bolt Protocol. `https://boltprotocol.org`. Accessed: 2022-10-21. 37

[105] Winkler S., et al. (2022). De novo identification of maximally deregulated subnetworks based on multi-omics data with deregnet. *BMC bioinformatics*, 23(1):1–28. 37, 96, 97, 102, 106

[106] Postman API Platform | Sign Up for Free. `https://www.postman.com`. Accessed: 2022-10-21. 37

[107] Rivest R. L. The MD5 Message-Digest Algorithm. `https://www.rfc-editor.org/info/rfc1321`. Accessed: 2022-10-21. 45

[108] Spring Data Neo4j - Reference Documentation. `https://docs.spring.io/spring-data/neo4j/docs/5.2.7.RELEASE/reference/html/`. Accessed: 2022-10-21. 58

[109] Dijkstra E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271. 61

[110] uniVocity/univocity-parsers: uniVocity-parsers is a suite of extremely fast and reliable parsers for Java. It provides a consistent interface for handling different file formats, and a solid framework for the development of new parsers. `https://github.com/uniVocity/univocity-parsers`. Accessed: 2022-10-21. 70

[111] sbml4j | Dissertation | tiede | swaggerhub. https://app.swaggerhub.com/apis-docs/tiede/sbml4j/Dissertation. Accessed: 2022-10-22. 71, 74, 75

[112] API Documentation & Design Tools for Teams | Swagger. https://swagger.io. Accessed: 2022-10-21. 71

[113] thortiede/sbml4j-compose: manage and run sbml4j using docker-compose. https://github.com/thortiede/sbml4j-compose. Accessed: 2022-10-21. 71, 81, 107

[114] Thorsten Tiede. Kohlbacherlab/sbml4j: Persist biological models in sbml format in neo4j, create and explore mappings. https://github.com/KohlbacherLab/sbml4j. Accessed: 2022-10-21. 76, 80, 116

[115] Hagberg A., Swart P., and S Chult D. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2008). 77

[116] urllib3.readthedocs.io | 1.26.12 documenation. https://urllib3.readthedocs.io/en/stable/. Accessed: 2022-10-21. 78, 105

[117] The MIT license | Open Source Initiative. https://opensource.org/licenses/MIT. Accessed: 2022-10-21. 80

[118] Maven - Welcome to Apache Maven. https://maven.apache.org. Accessed: 2022-10-21. 81

[119] thortiede/sbml4j - Docker image | Docker Hub. https://hub.docker.com/r/thortiede/sbml4j. Accessed: 2022-10-21. 81

[120] swaggerapi/swagger-ui - Docker image | Docker Hub. https://hub.docker.com/r/swaggerapi/swagger-ui. Accessed: 2022-10-21. 81

[121] REST API Documenation Tool | Swagger UI. https://swagger.io/tools/swagger-ui/. Accessed: 2022-10-21. 81

[122] PyPI · The Python Package Index. https://pypi.org. Accessed: 2022-10-21. 81

[123] pip · PyPI. https://pypi.org/project/pip/. Accessed: 2022-10-21. 81

[124] KohlbacherLab/pysbml4j: python client library for interaction with sbml4j REST API. https://github.com/KohlbacherLab/pysbml4j. Accessed: 2022-10-21. 81

[125] Mirjam Figaschewski. Kohlbacherlab/biographvisart: Web-based visualization of graphml files using cytoscape.js. https://github.com/KohlbacherLab/BioGraphVisart. Accessed: 2022-10-21. 82, 108, 116, 121

[126] Wrzodek C., Dräger A., and Zell A. (2011). Keggtranslator: visualizing and converting the kegg pathway database to various formats. *Bioinformatics*, 27(16):2314–2315. 82, 97, 100, 119

[127] Wrzodek C., Büchel F., Ruff M., Dräger A., and Zell A. (2013). Precise generation of systems biology models from kegg pathways. *BMC systems biology*, 7(1):1–12. 82, 119

[128] Vogelstein B., et al. (2013). Cancer genome landscapes. *science*, 339(6127):1546–1558. 85

[129] Neo4j Graph Data Science | Graph Algorithms and ML | Graph Analytics. `https://neo4j.com/product/graph-data-science/`. Accessed: 2022-10-21. 90

[130] Hartig O. and Pérez J. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference*, pages 1155–1164 (2018). 91

[131] Hardt D. et al. The oauth 2.0 authorization framework. `https://www.rfc-editor.org/info/rfc6749`. Accessed: 2022-10-21. 91

[132] Stupple A., Singerman D., and Celi L. A. (2019). The reproducibility crisis in the age of digital medicine. *NPJ digital medicine*, 2(1):1–3. 95

[133] Baker M. (2016). Reproducibility crisis. *Nature*, 533(26):353–66. 95

[134] Allison D. B., Brown A. W., George B. J., and Kaiser K. A. (2016). Reproducibility: A tragedy of errors. *Nature News*, 530(7588):27. 95

[135] Begley C. G. and Ellis L. M. (2012). Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533. 95

[136] Tiwari K., et al. (2021). Reproducibility in systems biology modelling. *Molecular Systems Biology*, 17(2):e9982. 95

[137] Errington T. M., et al. (2021). Investigating the replicability of preclinical cancer biology. *eLife*, 10:e71601. 96

[138] Baker M. and Dolgin E. (2017). Cancer reproducibility project releases first results. *Nature News*, 541(7637):269. 96

[139] The Cancer Genome Atlas Program - NCI. `https://www.cancer.gov/tcga`. Accessed: 2022-10-21. 96

[140] KEGG API. `https://www.kegg.jp/kegg/rest/keggapi.html`. Accessed: 2022-10-21. 99

[141] KEGG PATHWAY database. `https://www.kegg.jp/kegg/pathway.html`. Accessed: 2022-10-21. 100

[142] thortiede/DeRegNet_Pipeline: A Pipeline to work with DeRegNet and SBML4j to create reproducible maximally deregulated subnetworks. `https://github.com/thortiede/DeRegNet_Pipeline`. Accessed: 2022-10-21. 102, 105, 106, 107, 109

[143] Project - CPTAC-3 - TCGA. `https://portal.gdc.cancer.gov/projects/CPTAC-3`. Accessed: 2022-10-21. 103

[144] Ellis M. J., et al. (2013). Connecting genomic alterations to cancer biology with proteomics: the nci clinical proteomic tumor analysis consortium. *Cancer discovery*, 3(10):1108–1112. 103

[145] Satija R., Farrell J. A., Gennert D., Schier A. F., and Regev A. (2015). Spatial reconstruction of single-cell gene expression data. *Nature biotechnology*, 33(5):495–502. 103

[146] satijalab/seurat: R toolkit for single cell genomics. `https://github.com/satijalab/seurat`. Accessed: 2022-10-21. 103

[147] Gurobi - The Fastest Solver - Gurobi. `https://www.gurobi.com/`. Accessed: 2022-10-21. 106

[148] KohlbacherLab/DeRegNet: Find deregulated signaling networks. `https://github.com/KohlbacherLab/deregnet`. Accessed: 2022-10-21. 106, 111

[149] sebwink/deregnet-grb9.5.2 - Docker Image | Docker Hub. `https://hub.docker.com/r/sebwink/deregnet-grb9.5.2`. Accessed: 2022-10-21. 106

[150] neo4j - Official Image | Docker Hub. `https://hub.docker.com/_/neo4j`. Accessed: 2022-10-21. 107

[151] LEMON. `https://lemon.cs.elte.hu/trac/lemon`. Accessed: 2022-10-21. 112

[152] LEMON: LEMON Graph File Format. `http://lemon.cs.elte.hu/pub/doc/latest-svn/lemon_file_format.html`. Accessed: 2022-10-21. 112

