technische universität
dortmund

Projektgruppe 558

Eclipse4Bio

Endbericht

März 2012

Baris Ayaz
Benjamin Nagel
Christopher Schröder
Dawid Kopetzki
Lidiya Kaltchev
Lukas Lerche
Manuel Allhoff
Martin Kozianka
Mehmet Ali Yaman
Mevlüt Cetin
Youssef Zaki

Veranstalter:
Anna-Lena Lamprecht
Stefan Naujokat

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Programmiersysteme (LS 5)
http://ls5-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1. Einleitung

Im Rahmen der Projektgruppe 558 der TU Dortmund wurde eine Eclipse-basierte Open-Source Plattform für Workflows in der Bioinformatik entwickelt. Ein Workflow stellt die Automatisierung eines Prozesses dar, bei dem Dokumente, Informationen oder Aufgaben zwischen den Aktivitäten, aus denen der Prozess aufgebaut ist, ausgetauscht werden. Dieser Datenfluss wird mit einer Menge von Regeln gesteuert, was unter dem Begriff Kontrollfluss zusammengefasst wird.

Ein Prozess ist eine Menge von manuellen, teil-automatisierten oder voll automatisierten Aktivitäten, die nach bestimmten Regeln ausgeführt werden. Die Prozesse hängen bezüglich ihrer betroffenen Ressourcen, wie z.B. Personen, Maschinen oder Dokumente zusammen. Die Ausführung der Prozesse wird entweder von Personen oder Maschinen initiiert. Ein Workflow setzt sich also aus Aktivitäten zusammen, die entweder von Anwendungsprogrammen ausgeführt werden oder ein menschliches Eingreifen erfordern.

Eclipse4Bio ist ein Workflow-Management- und -Execution-Tool für wissenschaftliche Workflows, speziell im Bereich der Bioinformatik. Es ist in der Programmierspache Java implementiert und basiert auf dem Entwicklungsframework *Eclipse*. Es bietet dem Anwender die Möglichkeit, die Bestandteile eines Workflows in einer speziellen Syntax textuell oder mittels einer graphischen Eingabe zu definieren. Anschließend kann der Workflow durch die Eclipse4Bio eigene Engine ausgeführt werden.

Ursprünglich war die Entwicklung von Eclipse4Bio darauf ausgerichtet, Workflows, die speziell auf die Verbindung mit Webservices spezialisiert sind, erstellen zu können. Hierbei stand insbesondere BioCatalogue im Vordergrund, ein Online-Katalog für frei angebotene Webservices aus dem Bereich der Bioinformatik. Im Laufe der Zeit wurde durch die Hilfe der Bioinformatiker des Lehrstuhl 11 der TU Dortmund deutlich, dass aus ihrer Sicht an einem solchen System wenig Bedarf besteht. Die Größe der zu verarbeitenden Daten von oftmals mehreren Gigabyte ist in der heutigen Zeit zu groß für ein Versenden an entsprechende Webservices. Auch die Verarbeitung dieser Daten kann mit der derzeit zur Verfügung stehenden Rechenkapazität mehrere Tage dauern. Diese Kapazitäten werden nicht frei zur Verfügung gestellt, somit sind die Eingabegrößen für die Webservices in der Regel limitiert. Durch diese Einschränkungen besteht stattdessen Interesse an einem System, das die Eingaben lokal oder über einen Rechner im lokalen Netz berechnet. Die Einsatzgebiete von Eclipse4Bio sind somit nicht mehr Workflows, die speziell auf eine einfache Implementierung und Kommunikation mit im Internet frei angebotenen Webservices

auslegt sind. Eclipse4Bio stellt ein lokales System dar, das das Spezifizieren, Ausführen, Beobachten und Steuern eines Workflows ermöglicht. Die Basis dieses Systems bildet die Eclipse-Plattform. Sie bietet den Vorteil vieler etablierter Frameworks und Plugins, eines einheitlichen Designs und einer einfachen Bedienung.

1.1. Aufbau des Dokuments

Dieser Bericht beschreibt das von der Projektgruppe entwickelte Eclipse4Bio. Als Motivation für dieses Workflowmanagementsystem wird in diesem Kapitel 1 zunächst in die Thematik der Bioinformatik eingeführt. Es werden Anwendungsfelder in der Bioinformatik vorgestellt, in denen der Einsatz von Workflows sinnvoll ist. Dazu zählt insbesondere die Analyse von Daten, die mit Hochdurchsatztechnologien erstellt worden sind. Exemplarisch werden dazu zwei Workflows mit einem realen Hintergrund beschrieben. Zudem werden bereits existierende Managementsysteme vorgestellt und die besonderen Anforderungen, die an ein Managementsystem in der Bioinformatik gestellt werden, dargelegt.

Anschließend wird in Kapitel 2 und in den folgenden Kapiteln die Modellierung der Workflows für Eclipse4Bio erläutert. Es wird in das Konzept der Metamodellierung eingeführt und erklärt, wie eine textuelle und eine grafische Repräsentation eines Workflows realisiert werden kann. Die nötigen Werkzeuge und Technologien werden genannt und eine Einführung in die Beschreibungssprache der Workflows gegeben, sowie erläutert, wie die Validierung der Benutzereingaben funktioniert.

Im Anschluss an die Beschreibung des Eclipse4Bio-Meta-Modells wird ab Kapitel 4 darauf eingegangen, wie ein Workflow grafisch modelliert werden kann. Es werden verschiedene Technologien zur Erzeugung von grafischen Editoren, wie zum Beispiel EuGENia und Graphiti, beschrieben und jeweils erläutert, wieso Eclipse4Bio sie bei der Realisierung eines Projekts verwendet oder nicht verwendet. Die Benutzeroberfläche und ihre Komponenten werden beschrieben, sodass deutlich wird, wie ein Workflow grafisch modelliert werden kann.

Nach der Modellierung eines Workflows besteht die Möglichkeit, diesen auszuführen. Ab Kapitel 7 wird beschreiben, wie dies realisiert wird. Dabei werden die besonderen Anforderungen bei der Ausführung eines Workflows aufgezeigt. Das Konzept des zur Ausführung verwendeten Frameworks SpringBatch wird dargestellt. Schließlich werden weitere notwendige Technologien eingeführt, sowie deren Anwendung bei konkreten Workflows in Eclipse4Bio.

1.2. Workflowanwendungen in der Bioinformatik

An dieser Stelle soll verdeutlicht werden, in welchen Bereichen die Anwendung von Workflows in der Bioinformatik sinnvoll ist. Dabei wird klar, dass die Entwicklung von neuen

Hochdurchsatztechnologien ein wichtiges Argument für die Benutzung von Workflows darstellt. Mit Hochdurchsatzverfahren ist es möglich, große Mengen an biologischen Daten (z.B. Genome als DNA-String) kostengünstig in Laboren zu produzieren, um sie anschließend zu analysieren.

Die enormen Datenmengen machen ein manuelles Bearbeiten unmöglich. Zudem sind die Analyseprozesse sehr komplex, sodass eine Reproduzierbarkeit der Ergebnisse keine triviale Aufgabe darstellt. Insbesondere die vielen unterschiedlichen Datentypen, mit denen gearbeitet werden muss, stellen ein großes Hindernis bei der korrekten Analyse dar. Es existieren viele Werkzeuge, die Dienste anbieten, welche typischerweise bei so einer Analyse benötigt werden.

Durch Workflowmanagementsysteme wird versucht, den neuen Herausforderungen gerecht zu werden und ein einfach zu bedienendes und effizientes Analysewerkzeug bereitzustellen. Es haben sich im Laufe der Zeit wichtige Anforderungen an ein Workflowmanagementsystem etabliert. Zum einen soll es möglich sein, mehrere Datenquellen abzufragen und die Ergebnisse vereinen zu können. Zum anderen sollen diese Datenbankabfragen mit verschiedenen Bioinformatikanwendungen zu Analysen verknüpft werden können. Außerdem existieren weitere Anforderungen, die im folgenden Abschnitt über existierende Workflowsysteme aufgegriffen und erläutert werden.

Ein wichtiges Anwendungsgebiet von Workflows ist die Analyse von Daten, die mit Hochdurchsatztechnologien erzeugt wurden. Dabei werden z.B. Genome sequenziert. Diese Genome können in einem anschließenden Schritt analysiert werden. Es existieren viele verschiedene Datenbanken mit Informationen zu den Genomen (z.B. die Position der Gene auf dem Genom). Es muss mit einem Workflowsystem auf einfache Weise möglich sein, diese Informationen in die Analyse einzubauen und die Daten mit weiteren Bioinformatikwerkzeugen zu analysieren.

Es existieren bereits Workflowsysteme für die Bioinformatik, die in diesem Kapitel exemplarisch Anhand von *Taverna* [43] und *DiscoveryNet* [1] erläutert werden. Dabei wird auf ihre praktische Anwendung sowie auf die benutzten Workflowsprachen eingegangen.

Zudem werden zwei konkrete Beispiele für Workflows gegeben. Es soll gezeigt werden, wie ein Workflow mit Daten, die durch Hochdurchsatzverfahren gewonnen wurden, aussehen kann. Die biologischen Grundlagen, die nötig sind, um diese Beispielworkflows zu verstehen, werden eingeführt.

1.3. Workflowsysteme

Es existieren bereits viele Workflowsysteme, die sich auf die besonderen Anforderungen der Bioinformatik spezialisiert haben. An dieser Stelle werden exemplarisch die Systeme *Taverna* und *DiscoveryNet* beschrieben.

Wichtige Anforderungen sind unter anderem:

- **Hoher Durchsatz.** In der Bioinformatik muss ein hoher Durchsatz von Daten möglich sein. Neue Technologien haben dazu geführt, dass extrem große Datenmengen sehr billig produziert werden können. Diese Daten müssen ausgewertet werden. Workflows sollen dem Benutzer helfen, diese Analysen durchzuführen. Deshalb müssen die Workflowsysteme einen hohen Datendurchsatz unterstützen.