[153] SIF (file format) - BioUML platform. `http://wiki.biouml.org/index.php/SIF_(file_format)`. Accessed: 2022-10-21. 112

[154] Figaschewski M., Sürün B., Tiede T., and Kohlbacher O. (2023). The personalized cancer network explorer (PeCaX) as a visual analytics tool to support molecular tumor boards. *BMC bioinformatics*, 24(1):1–11. 115

[155] Jain K. K. Principles of personalized oncology. In *Textbook of Personalized Medicine*, pages 403–478. Springer International Publishing, Cham (2021). 115

[156] Hart S. N., et al. (2016). Vcf-miner: Gui-based application for mining variants and annotations stored in vcf files. *Briefings in bioinformatics*, 17(2):346–351. 116

[157] Salatino S. and Ramraj V. (2017). Browsevcf: a web-based application and workflow to quickly prioritize disease-causative variants in vcf files. *Briefings in bioinformatics*, 18(5):774–779. 116

[158] Akgün M. and Demirci H. (2017). Vcf-explorer: filtering and analysing whole genome vcf files. *Bioinformatics*, 33(21):3468–3470. 116

[159] Jiang J., Gu J., Zhao T., and Lu H. (2019). Vcf-server: A web-based visualization tool for high-throughput variant data mining and management. *Molecular Genetics & Genomic Medicine*, 7(7):e00641. 116

[160] Perera-Bel J., et al. (2018). From somatic variants towards precision oncology: evidence-driven reporting of treatment options in molecular tumor boards. *Genome medicine*, 10(1):1–15. 116

[161] Xu Q., et al. (2020). Oncopdss: an evidence-based clinical decision support system for oncology pharmacotherapy at the individual level. *BMC cancer*, 20(1):1–10. 116

[162] Sürün B., et al. (2020). Clinvap: a reporting strategy from variants to therapeutic options. *Bioinformatics*, 36(7):2316–2317. 116, 125

[163] McLaren W., et al. (2016). The ensembl variant effect predictor. *Genome biology*, 17(1):1–14. 117

[164] Forbes S., et al. (2006). Cosmic 2005. *British journal of cancer*, 94(2):318–322. 117

[165] Tamborero D., et al. (2018). Cancer genome interpreter annotates the biological and clinical relevance of tumor alterations. *Genome medicine*, 10(1):1–8. 117

[166] Wishart D. S., et al. (2008). Drugbank: a knowledgebase for drugs, drug actions and drug targets. *Nucleic acids research*, 36(suppl_1):D901–D906. 120, 125

[167] Drugbank online | database for drug and drug target info. `https://go.drugbank.com`. Accessed: 2022-10-21. 120

[168] samtools/hts-specs: Specifications of SAM/BAM and related high-throughput sequencing file formats. `https://github.com/samtools/hts-specs`. Accessed: 2022-10-21. 122

[169] KohlbacherLab/PeCaX-docker github.com. `https://github.com/KohlbacherLab/PeCaX-docker`. Accessed: 2022-10-21. 123

[170] Consortium U. (2015). Uniprot: a hub for protein information. *Nucleic acids research*, 43(D1):D204–D212. 124

[171] Hubbard T., et al. (2002). The ensembl genome database project. *Nucleic acids research*, 30(1):38–41. 124

[172] Povey S., et al. (2001). The hugo gene nomenclature committee (hgnc). *Human genetics*, 109(6):678–680. 125

[173] Kouranov A., et al. (2006). The rcsb pdb information portal for structural genomics. *Nucleic acids research*, 34(suppl_1):D302–D305. 125

Appendix A

Abbreviations

A	
API	*Application programming interface*
APOC	*Awesome Procedures on Cypher*
ATP	*Adenosine triphosphate*

B	
bfs	*breadth-first search*
BioPAX	*Biological Pathway Exchange*
bp	*base pair*

C	
CNV	*Copy-number variation*
COMBINE	*COmputational Modeling in BIology NEtwork*
CPTAC	*Clinical Proteomic Tumor Analysis Consortium*

D	
DGE	*Differential gene expression*
DI	*Dependency injection*
DNA	*Deoxyribonucleic acid*

E	
EMT	*Epithelial-mesynchymal-transition*

G	
GraphML	*Graph Markup Language*
GSEA	*Gene-set enrichment analysis*

GWAS	*Genome-wide association study*

H

hM	*hadMember (Provenance edge type)*
HNPCC	*hereditary non-polyposis colon cancer*
HPV	*Human papillomavirus*
HTTP	*Hypertext transfer protocol*

J

JSON	*JavaScript Object Notation*

K

KEGG	*Kyoto Encyclopedia of Genes and Genomes*
KGML	*KEGG markup language*
KPWD	*KEGG Pathway Downloader*

L

Log2FC	*Logarithmic (base 2) fold change*

M

ML	*Machine Learning*
MTB	*Molecular Tumor Board*

N

NIH	*National Human Genome Research Institute*

O

OCCC	*ovarian clear cell carcinoma*
OGM	*object-graph mapper*
OWL	*Web Ontology Language*

P

PPI	*Protein-protein interaction*

R

RDF	*Resource Description Format*
RefSeq	*NCBI Reference Sequence Database*
REST	*Representational state transfer*
RNA	*ribonucleic-acid*
RSV	*Rous sarcoma virus*

S

SBGN	*Systems Biology Graphical Notation*
SBGN-ML	*Systems Biology Graphical Notation Markup Language*
SBML	*Systems Biology Markup Language*
SNV	*Single-nucleotide variant*
SBO	*Systems Biology Ontology*
SQL	*Structured Query Language*
STON	*SBGN To Neo4j*

T

TCGA	*The Cancer Genome Atlas*
TSG	*Tumor Suppressor Gene*
TSV	*Tabulator Separated Values*

U

URI	*Uniform resource identifier*
URL	*Uniform Resource Locator*

W

W3C	*World Wide Web Consortium*
wAT	*wasAttributedTo (Provenance edge type)*
wAW	*wasAssociatedWith (Provenance edge type)*
wDF	*wasDerivedFrom (Provenance edge type)*
wGB	*wasGeneratedBy (Provenance edge type)*

X

XML	*eXtensible Markup Language*

Appendix B

Contributions

All ideas, approaches and results presented in this work were developed and discussed with my supervisors Prof. Dr. Oliver Kohlbacher (OK) and Prof. Dr. Kay Nieselt. The following co-workers also contributed to **Chapter 5 Application: Personalized Cancer and Network Explorer – PeCaX**:

- Bilge Sürün (BS)
- Mirjam Figaschewski (MF)

MF wrote the first draft of the manuskript. BS and Thorsten Tiede (TT) revised the first draft. MF, BS and TT reworked the manuskript together a final time. OK provided final corrections to this manuskript.
TT extended the manuskript for this thesis in the areas of his work, notably the parts concerning the tool SBML4j and the network generation.
BS implemented the Variant annotation part of the software, TT implemented the graph service part of the software, MF implemented the Graph visualization part and the graphical user interface. MF, BS and TT built the enclosing software setup that ties the three components together.
The Software that TT contributed to this publication is an essential building block and delivers the novel feature to view the data in the genetic neighborhood via biological networks.

Appendix C

Publications

Accepted manuscripts

Sebastian Winkler, Ivana Winkler, Mirjam Figaschewski, Thorsten Tiede, Alfred Nordheim & Oliver Kohlbacher
"De novo identification of maximally deregulated subnetworks based on multi-omics data with DeRegNet"
BMC Bioinformatics **23**, 139 (2022) DOI: 10.1186/s12859-022-04670-6

Book Chapters

Matthias Ganzinger, Enrico Glaab, Jules Kerssemakers, Sven Nahnsen, Ulrich Sax, Nadine Sarah Schaadt, Matthieu-P. Schapranow, ThorstenTiede
"Biomedical and Clinical Research Data Management"
Systems Medicine — Integrative, Qualitative and Computational Approaches **Volume 3**, 2021, Pages 532–543

Submitted manuscripts

Mirjam Figaschewski, Bilge Sürün, Thorsten Tiede, Oliver Kohlbacher
"The Personalized Cancer Network Explorer (PeCaX) as a visual analytics tool
to support molecular tumor boards"
BMC Bioinformatics

Appendix D

Supporting Tables

Table D.1: Available annotation-names for which a custom separator can be configured with the sbml4j.annotation config option.

annotation name
keggGenesSeparator
entrezGeneSeparator
ensembleSeparator
hgncSeparator
omimSeparator
uniprotSeparator
obo_chebiSeparator
pdb_ccdSeparator
ecCodeSeparator
secondaryNamesSeparator
pathwaysSeparator

Table D.2: REST endpoints of SBML4j

Request type	Endpoint URI	Function description	Return type
GET	/**networks**	List available networks	json
POST	/**networks**	Create a copy of a network	json
GET	/networks/**{UUID}**	Retrieve network by UUID	GraphML
DELETE	/networks/**{UUID}**	Delete network by UUID	header only
POST	/networks/{UUID}/**csv**	Provide a csv-file with node-annotation information	json
GET	/networks/{UUID}/**context**	Get network context for one or multiple input node symbols	GraphML
POST	/networks/{UUID}/**context**	Create a context from one or multiple input node symbols	json
GET	/networks/{UUID}/**options**	Get network options	json
POST	/networks/{UUID}/**filter**	Filter network	json
POST	/networks/{UUID}/**annotation**	Provide a json-body with node-annotation information	json
GET	/**overview**	retrieve a previously created overview network for a user by its name	GraphML
POST	/**overview**	Convenience endpoint to create context and add annotation	json
POST	/**sbml**	Upload SBML files	json
POST	/**graphml**	Upload GraphML files	json
GET	/**pathways**	List available pathways	json
POST	/**pathwayCollection**	Create a pathway collection from given pathways	json
POST	/**mapping**/{UUID}	Map pathway contents to a new network mapping	json
GET	/**databaseUUID**	Get the uuid of a database node for source with version	json
GET	/**fileorigin**/{UUID}	Get the uuid of the file node that connects via wasDerivedFrom	json
PUT	/**prov**/{UUID}/	Add provenance annotation to an entity	json
GET	/**prov**/{UUID}/	Retrieve provenance report of an entity	json

Table D.3: Processing time [s] for the steps of PeCaX. Processing times of the clinical annotation from the time of job submission until the report is displayed, network generation from receiving the genes until the networks are displayed, and overall time from submitting a job until the full report is displayed. Data was analyzed three times.

Data	Clinical Annotation			Network generation			Overall		
	1	2	3	1	2	3	1	2	3
any_cancer_type.vcf	36	42	43	122	140	130	158	182	173
breast_adenocarcinoma.vcf	43	46	43	23	20	20	66	66	63
chronic_myeloid_leukemia.vcf	42	50	53	10	9	9	52	59	62
colorectal_adenocarcinoma.vcf	42	42	42	48	48	50	90	90	92
cutaneous_melanoma.vcf	42	42	53	63	48	55	105	90	108
lung_adenocarcinoma.vcf	43	42	42	65	69	73	108	111	115
lung.vcf	42	52	42	22	28	30	64	80	72
non_small_cell_lung.vcf	42	42	52	16	20	25	58	62	77
thyroid_carcinom.vcf	42	52	62	17	17	20	59	69	82
any_cancer_type.vcf + .tsv	381	374	382	248	291	349	629	665	731
breast_adenocarcinoma.vcf + .tsv	283	289	296	63	82	88	346	371	384
colorectal_adenocarcinoma.vcf + .tsv	254	268	137	73	81	81	327	349	218
cutaneous_melanoma.vcf + .tsv	194	166	183	183	221	208	377	387	391
lung_adenocarcinoma.vcf + .tsv	182	183	180	64	110	95	246	293	275
lung.vcf + .tsv	208	163	172	86	133	136	294	296	308
non_small_cell_lung.vcf + .tsv	85	112	82	21	38	33	106	150	115

Appendix E

Supporting Listings

```
version: "3.8"
services:
  sbml4j_api_doc:
    image: swaggerapi/swagger-ui
    container_name: sbml4japidoc
    volumes:
      - sbml4j_api_doc_vol:/definition
    environment:
      - SWAGGER_JSON=/definition/sbml4j.yaml
    ports:
      - "9080:8080"

  sbml4jdatabase:
    image: neo4j:4.1.6
    container_name: sbml4jdb
    volumes:
      - sbml4j_neo4j_vol:/vol
    ports:
      - "7474:7474"
      - "7687:7687"
    environment:
      - NEO4J_CONF=/vol/conf
    restart: unless-stopped
    command: ["neo4j"]
  sbml4j:
    image: thortiede/sbml4j:1.0.5
    container_name: sbml4j
    volumes:
      - sbml4j_service_vol:/logs
    environment:
      - SPRING_PROFILES_ACTIVE=test
      - SPRING_DATA_NEO4J_URI=bolt://sbml4jdb:7687
      - SERVER_SERVLET_CONTEXTPATH=/sbml4j
      - SERVER_PORT=8080
      - SBML4J_GENERAL_API-DOCUMENTATION-URL=http://localhost:9080/

      - SBML4J_context_minSize=1
```

```
38          - SBML4J_context_maxSize=3
39          - SBML4J_context_terminateAt=
40          - SBML4J_context_direction=both
41
42          - SBML4J_OUTPUT_HIDE_MODEL_U-U-I-DS=False
43
44          - SBML4J_NETWORK_HARD-DELETE=True
45          - SBML4J_NETWORK_FORCE-DELETE-OF-PUBLIC-NETWORK=True
46          - SBML4J_NETWORK_DELETE-EXISTING=True
47          - SBML4J_NETWORK_DELETE-DERIVED=True
48          - SBML4J_NETWORK_USE-SHARED-PATHWAY-SEARCH=False
49          - SBML4J_NETWORK_PUBLIC-USER=sbml4j
50          - SBML4J_NETWORK_SHOW-INACTIVE-NETWORKS=False
51          - SBML4J_NETWORK_ALLOW-INACTIVE-DUPLICATES=True
52
53          - SBML4J_ANNOTATION_APPEND=True
54          - SBML4J_ANNOTATION_KEGGGENESSEPARATOR=+
55          - SBML4J_ANNOTATION_ANNOTATE-WITH-LINKS=True
56
57          - SBML4J_csv_matching-column-name[0]=gene name
58          - SBML4J_csv_matching-column-name[1]=genename
59          - SBML4J_csv_matching-column-name[2]=gene.name
60          - SBML4J_csv_matching-column-name[3]=gene_name
61          - SBML4J_csv_matching-column-name[4]=gene symbol
62          - SBML4J_csv_matching-column-name[5]=genesymbol
63          - SBML4J_csv_matching-column-name[6]=gene.symbol
64          - SBML4J_csv_matching-column-name[7]=gene_symbol
65          - SBML4J_csv_matching-column-name[8]=gene
66          - SBML4J_csv_matching-column-name[9]=symbol
67
68          - SBML4J_externalresources_mdanderson_add-md-anderson-annotation=True
69          - SBML4J_externalresources_mdanderson_genelist=ABL1, AKT1, ALK,  \
70              BRAF, CDK4, CDK6, CDKN2A, EGFR, ERBB2, FGFR1, FGFR2, FLT3,  \
71              IDH1, KDR, KIT, KRAS, MDM2, MET, NRAS, NTRK1, NTRK2, PDGFRA, \
72              PIK3CA, PIK3R1, PTCH1, PTEN, PTPN11, PET, ROS1, SMO
73          - SBML4J_externalresources_mdanderson_baseurl= \
74                                                  https://pct.mdanderson.org/home/
75          - SBML4J_externalresources_mdanderson_section=Overview
76          - SBML4J_externalresources_biologicalqualifer_default-database=KEGG
77          - SBML4J_externalresources_keggdatabase_pathway-link-prefix= \
78                                      https://www.genome.jp/kegg-bin/show_pathway?
79          - SBML4J_externalresources_keggdatabase_pathway-search-prefix= \
80              https://www.genome.jp/dbget-bin/www_bfind_sub?dbkey=pathway&keywords=
81      depends_on:
82          - sbml4jdatabase
83          - sbml4j_api_doc
84      ports:
85          - "12342:8080"
86      restart: on-failure
87
88  volumes:
89      sbml4j_service_vol:
90      sbml4j_api_doc_vol:
```

```
91          sbml4j_neo4j_vol:
```