- **Ungewöhnliche und komplexe Datentypen.** Die Bioinformatik ist einem schnellen Wandel unterworfen. Es werden schnell neue Technologien entwickelt aus denen neue Datentypen hervorgehen mit denen gearbeitet werden muss. Zudem muss die Bioinformatik mit vielen verschiedenen Daten wie z.B. Alignments, DNA-Sequenzen oder sogenannten Tracks (Listen von besonderen Stellen im Genom) arbeiten. Es muss im Workflow möglich sein, mit vielen verschiedenen Datentypen zu arbeiten.

- **Bedienbarkeit.** Die Analysen von biologischen Daten können sehr komplex werden. Wenn sie mit Workflows durchgeführt werden sollen, muss sichergestellt werden, dass das Workflowsystem einfach und intuitiv bedienbar ist.

1.3.1. Taverna

Taverna ist ein Open-Source-Workflowmanagementsystem [43]. Es hat sich auf die Verwendung von *Webservices* im Workflow spezialisiert. Die Verwendung von Webservices bietet einige Vorteile:

- Die Anwendungsprogramme und Datenbanken müssen nicht lokal installiert werden. Der Webservice kann ohne aufwändige Installation benutzt werden.

- Die Anwendungsprogramme können in verschiedenen Sprachen für verschiedene Betriebssysteme implementiert sein. Der Zugriff auf diese Programme durch den Webservice versteckt diese technische Ebene vor dem Endbenutzer, sodass er das Programm leichter verwenden kann.

- Es ist relativ einfach, eine komplexe Pipeline von Anwendungsprogrammen aufzubauen. Dieser Aufbau geschieht auf einer sehr hohen Ebene, da sich der Benutzer nicht mit Installationsproblemen beschäftigen muss.

Ihre Verwendung hat allerdings auch einige Nachteile:

- Wenn Webservices zu einer Pipeline zusammenschaltet werden, ist nicht sichergestellt, dass die Eingabe- und Ausgabedaten jeweils das passende Format haben. Hier müssen gegebenenfalls selbst geschriebene Programme die Daten zunächst konvertieren.

- Webservices können eine schlecht vorhersagbare Laufzeit aufweisen. Die Laufzeit hängt nicht nur von den verwendeten Algorithmen ab, sondern auch von der Leistungsfähigkeit der Internetverbindung und des Servers, auf dem der Webservice installiert ist.

- Viele Operationen sind sehr rechen- und zeitintensiv. Sehr wahrscheinlich werden für solche Operationen keine Webserviceanbieter ihre Rechenkraft kostenlos zur Verfügung stellen.

- Die Berechnungen brauchen unter Umständen sehr große Datenmengen. Diese Datenmengen müssen zum Server, auf dem der Webservice läuft, transferiert werden. Das kann sehr zeitaufwändig sein.

Taverna benutzt *SCUFL* [3] um Workflows zu beschreiben. SCUFL steht für *Simple Conceptual Unified Flow Language*. Diese Sprache repräsentiert Workflows als gerichtete, azyklische Graphen. Die Knoten im Graphen heißen *Processors* und stellen die Anwendungsprogramme dar. Ein Processor kann abstrakt als Funktion angesehen werden, die Eingabedaten in Ausgabedaten transformiert. Diese Daten liegen bei Processors an sogenannten *Ports* an. Eingabeparameter sind als Quellen und Ausgabewerte als Senken definiert. Es existieren zwei verschiedene Kantentypen zwischen den Knoten. Die *Data Links* beschreiben den Datenfluss innerhalb des Workflows. Die sogenannten *Coordination Constraints* koordinieren die Ausführungsreihenfolge unabhängig vom Datenfluss.

Die Ausführung eines SCUFL-Workflows kann als Durchlauf durch den Graphen angesehen werden. Es wird an der Quelle gestartet und ein Weg zur Senke gesucht. Sind alle Senken erreicht, ist der Workflow abgearbeitet worden.

Abbildung 1.1 zeigt die GUI von Taverna. Dabei ist ein sehr einfacher Workflow abgebildet, der als Graph dargestellt ist. Eingabedaten sind als Quelle und Ausgabedaten als Senke beschrieben. Dieser Workflow konkateniert Strings, die Aminosäuren im Einbuchstabencode beschreiben. Ein Protein kann als Sequenz von Aminosäuren angesehen werden. Der String, der am Ende ausgegeben wird, könnte also ein Protein beschreiben. Als Eingabedatum ist hier der String M gewählt worden. Die Aminosäuresequenz eines Proteins startet immer mit der Aminosäure Methionin (M). Im Knoten *Concat* wird das M zunächst mit einem A konkateniert. Anschließend werden die anderen Knoten des Workflows durchlaufen, wobei an den String noch FP, WL und QE angehängt wird. Dadurch ergibt sich als Ausgabedatum der String $MAFPWLQE$, der ein Protein beschreiben könnte. Statt der einfachen Konkatenierungsfunktion können auch Webservices benutzt werden, um so eine komplexere Pipeline aufzubauen.

Der Kontrollfluss orientiert sich an dem Datenflussmodell, das durch die Data Links beschrieben wird. Ein Processor hat, sofern er mit einem Data Link verbunden ist, direkten Einfluss auf den Kontrollfluss. Es gibt aber Situationen, in denen ein Processor Einfluss

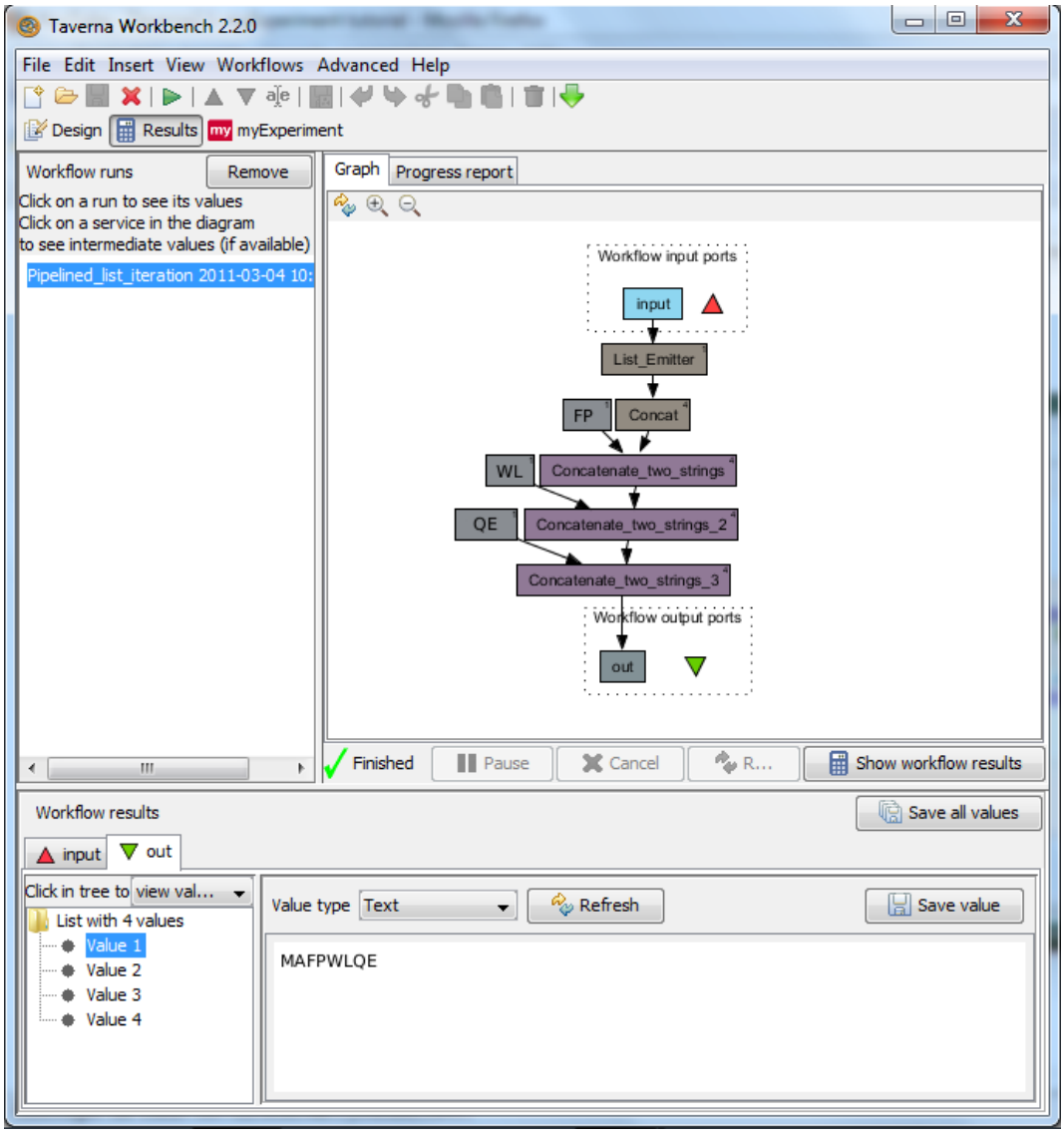

Abbildung 1.1.: Taverna 2.2.0. Das Programm Taverna zeigt den SCUFL-Graphen, der einen Workflow beschreibt. Es existiert ein Weg vom Eingabedatum (Quelle) zum Ausgabedatum (Senke). Die einzelnen Knoten, die sogenannten Processors, stellen Aktionen dar. In diesem Fall wird der Eingabestring mit anderen Strings konkateniert. Es ist denkbar, komplexere Processors zu verwenden, um eine Pipeline zu erstellen, mit der biologische Daten analysiert werden können. Das Ausgabedatum in diesem Beispiel lautet *MAFPWLQE*, wobei *M* zu Anfang des Strings der Startwert war. Dieser String stellt eine Aminosäuresequenz im Einbuchstabencode dar. Proteine kann man als Sequenz von Aminosäuren beschreiben.

nehmen muss, obwohl er keine direkte Verbindung zum Datenfluss hat. Als Beispiel sei ein Workflow gegeben, bei dem eine Aktivität darin besteht, eine Datenbank zu aktualisieren und zu indizieren. Die Aufgabe des Indizierens hat dabei keine Data Links zu den anderen Aufgaben. Das Indizieren muss jedoch geschehen, bevor der Workflow weiterarbeiten kann. Das wird durch die Coordination Constraints gelöst. Ein Coordination Constraint kann unabhängig von Data Links definiert werden und gibt einem Processor die Möglichkeit die Ausführungsreihenfolge zu beeinflussen, ohne mit einem Data Link verbunden zu sein. Im Beispiel wird also ein Coordination Constraint bei der Indizierungsaktivität erstellt, der definiert, dass Aktivitäten nur dann in andere Zustände überführt werden können, wenn die Bedingung, dass die Datenbank indiziert wurde, erfüllt ist.

1.3.2. Discovery Net

Discovery Net ist ein Workflowmanagementsystem, das darauf ausgelegt ist, verteilte Daten mit Analysesoftware in einer Grid-Umgebung zu bearbeiten [1]. Eine Grid-Umgebung ist eine lose Kopplung von mehreren Computern, auf denen Anwendungsprogramme ausgeführt werden können.

Das System basiert auf einem Workflowserver. Mit Hilfe dieses Servers kann innerhalb des Grids mit verschiedenen Anwendungsprogrammen gearbeitet werden. Der Server behandelt die Anmeldung und Ausführung der Programme in dem Grid. Außerdem koordiniert er die Daten, mit denen gerechnet wird. Ziel dieses Systems ist es, dass Workflows sehr einfach mit *Drag & Drop* durch den Benutzer generiert werden können. Dabei ist es möglich, die Workflows mit z.B. Web-basierten Oberflächen zu starten, zu bearbeiten oder zu warten. Der Benutzer soll also auf einer möglichst abstrakten Ebene einen Workflow definieren können. Die technischen Einzelheiten sollen dabei verborgen bleiben.

Discovery Net benutzt *DPML*, um die Workflows zu beschreiben. DPML steht für *Discovery Process Markup Language* und ist eine XML-basierte Sprache, um Workflowgraphen zu beschreiben. Es ist möglich, Datenfluss und Kontrollfluss separat zu bearbeiten.

Jeder Knoten in einem DPML-Workflowgraphen repräsentiert eine ausführbare Komponente. Jede dieser Komponenten hat eine Menge von Parametern, Eingabedaten und Ausgabedaten. Die Parameter werden dabei an ein ausführbares Programm weitergeleitet. Die Eingabe- und Ausgabedaten werden als sogenannte Ports an den Komponenten definiert. Diese enthalten zusätzlich Metadaten, um z.B. die Datentypen zu beschreiben, die der Port benutzen kann. Jede Kante im Graphen repräsentiert eine Verbindung zwischen einem Ausgangs- und einem Eingangsport. Es ist darauf zu achten, dass die entsprechenden Datentypen kompatibel sind.

Der Daten- und Kontrollfluss ist hierarchisch organisiert. Die oberste Ebene wird durch den Kontrollflussgraphen beschrieben. Hier wird koordiniert, wann welches Anwendungsprogramm aufgerufen wird. Auf dieser Ebene werden Konzepte wie Schleifen und Wenn-

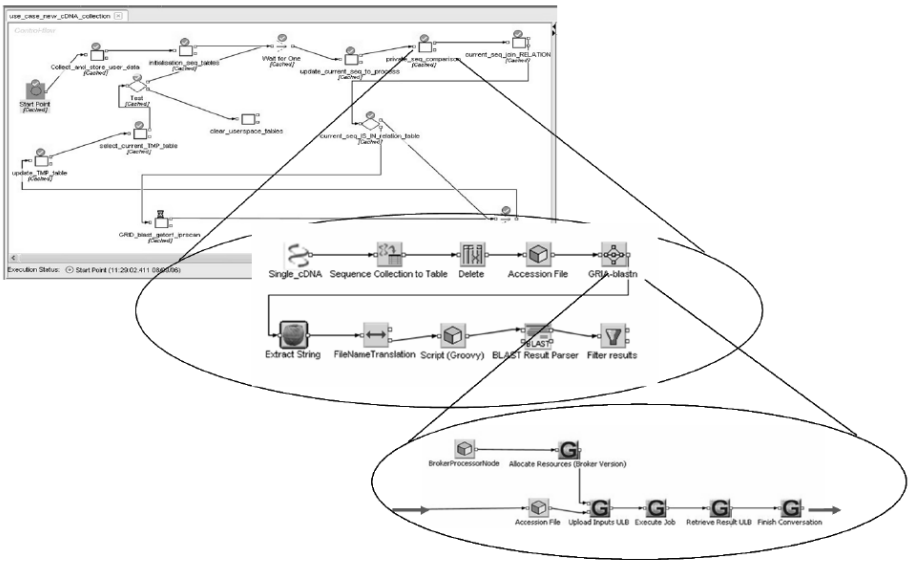

Abbildung 1.2.: Hierarchische Ansicht auf ein Discovery-Net-Workflow. Die oberste Ebene wird durch einen Kontrollflussgraphen beschrieben, der die Ausführungsreihenfolge definiert. Die mittlere Ebene, die Datenfluss-Ebene, beschreibt, wie die Daten bearbeitet werden. In der untersten Ebene wird der Zugriff auf die Grid-Ressourcen sichergestellt. Quelle: [65]

Dann-Beziehungen innerhalb des Workflows realisiert. Dazu wurde das Prinzip der *Tokens* eingeführt. Jede Komponente in einem Kontroll-Fluss-Graphen kann einen Token empfangen, was die Ausführung der Aktion, die diese Komponente beschreibt, veranlasst. Es können mehrere Tokens gleichzeitig in einem Graphen existieren. Komponenten, die mehrere Ausgangsports aufweisen, können selbst entscheiden, an welche Ports sie die Tokens weiterleiten, um so die Richtung, in der das Workflow ausgeführt wird, zu bestimmen. Diese Kontrollflussgraphen werden meist dafür benutzt Workflows zu koordinieren, die im Sinne eines Datenaustauschs nicht miteinander verbunden sind, aber trotzdem voneinander abhängen. Die zweite Ebene beschreibt den Datenfluss. Auf dieser Ebene können Daten in das Workflow geladen werden und von verteilten Anwendungen transformiert werden. Die dritte Ebene ist die Grid-Kontroll-Ebene. Hier wird der Zugriff auf die Grid-Ressourcen koordiniert. Dabei sind typische Aufgaben z.B. die Anmeldung in einem Grid-System, das Hinzufügen von *Jobs* (also Anwendungsprogramme mit festen Parametern, die ausgeführt werden sollen) und das Aufbereiten der Ergebnisse. In Abbildung 1.2 ist ein Workflow mit den unterschiedlichen Ebenen grafisch dargestellt.

Diese hierarchische Sicht hat den Vorteil, dass Benutzer sich nur auf eine Ebene konzentrieren müssen. So können beispielsweise Biologen die erste Schicht bearbeiten und Analysen für Daten definieren, während die Techniker sich mit der dritten Ebene auseinandersetzen.

1.4. Beispiele für Bioinformatikworkflows

In diesem Kapitel werden zwei Analysen von DNA-Daten beschrieben, die beispielhafte reale Anwendungen für ein Workflowmanagementsystem darstellen. Dazu werden zunächst die bio- und technologischen Grundlagen beschrieben, die nötig sind, um die Funktion der Workflows zu verstehen. Anschließend werden die benutzten Dateiformate und Programme vorgestellt. Mit diesem Wissen ist es dann möglich, die Workflows selbst zu erläutern.

1.4.1. Bio- und technologische Grundlagen

Die Desoxyribonukleinsäure (DNA) ist ein Molekül, das in der Form einer Doppelhelix organisiert ist. Sie besteht aus vier verschiedenen Nukleotiden, die jeweils aus einem Zuckermolekül (der Desoxyribose), einem Phosphorsäurerest und einer organischen Base aufgebaut sind. Diese organische Base ist entweder Adenin (A), Guanin (G), Cytosin (C) oder Thymin (T). Die Nukleotide sind durch die Bindung zwischen dem Zucker und dem Phosphat miteinander verknüpft, sodass zwei fadenförmige DNA-Moleküle eine Doppelhelix bilden. Dabei können immer nur ein A und T oder ein G und C eine Bindung eingehen (siehe Abbildung 1.3 a). Die DNA ist der Träger des Erbguts eines Organismus. Die gesamte DNA eines Lebewesens wird auch als Genom bezeichnet. Das Genom ist häufig in verschiedene Teilstücke zerlegt, die zusammen mit Strukturproteinen, die die Ausbildung eines Trägergerüsts ermöglichen, als Chromosom bezeichnet werden. Die Chromosomen eines Menschen befinden sich im Zellkern. Ein genetisch gesunder Mensch hat 23 homologe Chromosomenpaare. Homolog bedeutet, dass beide Chromosomen eines Paares die gleiche Erbinformation kodieren, die Erbinformation daher praktisch doppelt vorhanden ist. Zusätzlich besitzt er ein Geschlechtschromosomenpaar. Frauen haben hier zwei X-Chromosomen, während sich das Paar beim Mann aus einem X- und einem Y-Chromosom zusammensetzt. Somit ergeben sich insgesamt 48 Chromosomen im Zellkern.

Ein *Single Nucleotide Polymorphism* (SNP) ist eine Variation eines Basenpaars in einem DNA-Strang (siehe Abbildung 1.3 b)). Sie sind die häufigste Ursache für genetische Varianten des menschlichen Genoms und sind für genetische Erkrankungen von besonderem Interesse, da sie die DNA so verändern können, dass bestimmte Proteine nicht mehr richtig erzeugt werden können. Dadurch kann das betroffene Protein seine Funktion im Körper möglicherweise nicht mehr richtig ausführen. Der Körper produziert Proteine, in dem er die DNA an bestimmten Stellen abliest.