Listing E.1: Docker-compose.yml file for the sbml4j-compose project.

```
1  spring.data.neo4j.uri=bolt://localhost:7687
2  spring.data.neo4j.username=neo4j
3  spring.data.neo4j.password=sbml4j
4  server.servlet.contextPath=/sbml4j
5  server.port=8080
6  server.servlet.session.timeout=-1
7
8  sbml4j.general.api-documentation-url=http://localhost:80
9
10 sbml4j.context.minSize=1
11 sbml4j.context.maxSize=3
12 sbml4j.context.terminateAt=Drugtarget
13 sbml4j.context.direction=both
14
15 sbml4j.network.hard-delete=false
16 sbml4j.network.force-delete-of-public-network=true
17 sbml4j.network.delete-existing=true
18 sbml4j.network.delete-derived=true
19 sbml4j.network.public-user=sbml4j
20 sbml4j.network.show-inactive-networks=false
21 sbml4j.network.allow-inactive-duplicates=true
22
23 sbml4j.annotation.append = true
24 sbml4j.annotation.keggGenesSeparator = +
25 sbml4j.annotation.annotate-with-links=true
26
27 sbml4j.csv.matching-column-name[0]=gene name
28 sbml4j.csv.matching-column-name[1]=genename
29 sbml4j.csv.matching-column-name[2]=gene.name
30 sbml4j.csv.matching-column-name[3]=gene symbol
31 sbml4j.csv.matching-column-name[4]=genesymbol
32 sbml4j.csv.matching-column-name[5]=gene.symbol
33 sbml4j.csv.matching-column-name[6]=gene
34 sbml4j.csv.matching-column-name[7]=symbol
35
36 sbml4j.externalresources.mdanderson.add-md-anderson-annotation=true
37 sbml4j.externalresources.mdanderson.genelist = ABL1, AKT1, ALK, \
38       BRAF, CDK4, CDK6, CDKN2A, EGFR, ERBB2, FGFR1, FGFR2, FLT3, \
39       IDH1, KDR, KIT, KRAS, MDM2, MET, NRAS, NTRK1, NTRK2, PDGFRA, \
40       PIK3CA, PIK3R1, PTCH1, PTEN, PTPN11, PET, ROS1, SMO
41 sbml4j.externalresources.mdanderson.baseurl = https://pct.mdanderson.org/home/
42 sbml4j.externalresources.mdanderson.section = Overview
43
44 sbml4j.externalresources.biologicalqualifer.default-database=KEGG
45 sbml4j.externalresources.keggdatabase.pathway-link-prefix=
46       https://www.genome.jp/kegg-bin/show_pathway?
47 sbml4j.externalresources.keggdatabase.pathway-search-prefix=
48       https://www.genome.jp/dbget-bin/www_bfind_sub?dbkey=pathway&keywords=
```

Listing E.2: Example application.properties file.

```
java -jar KEGGtranslator_v2.5.jar
--input /path/to/kgml
--output /path/to/sbml
--format SBML_CORE_AND_QUAL
--remove-white-gene-nodes TRUE
--autocomplete-reactions TRUE
--gene-names FIRST_NAME
--add-layout-extension FALSE
--use-groups-extension FALSE
--remove-pathway-references TRUE
```