Aus der Sicht der Informatik ist es ausreichend, die organischen Basen von einem der beiden DNA-Stränge zu speichern. Mit dieser Information kann der gegenüberliegende reverse Strang abgeleitet werden. Die DNA kann also als (sehr langer) String über dem Alphabet A, C, G, T angesehen werden. Das menschliche Genom besteht zum Beispiel aus ca. drei Milliarden Nukleotiden.

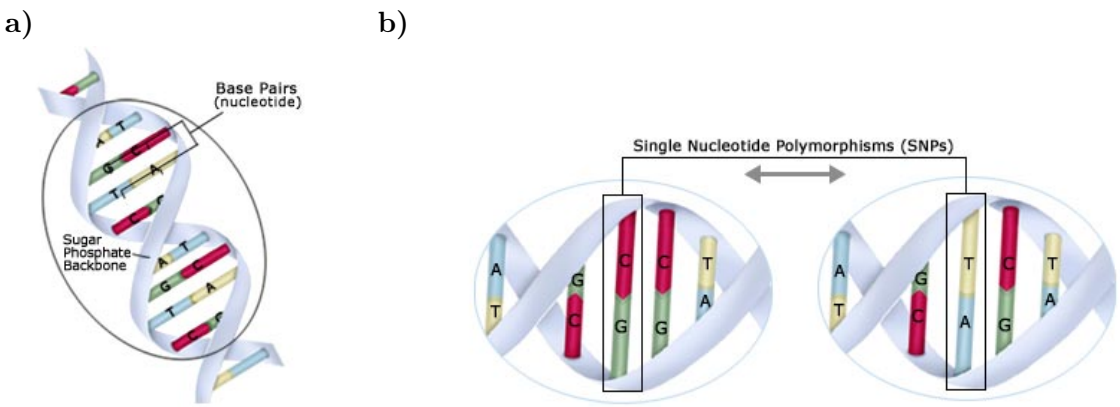

Abbildung 1.3.: DNA Doppenhelix mit Bindungsdarstellung und single nucleotide polymorphism. **a)** Die DNA Doppelhelix: Nur zwischen A, T und G, C kann eine Bindung entstehen. **b)** SNPs sind die häufigsten Varianten im Genom. Bei zwei verschiedenen Genomen steht hier an der selben Position einmal ein C und einmal ein T. Quellen: Abbildung a) und b) sind aus [23] entnommen.

ChIP-Sequencing (ChIP-seq) ist ein Verfahren, um die Interaktionen von Proteinen an DNA-Strängen zu analysieren. Es setzt sich aus der *ChIP-* und der *Sequenzierungstechnik* zusammen. ChIP steht für Chromatin-Immunopräzipitation und ist eine Methode, um festzustellen, ob bestimmte Proteine an Teilen von Chromatin binden. Da Chromatin das Material ist, aus dem Chromosomen bestehen, können so Rückschlüsse auf das Bindungsverhalten von einem Protein an bestimmten Stellen eines DNA-Strangs gezogen werden. Bei der Chromatin-Immunopräzipitation wird die DNA-Protein-Bindung zunächst durch geeignete Methoden fixiert. Anschließend werden die so bearbeiteten Zellen zerstört, so dass die DNA mittels Ultraschallanwendung in Fragmente der Länge einiger hundert Basenpaare zerlegt werden kann. Die DNA-Stücke, an denen das zu untersuchende Protein gebunden ist, werden isoliert und können weiter analysiert werden. Mit Hilfe verschiedener Sequenziertechniken kann die Position dieser DNA Fragmente auf dem Chromosom bestimmt werden.

ChIP-seq wird häufig benutzt um sogenannte Transkriptionsfaktoren zu untersuchen. Das sind Proteine, die sich an bestimmte DNA-Stellen binden und so den Prozess der Proteinsynthese initiieren. In der Biologie sind diese DNA-Stellen von großer Bedeutung. Abbildung 1.4 verdeutlicht den Ablauf der ChIP-Sequencing Methode noch einmal.

1.4.2. Dateiformate

Es haben sich mehrere Dateiformate bei der Arbeit mit DNA-Daten etabliert.

FASTA ist ein einfaches textbasiertes Format, um Sequenzen von Proteinen oder Nukleotiden zu speichern. Die Basen der DNA werden in einem Ein-Buchstaben-Code dar-

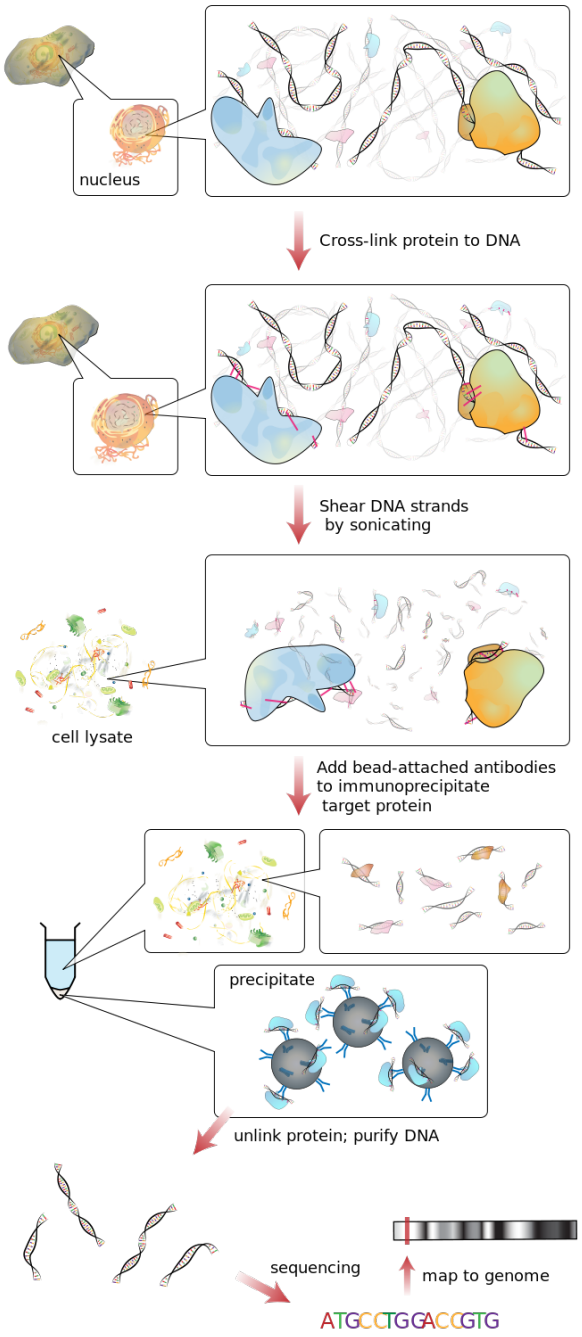

Abbildung 1.4.: Das ChIP-seq Verfahren. Es werden die Stellen der DNA untersucht, an denen bestimmte Proteine binden. Dazu werden zunächst die Zellen, in den die Bindung stattgefunden hat, mittels Formaldehyd fixiert. Anschließend werden sie zerstört und die in ihr enthaltene DNA durch Ultraschallbestrahlung in kleine Fragmente zerlegt. Mit Hilfe von passenden Antikörpern können nun die DNA-Fragmente, an denen sich die zu untersuchenden Proteine befinden, isoliert werden. Diese so aufbereitete DNA kann anschließend sequenziert werden, um ihre Position im Genom zu bestimmen. Quelle: [22]

gestellt. Eine FASTA-Datei beginnt mit einer Kopfzeile, die mit dem Sonderzeichen ">" eingeleitet wird. Die nächsten Zeilen enthalten dann die eigentliche Sequenz. Das Referenzgenom und die Reads aus diesem Beispiel sind in diesem Format gespeichert.

SAM ist eine Abkürzung für *Sequence Alignment/Map*. Es ist ein textbasiertes Format, das mit *TAB* als Trennsymbol arbeitet und ein Alignment beschreibt. Es kann eine oder mehrere Kopfzeilen enthalten, die mit einem "@" beginnen. Die Kopfzeilen enthalten Zusatzinformationen zu dem Alignment wie z.B. den Namen des Programms, mit dem das Alignment erstellt wurde. Jede Zeile steht für ein Alignment, das mit elf Pflichtfeldern beschrieben wird. Es wird unter anderem die Position auf dem Referenzgenom angegeben, sowie ein Qualitätswert zu dem Alignment, der angibt, mit welcher Wahrscheinlichkeit dieses Alignment falsch berechnet worden ist.

BAM ist ein komprimiertes SAM-Format. Eine BAM-Datei ist deutlich kleiner, was bei einer großen Anzahl von Alignments, die durch die Datei beschrieben werden können, von großer praktischer Relevanz ist. Da sie komprimiert ist, ist sie nicht mehr vom Menschen lesbar.

VCF ist ein textbasiertes Dateiformat, mit dem Genompositionen beschrieben werden. Die Abkürzung steht für *Variant Call Format*. Es kann Metainformationen in einer Kopfzeile speichern. Mit diesem Format werden SNPs beschrieben. Es wird unter anderem die Position im Genom, das Referenznukleotid, sowie das Nukleotid, aus dem der SNP besteht, aufgelistet.

BED ist ein Format, um Abschnitte in einem Genom zu beschreiben. Es hat drei Pflicht- und neun optionale Felder. Für jeden Abschnitt muss angegeben werden, auf welchem Chromosom er sich befindet und was seine Start- und Endpositionen sind.

1.4.3. Programme

Die gerade beschriebenen Formate werden von den gängigsten Programmen, die DNA-Daten analysieren, akzeptiert. In den Workflows sind verschiedene Programme nötig, um die DNA-Daten zu analysieren.