Listing E.3: KEGGtranslator command to translate kgml files to SBML files.

```
{
    "type": "Network",
    "contents": {
        "uuid": "b5f3b7c9-820b-4e8e-8549-4d30eff67127",
        "UUID": "b5f3b7c9-820b-4e8e-8549-4d30eff67127",
        "name": "NOTCH_SIGNALLING_MAPPING",
        "organismCode": "hsa",
        "numberOfNodes": 22,
        "numberOfRelations": 44,
        "numberOfReactions": 0,
        "nodeTypes": [
            "polypeptide chain"
        ],
        "relationTypes": [
            "INHIBITION",
            "STIMULATION",
            "PROTEINCOMPLEXFORMATION"
        ],
        "networkMappingType": "SIGNALLING"
    },
    "wasAttributedTo": {
        "name": "sbml4j",
        "type": "User"
    },
    "wasGeneratedBy": [
        {
            "name": "Create_NOTCH_SIGNALLING_MAPPING",
            "type": "createMapping",
            "operation": "POST",
            "endpoint": "/mapping/14f1cedf-1eb9-4002-9064-361038ea93b7",
            "params": [
                {
                    "parameter": "mappingType",
                    "value": "SIGNALLING"
                },
                {
                    "parameter": "prefixName",
                    "value": "false"
                },
                {
                    "parameter": "suffixName",
```

```
42                      "value": "false"
43                  },
44                  {
45                      "parameter": "networkname",
46                      "value": "NOTCH_SIGNALLING_MAPPING"
47                  },
48                  {
49                      "parameter": "UUID",
50                      "value": "14f1cedf-1eb9-4002-9064-361038ea93b7"
51                  },
52                  {
53                      "parameter": "user",
54                      "value": "sbml4j"
55                  }
56              ]
57          }
58      ],
59      "wasDerivedFrom": [
60          {
61              "type": "Pathway",
62              "contents": {
63                  "uuid": "14f1cedf-1eb9-4002-9064-361038ea93b7",
64                  "UUID": "14f1cedf-1eb9-4002-9064-361038ea93b7",
65                  "name": "Notch signaling pathway",
66                  "pathwayId": "path_hsa04330",
67                  "organismCode": "hsa",
68                  "numberOfNodes": 31,
69                  "numberOfTransitions": 36,
70                  "numberOfReactions": 0,
71                  "nodeTypes": [
72                      "polypeptide chain",
73                      "non-covalent complex",
74                      "simple chemical"
75                  ],
76                  "transitionTypes": [
77                      "inhibition",
78                      "stimulation",
79                      "unknownFromSource",
80                      "control",
81                      "non-covalent binding"
82                  ],
83                  "compartments": [
84                      "default"
85                  ]
86              },
87              "wasAttributedTo": {
88                  "name": "sbml4j",
89                  "type": "User"
90              },
91              "wasGeneratedBy": [
92                  {
93                      "name": "hsa04330.sbml.xml",
94                      "type": "persistFile",
95                      "operation": "POST",
```

```
                    "endpoint": "/sbml",
                    "params": [
                        {
                            "parameter": "organism",
                            "value": "hsa"
                        },
                        {
                            "parameter": "files",
                            "value": "hsa04330.sbml.xml"
                        },
                        {
                            "parameter": "source",
                            "value": "KEGG"
                        },
                        {
                            "parameter": "user",
                            "value": "sbml4j"
                        },
                        {
                            "parameter": "version",
                            "value": "104.0"
                        }
                    ]
                }
            ],
            "wasDerivedFrom": [
                {
                    "provenance": [
                        {
                            "translation": {
                                "name": "KEGGtranslator",
                                "arguments": {
                                    "gene-names": "FIRST_NAME",
                                    "format": "SBML_CORE_AND_QUAL",
                                    "add-layout-extension": "FALSE",
                                    "use-groups-extension": "FALSE",
                                    "remove-pathway-references": "TRUE",
                                    "remove-white-gene-nodes": "TRUE",
                                    "autocomplete-reactions": "TRUE"
                                },
                                "version": "2.5"
                            }
                        },
                        {
                            "origin": {
                                "Download date": "Oct 18, 2022 19:15:42 +0000 (UTC)",
                                "Creation date": "Oct 3, 2022 16:25:37 +0900 (GMT+9)",
                                "filename": "hsa04330.xml",
                                "source": "KEGG Pathways database",
                                "version": "104.0"
                            }
                        },
                        {
                            "SBML4j": {
```

```
                        "version": "1.2.2"
                    }
                }
            ],
            "type": "File",
            "contents": {
                "filename": "hsa04330.sbml.xml",
                "md5sum": "0D19DFBEA5E680B3595A6C31C6B77BAC",
                "fileNodeType": "SBML"
            },
            "wasAttributedTo": {
                "name": "sbml4j",
                "type": "User"
            },
            "wasGeneratedBy": [
                {
                    "name": "hsa04330.sbml.xml",
                    "type": "persistFile",
                    "operation": "POST",
                    "endpoint": "/sbml",
                    "params": [
                        {
                            "parameter": "organism",
                            "value": "hsa"
                        },
                        {
                            "parameter": "files",
                            "value": "hsa04330.sbml.xml"
                        },
                        {
                            "parameter": "source",
                            "value": "KEGG"
                        },
                        {
                            "parameter": "user",
                            "value": "sbml4j"
                        },
                        {
                            "parameter": "version",
                            "value": "104.0"
                        }
                    ]
                }
            ],
            "wasDerivedFrom": [
                {
                    "type": "Database",
                    "contents": {
                        "source": "KEGG",
                        "version": "104.0"
                    },
                    "wasAttributedTo": {
                        "name": "sbml4j",
                        "type": "User"
```

```
                            },
                            "wasGeneratedBy": [
                                {
                                    "name": "hsa04330.sbml.xml",
                                    "type": "persistFile",
                                    "operation": "POST",
                                    "endpoint": "/sbml",
                                    "params": [
                                        {
                                            "parameter": "organism",
                                            "value": "hsa"
                                        },
                                        {
                                            "parameter": "files",
                                            "value": "hsa04330.sbml.xml"
                                        },
                                        {
                                            "parameter": "source",
                                            "value": "KEGG"
                                        },
                                        {
                                            "parameter": "user",
                                            "value": "sbml4j"
                                        },
                                        {
                                            "parameter": "version",
                                            "value": "104.0"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```

Listing E.4: Provenance report for the Notch pathway mapping.

```
import os
from pysbml4j import Sbml4j

client = Sbml4j()

pwuuids=[]
filepath = "/path/to/sbml/"

for filename in os.listdir(filepath):
    try:
        resp = client.uploadSBML(
                [os.path.join(filepath,filename)],
                "hsa", "KEGG", "97.0")
```

```
14          pwuuids.append(resp[0].get("uuid"))
15      except:
16          print(f"Failed to presist model in: {filename}. Response was: {resp}")
17
18  print(pwuuids)
```

Listing E.5: Python code example for uploading SBML files to a locally running SBML4j service using pysbml4j.

```
1   [kegg_api]
2   action= file
3   orgCode = hsa
4   map_ids=
5
6   [container_env]
7   download_folder = /data
```

Listing E.6: Configuration file for the KEGG Pathway Downloader.

```
1   [server]
2   host=host.docker.internal
3   port=12342
4   application_context=/sbml4j
5
6   [data]
7   sbml_dir=/data/sbml
8
9   [source]
10  name=KEGG
11  version=104.0
12  orgCode=hsa
13
14  [origin]
15  path=/data/origin
16  file_suffix=xml.meta
17
18  [transformation]
19  path=/data/transformation
20  steps=keggtranslator.json
21
22
23  [target]
24  collection_name=cancer_types
25  collection_desc=Collection pathway containing all cancer type specific pathways
26
27  mapping_types=SIGNALLING
28  mapping_name_suffix=cancer_types
```

Listing E.7: Configuration file for the SBML4j Initializer.

```
1   {
2     "provenance": [
3       {
4         "DeRegNet": {
```

```
5             "URL": "https://github.com/KohlbacherLab/deregnet",
6             "Release": "Gurobi 9.5.x Release"
7           }
8         }
9       ],
10      "type": "Network",
11      "contents": {
12        "uuid": "f8f8d12b-fbea-4ae6-9bf1-13c2c701a228",
13        "UUID": "f8f8d12b-fbea-4ae6-9bf1-13c2c701a228",
14        "name": "optimal_69a92ea6-7d1e-4cac-a65f-b8afa2380e97",
15        "organismCode": "hsa",
16        "numberOfNodes": 15,
17        "numberOfRelations": 15,
18        "numberOfReactions": 0,
19        "relationTypes": [
20          "STIMULATION"
21        ],
22        "networkMappingType": "SIGNALLING"
23      },
24      "wasAttributedTo": {
25        "name": "deregnet",
26        "type": "User"
27      },
28      "wasGeneratedBy": [
29        {
30          "name": "optimal.graphml",
31          "type": "persistFile",
32          "operation": "POST",
33          "endpoint": "/graphml",
34          "params": [
35            {
36              "parameter": "prefixName",
37              "value": "false"
38            },
39            {
40              "parameter": "suffixName",
41              "value": "false"
42            },
43            {
44              "parameter": "networkname",
45              "value": "optimal_69a92ea6-7d1e-4cac-a65f-b8afa2380e97"
46            },
47            {
48              "parameter": "files",
49              "value": "optimal.graphml"
50            },
51            {
52              "parameter": "UUID",
53              "value": "2366e0e5-5b83-4ad1-8b3b-fcde2a8a494a"
54            },
55            {
56              "parameter": "user",
57              "value": "deregnet"
58            }
```

```
59          ]
60        }
61      ],
62      "wasDerivedFrom": [
63        {
64          "type": "Network",
65          "contents": {
66            "uuid": "2366e0e5-5b83-4ad1-8b3b-fcde2a8a494a",
67            "UUID": "2366e0e5-5b83-4ad1-8b3b-fcde2a8a494a",
68            "name": "DeRegNet_SIG_Base_KEGG_Cancer_Types_chapter4",
69            "organismCode": "hsa",
70            "numberOfNodes": 197,
71            "numberOfRelations": 277,
72            "numberOfReactions": 0,
73            "nodeTypes": [
74              "polypeptide chain"
75            ],
76            "relationTypes": [
77              "STIMULATION",
78              "INHIBITION",
79              "PROTEINCOMPLEXFORMATION"
80            ],
81            "networkMappingType": "SIGNALLING"
82          },
83          "wasAttributedTo": {
84            "name": "deregnet",
85            "type": "User"
86          },
87          "wasGeneratedBy": [
88            {
89              "name": "DeRegNet_Raw_Data_69a92ea6-7d1e-4cac-a65f-b8afa2380e97-filterNetwork
                ↪ ->DeRegNet_SIG_Base_KEGG_Cancer_Types_chapter4",
90              "type": "filterNetwork",
91              "operation": "POST",
92              "endpoint": "/networks/37035f03-51cf-4212-8ea0-96ff3216a65c/filter",
93              "params": [
94                {
95                  "parameter": "prefixName",
96                  "value": "false"
97                },
98                {
99                  "parameter": "networkname",
100                 "value": "DeRegNet_SIG_Base_KEGG_Cancer_Types_chapter4"
101               },
102               {
103                 "parameter": "UUID",
104                 "value": "37035f03-51cf-4212-8ea0-96ff3216a65c"
105               },
106               {
107                 "parameter": "user",
108                 "value": "deregnet"
109               }
110             ],
111             "body": {
```

```
            "nodeTypes": [
              "polypeptide chain"
            ],
            "relationTypes": [
              "STIMULATION",
              "INHIBITION",
              "PROTEINCOMPLEXFORMATION"
            ],
            "nodeSymbols": [
              "SPI1",
              "HHIP",
              ...
            ],
            "relationSymbols": [
              "CHUK-STIMULATION->NFKB1",
              "CREB3L1-INHIBITION->EP300",
              ...
            ]
          }
        }
      ],
      "wasDerivedFrom": [
        {
          "provenance": [
            {
              "69a92ea6-7d1e-4cac-a65f-b8afa2380e97": {
                "download_file_69a92ea6-7d1e-4cac-a65f-b8afa2380e97_url": "https://api.
                ↪ gdc.cancer.gov/data/69a92ea6-7d1e-4cac-a65f-b8afa2380e97",
                "project_cptac3_url": "https://api.gdc.cancer.gov/projects/CPTAC-3?
                ↪ expand=summary,summary.experimental_strategies,summary.
                ↪ data_categories&pretty=false",
                "download_file_69a92ea6-7d1e-4cac-a65f-b8afa2380e97_Content-MD5": "1
                ↪ a94e07b777f946a794be856bc810e9f",
                "project_cptac3": {
                  "data": {
                    "primary_site": [
                      "Pancreas",
                      "Brain",
                      "Other and ill-defined sites",
                      "Kidney",
                      "Bronchus and lung",
                      "Uterus, NOS"
                    ],
                    "dbgap_accession_number": "phs001287",
                    "disease_type": [
                      "Gliomas",
                      "Squamous Cell Neoplasms",
                      "Adenomas and Adenocarcinomas",
                      "Not Applicable",
                      "Ductal and Lobular Neoplasms"
                    ],
                    "project_id": "CPTAC-3",
                    "name": "CPTAC-Brain, Head and Neck, Kidney, Lung, Pancreas, Uterus
                    ↪ ",
```

```
                    "releasable": true,
                    "state": "open",
                    "released": true
                  },
                  "warnings": {}
                },
                "download_file_69a92ea6-7d1e-4cac-a65f-b8afa2380e97_filename": "seurat.deg.tsv",
                "download_file_69a92ea6-7d1e-4cac-a65f-b8afa2380e97_Date": "Fri, 14 Oct 2022 18:08:46 GMT",
                "file_uuids": [
                  "69a92ea6-7d1e-4cac-a65f-b8afa2380e97"
                ],
                "download_file_69a92ea6-7d1e-4cac-a65f-b8afa2380e97_Content-Length": "320113",
                "status_url": "https://api.gdc.cancer.gov/status",
                "status": {
                  "data_release": "Data Release 35.0 - September 28, 2022",
                  "commit": "4dd3680528a19ed33cfc83c7d049426c97bb903b",
                  "tag": "3.0.0",
                  "version": 1,
                  "status": "OK"
                }
              }
            },
            {
              "data_preprocessing": {
                "input_file": "data/tcga/69a92ea6-7d1e-4cac-a65f-b8afa2380e97.tsv",
                "score_column": "avg_log2FC",
                "match_id": "name",
                "version": 1.0,
                "script": "TCGA_Data_Preprocessing.py"
              }
            }
          ],
          "type": "Network",
          "contents": {
            "uuid": "37035f03-51cf-4212-8ea0-96ff3216a65c",
            "UUID": "37035f03-51cf-4212-8ea0-96ff3216a65c",
            "name": "DeRegNet_Raw_Data_69a92ea6-7d1e-4cac-a65f-b8afa2380e97",
            "organismCode": "hsa",
            "numberOfNodes": 197,
            "numberOfRelations": 279,
            "numberOfReactions": 0,
            "nodeTypes": [
              "polypeptide chain"
            ],
            "relationTypes": [
              "STIMULATION",
              "INHIBITION",
              "PROTEINCOMPLEXFORMATION"
            ],
            "networkMappingType": "SIGNALLING"
          },
```

```
212             "wasAttributedTo": {
213               "name": "deregnet",
214               "type": "User"
215             },
216             "wasGeneratedBy": [
217               {
218                 "name": "SIGNALLING_cancer_types—addCsvAnnotation—>
                    ↪ DeRegNet_Raw_Data_69a92ea6—7d1e—4cac—a65f—b8afa2380e97",
219                 "type": "addCsvAnnotation",
220                 "operation": "POST",
221                 "endpoint": "/networks/6194654b—f0ef—465e—b81a—9d32d5bd9f49/csv",
222                 "params": [
223                   {
224                     "parameter": "prefixName",
225                     "value": "false"
226                   },
227                   {
228                     "parameter": "networkname",
229                     "value": "DeRegNet_Raw_Data_69a92ea6—7d1e—4cac—a65f—b8afa2380e97"
230                   },
231                   {
232                     "parameter": "UUID",
233                     "value": "6194654b—f0ef—465e—b81a—9d32d5bd9f49"
234                   },
235                   {
236                     "parameter": "type",
237                     "value": "69a92ea6—7d1e—4cac—a65f—b8afa2380e97"
238                   },
239                   {
240                     "parameter": "user",
241                     "value": "deregnet"
242                   },
243                   {
244                     "parameter": "derive",
245                     "value": "true"
246                   }
247                 ]
248               }
249             ],
250             "wasDerivedFrom": [
251               {
252                 "type": "Network",
253                 "contents": {
254                   "uuid": "6194654b—f0ef—465e—b81a—9d32d5bd9f49",
255                   "UUID": "6194654b—f0ef—465e—b81a—9d32d5bd9f49",
256                   "name": "SIGNALLING_cancer_types",
257                   "organismCode": "hsa",
258                   "numberOfNodes": 197,
259                   "numberOfRelations": 279,
260                   "numberOfReactions": 0,
261                   "nodeTypes": [
262                     "polypeptide chain"
263                   ],
264                   "relationTypes": [
```