BWA [24] (*Burrows-Wheeler-Aligner*) und ist darauf spezialisiert, relativ kurze Nukleotidsequenzen gegen ein großes Genom zu mappen. Es erstellt also ein Alignment, mit dessen Hilfe SNPs gefunden werden können. Außerdem ist es mit dem Programm möglich, das Referenzgenom für einen schnellen Zugriff zu indizieren.

samtools [25] ist ein Werkzeug, um Dateien im SAM-Format zu manipulieren. Insbesondere ist es mit samtools möglich, die Referenzdatei zu indizieren, sodass andere Werkzeuge von samtools einen schnellen Zugriff darauf haben. Zudem kann eine

SAM-Datei nach den Koordinaten sortiert und in das BAM-Format konvertiert werden.

picard [2] ist eine Sammlung von jar-Dateien, mit denen der Benutzer in der Lage ist, Dateien in verschiedene Formate zu transformieren.

MACS [68] steht für *Model-based Analysis of ChIP-Seq* und ist drauf ausgerichtet, Transkriptionsfaktorbindestellen zu identifizieren, die mit dem ChIP-Seq-Verfahren untersucht werden sollen. MACS verwendet einen Algorithmus der mit statistischen Mitteln und Kontrolldaten mögliche Bindestellen in den DNA-Fragmenten ermittelt.

GATK [26] (*Genome Analysis Toolkit*, kurz GATK) bietet verschiedene Funktionen um Genomdateien zu analysieren. Mit einer SAM-Datei und einem Referenzgenom ist es möglich, SNPs zu identifizieren und sie in eine VCF-Datei zu speichern.

1.4.4. Workflow zur Identifizierung eines SNPs

Eine mögliche Analyse von Genomen ist das Identifizieren von SNPs. Dabei wird eine DNA mit einer Referenz-DNA, dem sogenannten Referenzgenom, abgeglichen. Dieses Referenzgenom ist der Durchschnitt der Genome von vielen genetisch gesunden Menschen. Die DNA liegt dabei jedoch nicht als ein langer String vor, sondern als eine Menge von kurzen Wörtern. Diese Wörter werden Reads genannt und haben eine Länge von ca. $100 - 200$ Basen. Die DNA kann aus technischen Gründen nur in diesen kurzen Abschnitten gewonnen werden.

Die Information, aus welcher Stelle des Genoms die Reads kommen, geht dabei verloren. Es muss ein Alignment zwischen den Reads und dem Referenzgenom berechnet werden, um die Unterschiede in der DNA identifizieren zu können. Ein Alignment vergleicht zwei Strings auf ihre Ähnlichkeit. Wenn ein Read auf das Referenzgenom aligniert wurde, heißt das, dass an dieser Stelle (fast immer) die gleichen Basenpaare zu finden sind. Der Read wurde also aus dieser Stelle des Genoms gewonnen. Die Basenpaare sind nicht immer gleich, da SNPs im Read vorhanden sein können. Das Alignment muss also Fehlern gegenüber tolerant sein.

Ein Unterschied in der DNA kann beispielsweise auf eine mögliche genetische Ursache einer Krankheit hinweisen. Es ist anzumerken, dass die meisten SNPs jedoch nicht die Ursache einer Krankheit darstellen.

Wie bereits beschrieben, wird mit Hilfe des Workflows aus einer Referenzdatei und einer Datei, die Reads enthält, ein Alignment erstellt. Mit diesem Alignment wird ein SNP identifiziert und in einer VCF-Datei abgespeichert. Dazu werden die oben beschriebenen Programme und verschiedenen Dateien mit verschiedenen Formaten benutzt.

Im Workflow werden zuerst die nötigen Indizierungen an der Referenzdatei vorgenommen. Das geschieht mit den Programmen bwa und samtools (genauer mit den Kommandos

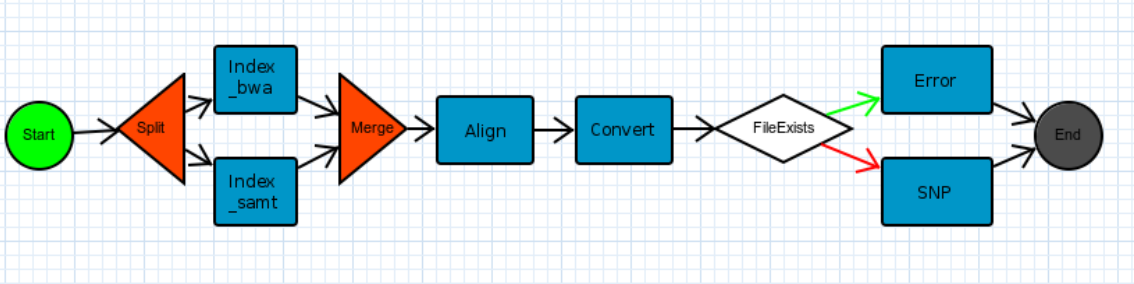

Abbildung 1.5.: Der Beispiel-Workflow als Graph dargestellt. Vom Start-Knoten aus wird mit Hilfe des Split- und Merge-Knotens eine Indizierung an der Referenzdatei vorgenommen. Anschließend wird ein Alignment erstellt und in das BAM-Format konvertiert. Es wird geprüft, ob das Alignment richtig erstellt worden ist. Ist dies der Fall, können SNPs gesucht werden. Ansonsten wird ein Fehler ausgegeben. Beide Knoten werden in den End-Knoten überführt. Der Workflow, die Lokalisierung von SNPs gegenüber einer Referenzdatei, ist beendet.

bwa index und *samtools faidx*) und kann parallel ausgeführt werden. Diese Indizierungen sind für den weiteren Workflowverlauf nötig. Anschließend wird mit bwa ein Alignment im SAM-Format erstellt (*bwa aln* und *bwa samse*). Die SAM-Datei wird mit Hilfe von samtools (*samtools view*) in das BAM-Format konvertiert. Die BAM-Datei wird manipuliert, sodass das GATK mit der Datei arbeiten kann. Dazu wird sie nach anhand der Koordinaten sortiert (*samtools sort*) und ein fehlendes Attribut gesetzt (*picard/AddOrReplaceReadGroups.jar*). Dieses Attribut stellt kein Pflichtfeld in der Formatspezifikation dar und wird nur hinzugefügt, da GATK dies benötigt. Anschließend wird sie mit samtools indiziert (*samtools index*). Nach diesen Modifikationen wird überprüft, ob die Alignmentdatei erstellt worden ist. Im positiven Fall, hat der Workflow ohne Probleme gearbeitet und durch GATK kann eine VCF-Datei erstellt werden, die SNPs auflistet (*GenomeAnalysisTK.jar -T UnifiedGenotyper*). Wenn die Datei nicht existiert, dann trat ein Problem innerhalb des Workflows auf. Dies kann dem Endbenutzer mitgeteilt werden.

In Abbildung 1.5 ist der Workflow als Graph dargestellt. Anhand der *split*- und *merge*-Knoten wird ersichtlich, dass die Indizierung parallel ausgeführt wird. Anschließend wird ein Alignment erstellt und in das passende Dateiformat konvertiert. Wenn das Alignment korrekt erstellt worden ist, die Alignmentdatei also existiert, können SNPs identifiziert werden. Falls das Alignment nicht korrekt erstellt worden ist, wird ein Fehler ausgegeben. Diese Fallunterscheidung wird in dem *Exist*-Knoten realisiert.

Abbildung 1.6 zeigt einen Ausschnitt der generierten VCF-Datei. Mit dem VCF-Format ist es möglich, wie in Kapitel 1.4.2 beschrieben, Positionen im Genom zu beschreiben. In den letzten beiden Zeilen ist zu sehen, dass ein SNP gefunden wurde. Die Reads wurden also zunächst gegen das Genom aligniert und mit Hilfe des Alignments konnte ein SNP

```
##INFO=<ID=MQ,Number=1,Type=Float,Description="RMS Mapping Quality">
##INFO=<ID=MQ0,Number=1,Type=Integer,Description="Total Mapping Quality Zero Reads">
##INFO=<ID=MQRankSum,Number=1,Type=Float,Description="Z-score From Wilcoxon rank sum test of Alt vs. Ref
##INFO=<ID=QD,Number=1,Type=Float,Description="Variant Confidence/Quality by Depth">
##INFO=<ID=ReadPosRankSum,Number=1,Type=Float,Description="Z-score from Wilcoxon rank sum test of Alt vs.
##INFO=<ID=SB,Number=1,Type=Float,Description="Strand Bias">
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  foo
chr22:16400000-48830000 37      .       C       T       63      .       AC=1;AF=0.50;AN=2;BaseQRankSum=0.
00;MQ0=0;MQRankSum=0.727;QD=15.75;ReadPosRankSum=0.727;SB=-0.01 GT:AD:DP:GQ:PL  0/1:1,3:4:21.41:93,0,21
```

Abbildung 1.6.: Ausschnitt der VCF-Datei: Ein Ausschnitt aus der VCF-Datei, die durch den Workflow erstellt wird, ist hier dargestellt. Die Kopfzeilen und Metainformationenzeilen werden mit # und ## eingeleitet. Die letzten beiden Zeilen enthalten Informationen über den SNP. Das SNP wurde an der Position 37 im Chromosom 22 lokalisiert. Das Referenznukleotid ist C. Der SNP hat an dieser Stelle ein T.

lokalisiert werden. Die zu untersuchende DNA unterscheidet sich also in einem Nukleotid von dem Referenzgenom.

1.4.5. Workflow zur Analyse von ChIP-seq-Daten

Mit diesem Workflow wird das Genom von Mäusen mit Hilfe der ChIP-seq-Methode untersucht. Es sollen Transkriptionsfaktorbindestellen in den Genomen identifiziert werden. Diese DNA-Abschnitte sind von besonderem Interesse, da dort spezielle Proteine, die sogenannten Transkriptionsfaktoren, binden. Diese Proteine lesen die DNA ab, um so den Prozess der Proteinsynthese zu starten.