```
265                     "STIMULATION",
266                     "INHIBITION",
267                     "PROTEINCOMPLEXFORMATION"
268                   ],
269                   "networkMappingType": "SIGNALLING"
270                 },
271                 "wasAttributedTo": {
272                   "name": "sbml4j",
273                   "type": "User"
274                 },
275                 "wasGeneratedBy": [
276                   {
277                     "name": "Create_SIGNALLING_cancer_types",
278                     "type": "createMapping",
279                     "operation": "POST",
280                     "endpoint": "/mapping/fa911939-6aff-4fde-a7fd-c1e2705cbe77",
281                     "params": [
282                       {
283                         "parameter": "mappingType",
284                         "value": "SIGNALLING"
285                       },
286                       {
287                         "parameter": "prefixName",
288                         "value": "false"
289                       },
290                       {
291                         "parameter": "suffixName",
292                         "value": "false"
293                       },
294                       {
295                         "parameter": "networkname",
296                         "value": "SIGNALLING_cancer_types"
297                       },
298                       {
299                         "parameter": "UUID",
300                         "value": "fa911939-6aff-4fde-a7fd-c1e2705cbe77"
301                       },
302                       {
303                         "parameter": "user",
304                         "value": "sbml4j"
305                       }
306                     ]
307                   }
308                 ],
309                 "wasDerivedFrom": [
310                   {
311                     "type": "Pathway",
312                     "contents": {
313                       "uuid": "fa911939-6aff-4fde-a7fd-c1e2705cbe77",
314                       "UUID": "fa911939-6aff-4fde-a7fd-c1e2705cbe77",
315                       "name": "Collection pathway containing all cancer type specific
                          ↪ pathways",
316                       "pathwayId": "cancer_types",
317                       "organismCode": "hsa",
```

```
318                   "numberOfNodes": 836,
319                   "numberOfTransitions": 1416,
320                   "numberOfReactions": 0,
321                   "nodeTypes": [
322                     "polypeptide chain",
323                     "non-covalent complex",
324                     "simple chemical"
325                   ],
326                   "transitionTypes": [
327                     "stimulation",
328                     "uncertain process",
329                     "inhibition",
330                     "phosphorylation",
331                     "non-covalent binding",
332                     "molecular interaction",
333                     "dissociation",
334                     "unknownFromSource",
335                     "dephosphorylation"
336                   ],
337                   "compartments": [
338                     "default"
339                   ]
340                 },
341                 "wasAttributedTo": {
342                   "name": "sbml4j",
343                   "type": "User"
344                 },
345                 "wasGeneratedBy": [
346                   {
347                     "name": "Create_PathwayCollection_for_3734df16_ab97bd7e_7524c678
                      ↪ _6a232d3a_cb1199e6_6181fa25_df64e8b3_70224b7e_0253ffac
                      ↪ _1fb39670_1e2e174c_34b46434_95c13a9f_ddca2875_fd1fe32a
                      ↪ _66e3062c_317194b7",
348                     "type": "createPathwayCollection",
349                     "operation": "POST",
350                     "endpoint": "/pathwayCollection",
351                     "body": {
352                       "name": "cancer_types",
353                       "description": "Collection pathway containing all cancer type
                        ↪ specific pathways",
354                       "sourcePathwayUUIDs": [
355                         "3734df16-7138-45a1-8b33-35adf2c5bd54",
356                         "ab97bd7e-a1b1-421a-bb8f-034b97dc3247",
357                         "7524c678-75cc-428d-a12c-f041b4a88339",
358                         "6a232d3a-9522-4101-80c2-5a9eb09b2d63",
359                         "cb1199e6-7573-472d-91b3-59fe0087a761",
360                         "6181fa25-d804-42d5-8397-49fed001e975",
361                         "df64e8b3-a1f6-4a31-a570-c1dc0f417a29",
362                         "70224b7e-9c5e-4d15-a9ea-4add945ce737",
363                         "0253ffac-463b-4dbb-9624-08a9f05c0be5",
364                         "1fb39670-df20-4676-9c33-71d12f12711a",
365                         "1e2e174c-cfbc-4abc-bec6-033f5c09fb7c",
366                         "34b46434-cdf9-4e7e-ae0b-97ca0cc85945",
367                         "95c13a9f-f450-4e5f-90e1-6157b2dcf3b6",
```

```
368                         "ddca2875-fc34-4744-82f9-e4830e9a09a3",
369                         "fd1fe32a-e901-4fba-ac84-275243ac6ea9",
370                         "66e3062c-eb79-4fae-b79a-908d63bbf554",
371                         "317194b7-ef3f-4a76-8f7e-ee487c99b811"
372                       ]
373                     }
374                   }
375                 ],
376                 "wasDerivedFrom": [
377                   {
378                     "type": "PathwayCollection",
379                     "contents": {
380                       "name": "PCN_of_cancer_types",
381                       "description": "PathwayCollectionNode of collection: Collection
                          ↪ pathway containing all cancer type specific pathways",
382                       "hadMember": [
383                         "317194b7-ef3f-4a76-8f7e-ee487c99b811",
384                         "66e3062c-eb79-4fae-b79a-908d63bbf554",
385                         "fd1fe32a-e901-4fba-ac84-275243ac6ea9",
386                         "ddca2875-fc34-4744-82f9-e4830e9a09a3",
387                         "95c13a9f-f450-4e5f-90e1-6157b2dcf3b6",
388                         "34b46434-cdf9-4e7e-ae0b-97ca0cc85945",
389                         "1e2e174c-cfbc-4abc-bec6-033f5c09fb7c",
390                         "1fb39670-df20-4676-9c33-71d12f12711a",
391                         "0253ffac-463b-4dbb-9624-08a9f05c0be5",
392                         "70224b7e-9c5e-4d15-a9ea-4add945ce737",
393                         "df64e8b3-a1f6-4a31-a570-c1dc0f417a29",
394                         "6181fa25-d804-42d5-8397-49fed001e975",
395                         "cb1199e6-7573-472d-91b3-59fe0087a761",
396                         "6a232d3a-9522-4101-80c2-5a9eb09b2d63",
397                         "7524c678-75cc-428d-a12c-f041b4a88339",
398                         "ab97bd7e-a1b1-421a-bb8f-034b97dc3247",
399                         "3734df16-7138-45a1-8b33-35adf2c5bd54"
400                       ]
401                     },
402                     "wasAttributedTo": {
403                       "name": "sbml4j",
404                       "type": "User"
405                     },
406                     "wasGeneratedBy": [
407                       {
408                         "name": "
                            ↪ Create_PathwayCollection_for_3734df16_ab97bd7e_7524c678
                            ↪ _6a232d3a_cb1199e6_6181fa25_df64e8b3_70224b7e_0253ffac
                            ↪ _1fb39670_1e2e174c_34b46434_95c13a9f_ddca2875_fd1fe32a
                            ↪ _66e3062c_317194b7",
409                         "type": "createPathwayCollection",
410                         "operation": "POST",
411                         "endpoint": "/pathwayCollection",
412                         "body": {
413                           "name": "cancer_types",
414                           "description": "Collection pathway containing all cancer
                              ↪ type specific pathways",
415                           "sourcePathwayUUIDs": [
```

```
                "3734df16-7138-45a1-8b33-35adf2c5bd54",
                "ab97bd7e-a1b1-421a-bb8f-034b97dc3247",
                "7524c678-75cc-428d-a12c-f041b4a88339",
                "6a232d3a-9522-4101-80c2-5a9eb09b2d63",
                "cb1199e6-7573-472d-91b3-59fe0087a761",
                "6181fa25-d804-42d5-8397-49fed001e975",
                "df64e8b3-a1f6-4a31-a570-c1dc0f417a29",
                "70224b7e-9c5e-4d15-a9ea-4add945ce737",
                "0253ffac-463b-4dbb-9624-08a9f05c0be5",
                "1fb39670-df20-4676-9c33-71d12f12711a",
                "1e2e174c-cfbc-4abc-bec6-033f5c09fb7c",
                "34b46434-cdf9-4e7e-ae0b-97ca0cc85945",
                "95c13a9f-f450-4e5f-90e1-6157b2dcf3b6",
                "ddca2875-fc34-4744-82f9-e4830e9a09a3",
                "fd1fe32a-e901-4fba-ac84-275243ac6ea9",
                "66e3062c-eb79-4fae-b79a-908d63bbf554",
                "317194b7-ef3f-4a76-8f7e-ee487c99b811"
              ]
            }
          }
        ],
        "wasDerivedFrom": [
          {
            "type": "Pathway",
            "contents": {
              "uuid": "317194b7-ef3f-4a76-8f7e-ee487c99b811",
              "UUID": "317194b7-ef3f-4a76-8f7e-ee487c99b811",
              "name": "Gastric cancer",
              "pathwayId": "path_hsa05226",
              "organismCode": "hsa",
              "numberOfNodes": 77,
              "numberOfTransitions": 124,
              "numberOfReactions": 0,
              "nodeTypes": [
                "polypeptide chain",
                "simple chemical",
                "non-covalent complex"
              ],
              "transitionTypes": [
                "stimulation",
                "uncertain process",
                "inhibition",
                "non-covalent binding",
                "phosphorylation",
                "dissociation",
                "molecular interaction"
              ],
              "compartments": [
                "default"
              ]
            },
            "wasAttributedTo": {
              "name": "sbml4j",
              "type": "User"
```