Wie bereits oben beschrieben, bietet die Chip-Seq Analyse die Möglichkeit, kleine DNA-Fragmente aus diesen Bindestellen zu extrahieren. Diese DNA-Fragmente werden gegen ein Referenzgenom aligniert, um so ihre Position auf dem Genom festzustellen. Dieses Prinzip wird auch bei der SNP-Analyse im ersten Workflowbeispiel verwendet. Dadurch sind bei einem so berechneten Alignment die Reads genau auf den Transkriptionsfaktorbindestellen gemappt. Anschließend müssen nur noch diese Stellen im Genom lokalisiert werden, um so die Position der Bindestellen zu erhalten.

Die Daten, die bei diesem Workflow analysiert werden und die durch das ChIP-seq-Verfahren gewonnen wurden, stammen aus drei verschiedenen Mausgenomen, sowie einem Kontrollgenom, dass aus technischen Gründen zur Normalisierung angegeben werden muss. Die Genome werden unter den Namen *Gli3Amyc*, *Gli3Rmyc*, *Trps1myc* geführt und das Kontrollgenom hat den Bezeichner *myc*. Die Daten sind als Reads im FASTA-Format gegeben. Zu jedem Genom sind diese DNA-Fragmente in mehreren FASTA-Dateien zu finden.

Zuerst werden, ähnlich wie im ersten Beispiel, alle vorhanden Reads gegen das Referenzgenom der Maus gemappt. Dabei wird im Workflow vorausgesetzt, dass die Referenz bereits entsprechend indiziert wurde. Die Indizierung muss nur einmal geschehen, jede Analyse kann dann auf sie zurückgreifen. Die vier Genome können gleichzeitig aligniert werden. Im

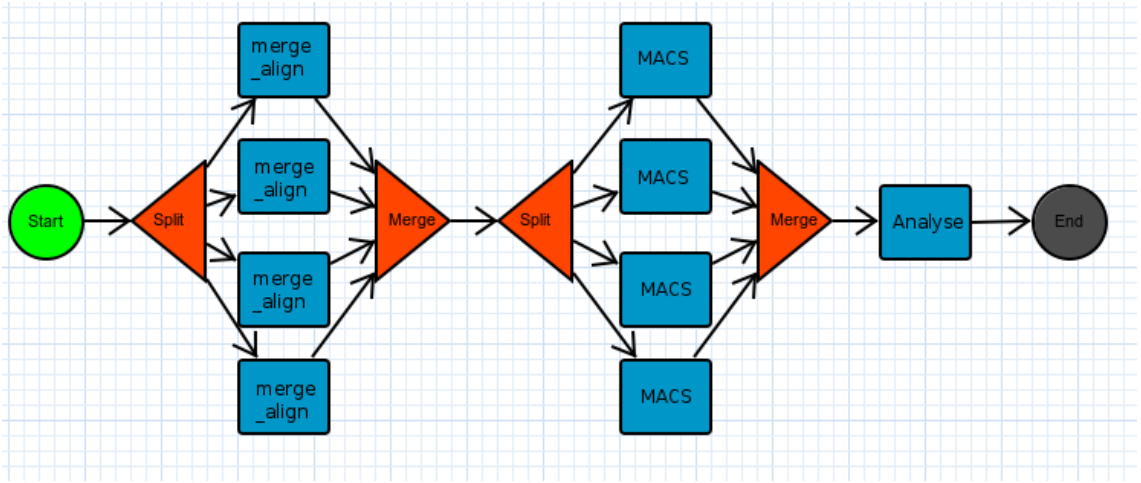

Abbildung 1.7.: Der Beispielworkflow als Graph dargestellt. Vier Mausgenome werden parallel gegen ein Referenzgenom gemappt. Das wird durch den ersten Split-Merge Knoten realisiert. Alle so alignierten Reads können nun auf Transkriptionsfaktorbindestellen hin untersucht werden. Das geschieht mit dem Programm MACS im nächsten Split-Merge-Konstrukt. Im letzten Workflowschritt können die von MACS erzeugten Daten weiter analysiert werden.

Workflow wird deshalb mit einem Split-Merge-Konstrukt das Skript *merge_ align* viermal mit unterschiedlichen Parametern gestartet. Der Parameter enthält den Namen des Genoms, das bearbeitet werden soll. Das Skript ruft *BWA* auf und aligniert im ersten Schritt alle Reads des entsprechenden Genoms auf das Referenzgenom und für jede FASTA-Datei erhält man so eine BAM-Datei, die das Alignment beschreibt. Diese BAM-Dateien werden dann in einem zweiten Schritt mit dem Befehl *samtools merge* in eine einzelne BAM-Datei konvertiert, die den Namen des Genoms mit der Endung *bam* bekommt. Man erhält somit vier BAM-Dateien (Gli3Amyc.bam, Gli3Rmyc.bam, Trps1myc.bam und myc.bam), die das Alignment der entsprechenden Reads gegenüber dem Mausreferenzgenom beschreiben.

Die alignierten Reads sollten dabei nur an den Transkriptionsfaktorbindestellen zu finden sein. Um die BAM-Dateien auf Bindestellen hin zu untersuchen, wird im nächsten Workflowschritt *MACS* verwendet. Dieser Schritt kann für alle drei BAM-Dateien parallel ausgeführt werden. *MACS* benötigt für den verwendeten Algorithmus die Datei *myc.bam* zur Normalisierung. Das Programm gibt die Stellen, an denen besonders viele Reads im Genom aligniert worden sind, unter anderem als BED-Datei aus. Somit hat man am Ende des Workflows potentielle Transkriptionsfaktorbindestellen in BED-Dateien abgespeichert, die näher untersucht werden können. Abbildung 1.8 zeigt so eine Datei. Jede Zeile beschreibt eine potentielle Bindestelle. Die erste Spalte gibt dabei jeweils den Namen der Referenzdatei an. In der zweiten und dritten Spalte kann man die Position im Genom ablesen. In der letzten Spalte kann man die durchschnittliche Anzahl an Reads ablesen, die auf diesem Bereich gemappt wurden. Diese Datei kann für weitere Analysen verwendet werden.

```
gi|149247747|ref|NT_166280.1|   16760   17919   MACS_peak_1     92.13
gi|149247747|ref|NT_166280.1|   18545   21318   MACS_peak_2     109.12
gi|149247747|ref|NT_166280.1|   24217   24714   MACS_peak_3     58.80
gi|149247747|ref|NT_166280.1|   31804   33878   MACS_peak_4     95.31
gi|149247747|ref|NT_166280.1|   34409   35259   MACS_peak_5     80.26
gi|149247747|ref|NT_166280.1|   77601   78216   MACS_peak_6     57.27
gi|149247747|ref|NT_166280.1|   136926  137954  MACS_peak_7     74.77
```

Abbildung 1.8.: Ausschnitt der BED-Datei. Ein Ausschnitt aus der BED-Datei, die von dem Programm *MACS* im letzten Schritt des Workflows generiert wird. Die Datei enthält keine Kopfzeilen, die die Spalten beschreiben würden. Jede Zeile der Datei gibt eine mögliche Bindestelle im Genom an. Die erste Spalte beschreibt dabei die Referenzdatei, die benutzt wurde. Anhand der zweiten und dritten Spalte werden die Start- und Endposition des berechneten Abschnittes abgelesen. Die vierte Spalte gibt den (von *MACS* generierten) Namen der Sequenz an und in der letzten wird die durchschnittliche Anzahl an Reads abgelesen, die auf diesen Genombereich gemappt wurden.

2. Beschreibung der Sprachen in Eclipse4Bio

In diesem Kapitel, werden Technologien beschrieben, die bei der Entwicklung der Service- und Workflowsprachen in Eclipse4Bio zum Einsatz gekommen sind. Im Anschluss daran werden im Kapitel 3 auf Seite 33 Technologien beschrieben, welche die Entwicklung einer textuellen Eingabesprache erlauben, mit deren Hilfe der Benutzer in der Lage ist, die benötigten Daten einzugeben. Es wurden zwei verschiedene Sprachen entwickelt, die zusammen die Funktionalität in Eclipse4Bio abbilden.

ServiceDefinition Diese Sprache wird benötigt um alle ausführbaren Komponenten in einem Workflow definieren zu können.

WorkflowDefinition Diese Sprache orchestriert die Komponenten die mit der ServiceDefinition Sprache definiert worden.

2.1. Metamodellierung

Um eine Sprache, d.h. auch eine Programmiersprache, zu entwickeln kann ein Metamodell definiert werden. Ein Metamodell ist eine formale Beschreibung für ein Modell, welches durch einen Benutzer erzeugt werden soll. Dabei kann ein Modell auch in Form einer Sprache definiert werden. Im Folgenden wird die Entwicklung von Metamodellen mit Hilfe der Eclipse-Technologien EMF und ECore [9] beschrieben. Im Anschluss daran wird eine kurze Einführung in die Arbeit mit ECore gegeben.

Die *Object Management Group* (kurz OMG) hat unter anderem eine Spezifikation für die Entwicklung von Metamodellen entwickelt [36]. Die *OMG* ist ein Konsortium, welches Standards im Bereich der objektorientierten Programmierung entwickelt und wartet. Die *OMG* verwaltet zur Zeit u.A. die Standards für BPMN [31], Corba [29], UML [37], [38] und XMI [33].
Neben diesen Bereichen beschäftigt sich die *OMG* auch mit der Entwicklung von Verfahren und Modellen, um ein einheitliches Format zu entwickeln, mit dem anderen Standards beschrieben werden können. Dafür und für die modellgetriebenen Entwicklung (MDA) [39] hat die OMG Standards wie die OMA (Object Management Architecture) [40], QVT (Query/View/Transformation) [32] und auch MOF (Meta Object Facility) [34] entwickelt.

Die MOF beschreibt den Aufbau von Metamodellen und geht dabei von einem Mehrschichtenmodell aus. Dieses Mehrschichtenmodell führt vier verschiedene Ebenen ein, die je nach Abstraktionsgrad die Realwelt beschreiben und von dieser Schicht ausgehend immer weitere Abstraktionen einführen (siehe [30]). Die Abbildung 2.1 zeigt den Aufbau des Mehrschichtenmodells anhand eines UML2-Beispiels (Abbildung nach [30, Abbildung 7.8, Seite 20]). Die M0-Schicht ist dabei das echte Objekt, welches durch das Modell beschrieben werden soll. In diesem Fall ist es ein echter Film, der als Medium vorliegt. Die M1-Schicht beschreibt dieses Video nun aus der Sicht der einfachen UML-Deklaration. Dafür wird mit dem UML-Klassendiagramm eine Klasse erzeugt, die ein Attribut enthält, um den Namen des Films zu speichern. Des Weiteren wird mit dem UML-Objektdiagramm (basierend auf der vorher erzeugten Klasse) ein Objekt definiert, welches als Repräsentation für den Film dienen soll. Dabei ist zu beachten, dass der echte Film aus der M0-Schicht als Vorlage für die Klasse *Film* in der M1-Schicht dient.

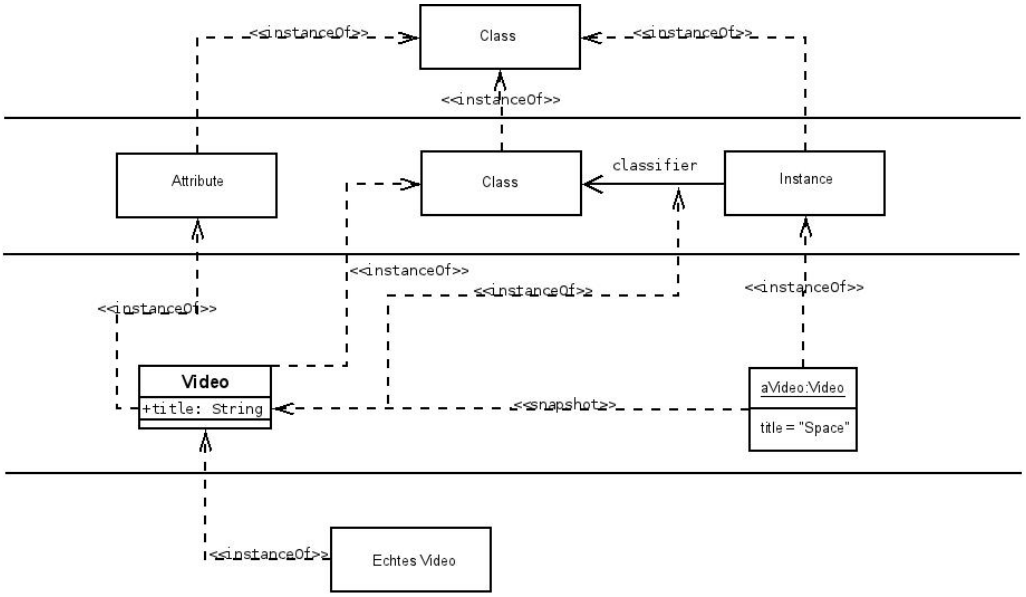

Abbildung 2.1.: M0 - M3 Schichtenmodell der OMG, Quelle: [30]

Dabei geht es um eine formale Beschreibung, welche nur auf das Wesentliche beschränkt ist.
Um die UML-Elemente *Klasse*, *Attribut*, *Objekt* und *Assoziation* benutzen zu können, muss es eine formale Beschreibung dieser Komponenten geben. Diese Komponenten werden in der M2-Schicht beschrieben. Die *Attribute*-Komponente aus der M2-Schicht beschreibt zum Beispiel den genauen Aufbau eines Attributes für die UML. Aus diesem Grund sind alle Elemente in der M1-Schicht immer Instanzen aus der formalen Beschreibung der M2-Schicht. Dabei ist zu beachten, dass zum Beispiel das Objekt-Diagramm und das Klassen-Diagramm in der M2-Schicht nur beschreiben wie eine konkretes Modell aussehen muss. Die eigentlichen Instanzen werden dann in der M1-Schicht definiert. Die Abhängigkeit von Objekt-

und Klassendiagrammen ist an der Stelle nicht wichtig [37] [38].
Die Komponenten der M2-Schicht müssen am Ende nochmals beschrieben werden. Theoretisch würde die formale Beschreibung in der M2-Schicht ausreichen, um mit der UML zu arbeiten (siehe [30]). Da aber die OMG mit dem MOF-Ansatz nicht nur die UML2 beschreiben, sondern diese als Technologie zur Beschreibung von allen Sprachen benutzen wollte, musste eine weitere Abstraktionsschicht hinzugefügt werden, mit deren Hilfe die UML und andere Sprachen definiert werden können. Diese Schicht beschreibt dann die Definition der anderen Sprachen. Diese Schicht wird M3 genannt. Zu beachten ist, dass sich die M3-Schicht reflexiv selbst beschreibt und somit auch eine formale Beschreibung besitzt.

MOF basiert intern auf zwei verschiedene Komponenten, die Essential MOF und die Complete MOF (siehe [35, Kapitel 12]). Die Essential MOF ist die Haupt- bzw. Kernkomponente von MOF und wird benötigt, um einfache Metamodelle zu erzeugen. Die Complete MOF wird benötigt, um die MOF vollständig zu beschreiben. Die CMOF fügt zum Beispiel Elemente des Package-Imports oder des Package-Merges hinzu.

Wenn eine neue Sprache entwickelt werden soll, kann der M0-M3-Ansatz der Metamodellierung benutzt werden, um sie formal zu beschreiben. Dabei muss der Sprachdesigner und -entwickler die M2-Schicht für seine neue Sprache entwickeln und definieren. Alle Benutzer dieser neuen Sprachen werden darauf aufbauend Instanzen der M1-Schicht erzeugen, welche dadurch formal valide zur neuen Sprache sind.

2.2. EMF

Das *Eclipse Modeling Framework* (kurz EMF) [9] ist ein Eclipse-Projekt, welches das Ziel hat, aus Modellen Code zu erzeugen. Dafür wurden in dem EMF-Projekt einige Komponenten entwickelt, mit denen entsprechende Modelle erzeugt (ECore) und transformiert (JET) [8] werden können. Des Weiteren existieren viele weitere Komponenten, welche die Arbeit mit den Metamodellen und der Generierung von Code vereinfachen sollen.

ECore

Das ECore-Teilprojekt wird innerhalb von EMF benutzt, um die eigentlichen Metamodelle zu erzeugen. Dabei beruht EMF auf dem Essential MOF der OMG. Die Abbildung 2.2 auf der nächsten Seite (siehe [7]) zeigt den Aufbau der ECore-Komponenten.

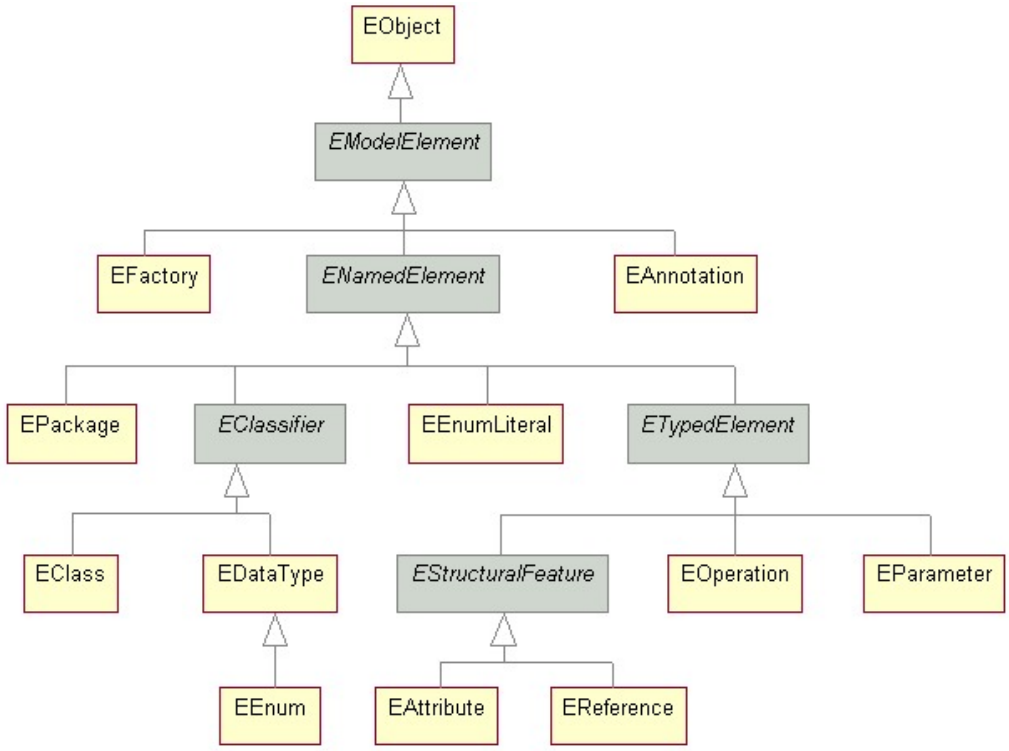

Abbildung 2.2.: ECore Komponenten, Quelle: [7]

Weitere Projekte

Das Eclipse Modeling Framework beinhaltet weitere Projekte:

Compare ist ein Teilprojekt, das es einem Entwickler erlaubt, verschiedene Metamodelle miteinander zu vergleichen.

Model Query ist eine Technologie mit der Anfragen an ein Modell erstellt werden können.

Teneo kann benutzt werden um die Modelldaten persistent in einer Datenbank zu speichern.

Validation Framework stellt Möglichkeiten zur Verfügung, Validierungsregeln in das Metamodell aufzunehmen.

2.3. ECore Modelle

Es gibt verschiedene Möglichkeiten, mit EMF Metamodelle zu erzeugen. Die gängigsten Methoden werden im Folgenden vorgestellt.