```
470                         },
471                         "wasGeneratedBy": [
472                           {
473                             "name": "hsa05226.sbml.xml",
474                             "type": "persistFile",
475                             "operation": "POST",
476                             "endpoint": "/sbml",
477                             "params": [
478                               {
479                                 "parameter": "organism",
480                                 "value": "hsa"
481                               },
482                               {
483                                 "parameter": "files",
484                                 "value": "hsa05226.sbml.xml"
485                               },
486                               {
487                                 "parameter": "source",
488                                 "value": "KEGG"
489                               },
490                               {
491                                 "parameter": "user",
492                                 "value": "sbml4j"
493                               },
494                               {
495                                 "parameter": "version",
496                                 "value": "104.0"
497                               }
498                             ]
499                           }
500                         ],
501                         "wasDerivedFrom": [
502                           {
503                             "provenance": [
504                               {
505                                 "origin": {
506                                   "Download date": "Oct 10, 2022 18:55:46 +0000 (UTC)
                                    ↪ ",
507                                   "Creation date": "Apr 11, 2018 17:06:20 +0900 (GMT
                                    ↪ +9)",
508                                   "Original Filename": "hsa05226.xml"
509                                 }
510                               },
511                               {
512                                 "transformation": {
513                                   "keggtranslator.json": {
514                                     "name": "KEGGtranslator",
515                                     "arguments": {
516                                       "gene-names": "FIRST_NAME",
517                                       "format": "SBML_CORE_AND_QUAL",
518                                       "add-layout-extension": "FALSE",
519                                       "use-groups-extension": "FALSE",
520                                       "remove-pathway-references": "TRUE",
521                                       "remove-white-gene-nodes": "TRUE",
```

```
                "autocomplete-reactions": "TRUE"
              },
              "version": "2.5"
            }
          }
        }
      ],
      "type": "File",
      "contents": {
        "filename": "hsa05226.sbml.xml",
        "md5sum": "F4D464044705FAA389DD303C5885D771",
        "fileNodeType": "SBML"
      },
      "wasAttributedTo": {
        "name": "sbml4j",
        "type": "User"
      },
      "wasGeneratedBy": [
        {
          "name": "hsa05226.sbml.xml",
          "type": "persistFile",
          "operation": "POST",
          "endpoint": "/sbml",
          "params": [
            {
              "parameter": "organism",
              "value": "hsa"
            },
            {
              "parameter": "files",
              "value": "hsa05226.sbml.xml"
            },
            {
              "parameter": "source",
              "value": "KEGG"
            },
            {
              "parameter": "user",
              "value": "sbml4j"
            },
            {
              "parameter": "version",
              "value": "104.0"
            }
          ]
        }
      ],
      "wasDerivedFrom": [
        {
          "type": "Database",
          "contents": {
            "source": "KEGG",
            "version": "104.0"
          },
```

```
576                                     "wasAttributedTo": {
577                                       "name": "sbml4j",
578                                       "type": "User"
579                                     },
580                                     "wasGeneratedBy": [
581                                       {
582                                         "name": "hsa05210.sbml.xml",
583                                         "type": "persistFile",
584                                         "operation": "POST",
585                                         "endpoint": "/sbml",
586                                         "params": [
587                                           {
588                                             "parameter": "organism",
589                                             "value": "hsa"
590                                           },
591                                           {
592                                             "parameter": "files",
593                                             "value": "hsa05210.sbml.xml"
594                                           },
595                                           {
596                                             "parameter": "source",
597                                             "value": "KEGG"
598                                           },
599                                           {
600                                             "parameter": "user",
601                                             "value": "sbml4j"
602                                           },
603                                           {
604                                             "parameter": "version",
605                                             "value": "104.0"
606                                           }
607                                         ]
608                                       }
609                                     ]
610                                   }
611                                 ]
612                               }
613                             ]
614                           },
615                           ...
616                         ]
617                       }
618                     ]
619                   }
620                 ]
621               }
622             ]
623           }
624         ]
625       }
626     ]
```

```
627 }
```

Listing E.8: Provenance-report of the optimal solution network created by DeRegNet that is created in SBML4j in Chapter 4.

```
# create a csv containing one entry per drug targeting a human gene

# load genes with associated drugs
genes <- read.csv("all.csv", header=TRUE, stringsAsFactors = FALSE)
genes <- genes[order(genes$Name),]

# load dictionary for drugbank id and drug name
drugbank_vocabulary <- read.csv("drugbank_vocabulary.csv", header=TRUE,
    stringsAsFactors = FALSE)

# create new dataframe
filtered_genes <- data.frame(matrix(ncol = 2, nrow = 0))
colnames(filtered_genes) <- c("Gene.Name", "Drug.IDs")

i <- 1
j <- 2

drugids <- genes[i,"Drug.IDs"]

# filter for human genes and only have one entry per gene
while(j<=nrow(genes) && i <= (nrow(genes)-1)) {
    gene1 <- genes[i,"Gene.Name"]
    gene2 <- genes[j,"Gene.Name"]
    species1 <- genes[i,"Species"]
    species2 <- genes[j,"Species"]
    if( gene1 == gene2 && species1 == "Humans" &&
        species2 == "Humans"){
        drugids <- paste(drugids, as.character(genes[j,"Drug.IDs"]),
            sep=";")
        j <- j+1
    }
    else{
        if(species1 == "Humans" && (is.na(genes[i,"Gene.Name"]) |
            nchar(genes[i,"Gene.Name"]) > 0)){
            filtered_genes[nrow(filtered_genes) + 1,] =
            list(genes[i,"Gene.Name"],drugids)
        }
        i <- j
        j <- j+1

        drugids <- genes[i,"Drug.IDs"]
    }
}

# merge drug common names and drug infos with associated genes
drugcsv <- data.frame(matrix(ncol = 8, nrow = 0))
colnames(drugcsv) <- c(colnames(drugbank_vocabulary), "Gene.Name")
for(row in 1:nrow(filtered_genes)){
    gene <- filtered_genes[row, "Gene.Name"]
    drugids <- unlist(strsplit(filtered_genes[row, "Drug.IDs"], ";"))
    for(drug in drugids){
        drug <- gsub("\\s", "", drug)
        drugcsv[nrow(drugcsv) + 1,] = c(
            drugbank_vocabulary[drugbank_vocabulary$DrugBank.ID==drug,],
            gene)
    }
}

# save new csv
write.csv(drugcsv,"drug_genes_approved.csv", row.names = FALSE)
```

Listing E.9: R script to preprocess the drug-target information from DrugBank.