XMI-Dokument erzeugen

Die eigentliche Repräsentation der Metamodelle erfolgt im XMI-Format, ein Container-Format, in dem viele verschiedene Modellinstanzen gespeichert werden können. Es gibt verschiedene Möglichkeiten ein XMI-Dokument zu erzeugen. Intern sind XMI-Dokumente nur XML-Dateien, die einem bestimmten Schema genügen müssen. Sie können auf verschiedene Arten erzeugt werden, beispielsweise mit einem Texteditor. Einfacher wird es mit einem XMI-fähigen Entwicklungstool, wie zum Beispiel *Rational Rose* [20]. Auch Eclipse selbst bietet verschiedene Editoren zum Erzeugen an, die für diese Aufgabe geeignet sind. Eine Möglichkeit stellt der XML-Editor dar (siehe Abbildung 2.3 auf der nächsten Seite und Abbildung 2.4 auf Seite 24), von dem zwei Arten, ein XML-Baumstruktur- und ein grafisch-basierter Editor, existieren (siehe Abbildung 2.5 auf Seite 25).

Annotierte Java-Klassen

Eine weitere Variante ist das Entwickeln von Metamodellen mit Hilfe von Java-Klassen und Annotationen. Diese Annotationen erweitern eine Java-Klasse um die entsprechenden Informationen, die nötig sind, um das Metamodell zu erzeugen. Die Attribute der Java-Klasse stellen dabei die Eigenschaften der Metamodellklasse dar. Im Listing 2.1 auf Seite 26 wird ein einfaches Metamodell beschrieben.

In diesem Beispiel werden zwei Modellklassen erzeugt. Diese sind durch das Doclet [41] *@model* gekennzeichnet. Diese Kennzeichnung wird auch für das Markieren der Attribute in der jeweiligen Klasse benötigt. Durch die Schlüsselwörter *type* und *containment* werden verschiedene Eigenschaften der Assoziation gekennzeichnet. Mit der Angabe des *type* wird der Typ des Assoziationsziels angegeben. Bei primitiven Java-Typen muss kein Wert angegeben werden, da diese automatisch erkannt werden können.

Das Attribut *containment* wird angegeben, wenn es sich bei der Assoziation um eine Aggregation handelt. Das bedeutet, dass die Elemente des Assoziationsziels durch diese Instanz verwaltet werden und nur durch diese erzeugt bzw. zerstört werden können. Es handelt sich also um eine A ist in B enthalten Beziehung. Um von anderen Modellelementen zu erben, kann das normale Vererbungsprinzip von Interfaces benutzt werden, was auch die Mehrfachvererbung ermöglicht.

XML Schemata

Eine weitere Variante zum Erzeugen von Metamodellen ist das Verwenden von XML-Schemata. Im Listing 2.2 auf Seite 27 ist das Beispiel aus dem vorherigen Kapitel (siehe Listing 2.1)nochmals als XML Schema dargestellt.

Zu beachten ist, dass in diesem Fall jeder *complexType* einer Komponente im Metamodell entspricht. Dabei ist die Angabe des Namespaces wichtig. Der Namespace *ecore*

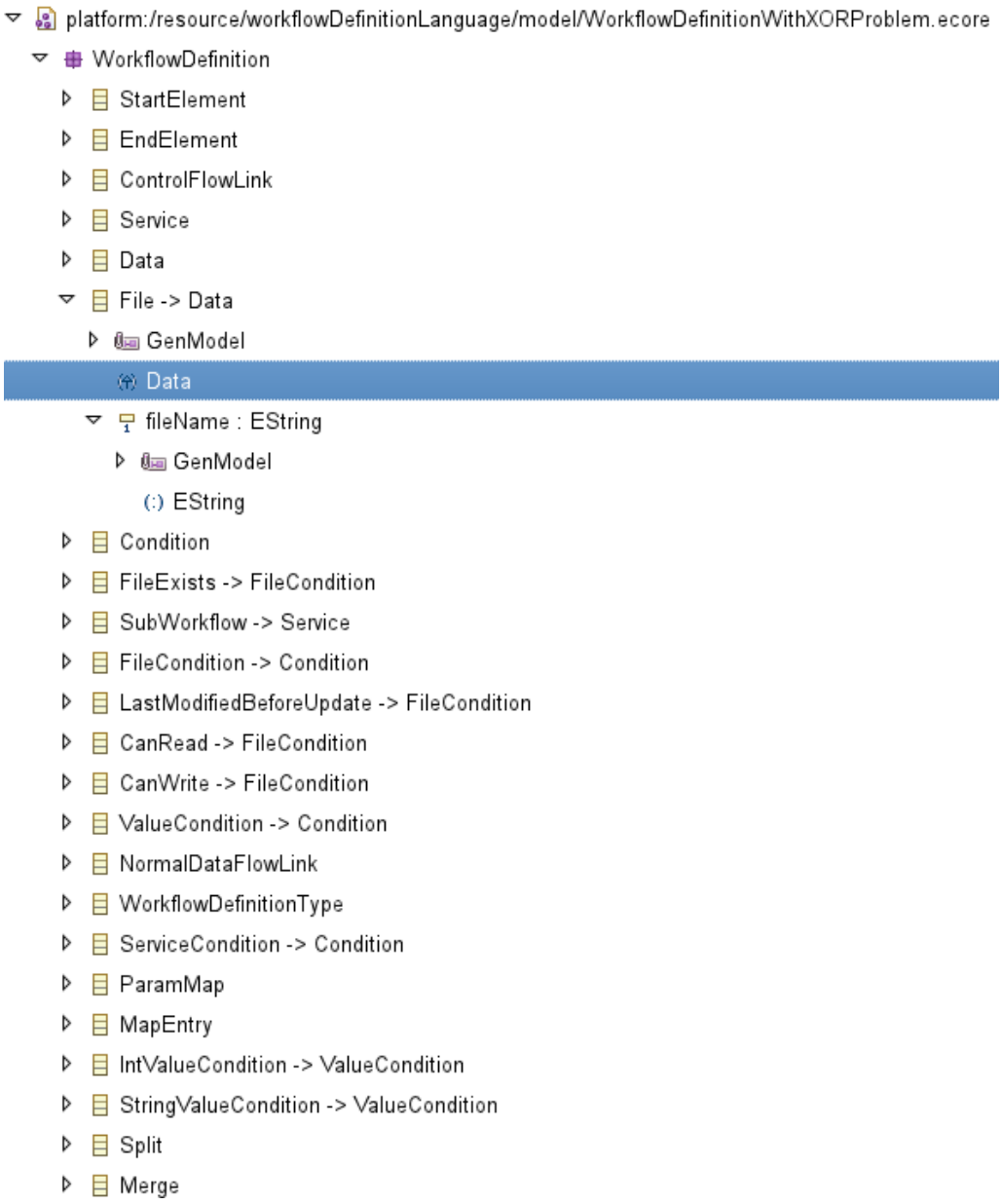

Abbildung 2.3.: Diese Ansicht stellt die Struktur eines XMI-Dokuments dar. Da ein XMI-Dokument ein XML-Dokument ist, ist die Darstellung als Baum aufgebaut. In diesem Beispiel gibt als Hauptkomponente des Metamodells das WorkflowDefinition-Element, welches sich in verschiedene Unterelemente aufteilt, welche wiederum aus verschiedenen Elementen bestehen kann (Attribute und Dokumentation). In Summe ergibt diese Darstellung das Metamodell für die Workflow-Definition.

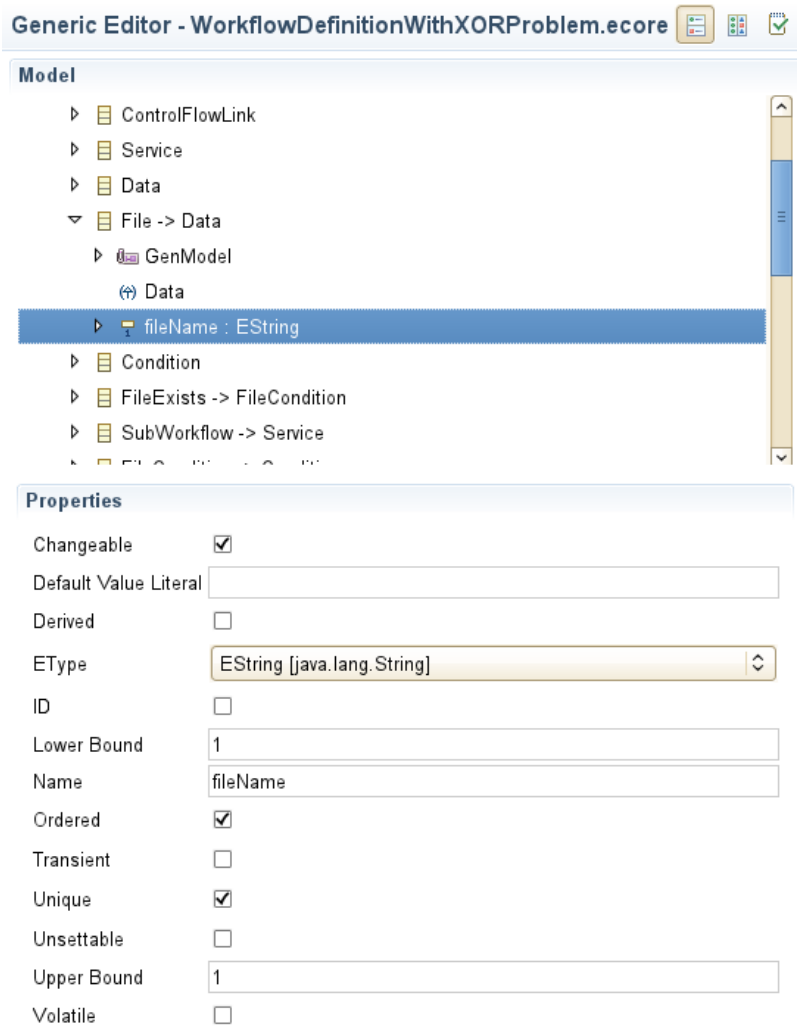

Abbildung 2.4.: Dieser Eclipse-Editor ermöglicht dem Entwickler die Manipulation des Metamodells. Dabei können verschiedene Elemente hinzugefügt werden und die Attribute definiert werden.

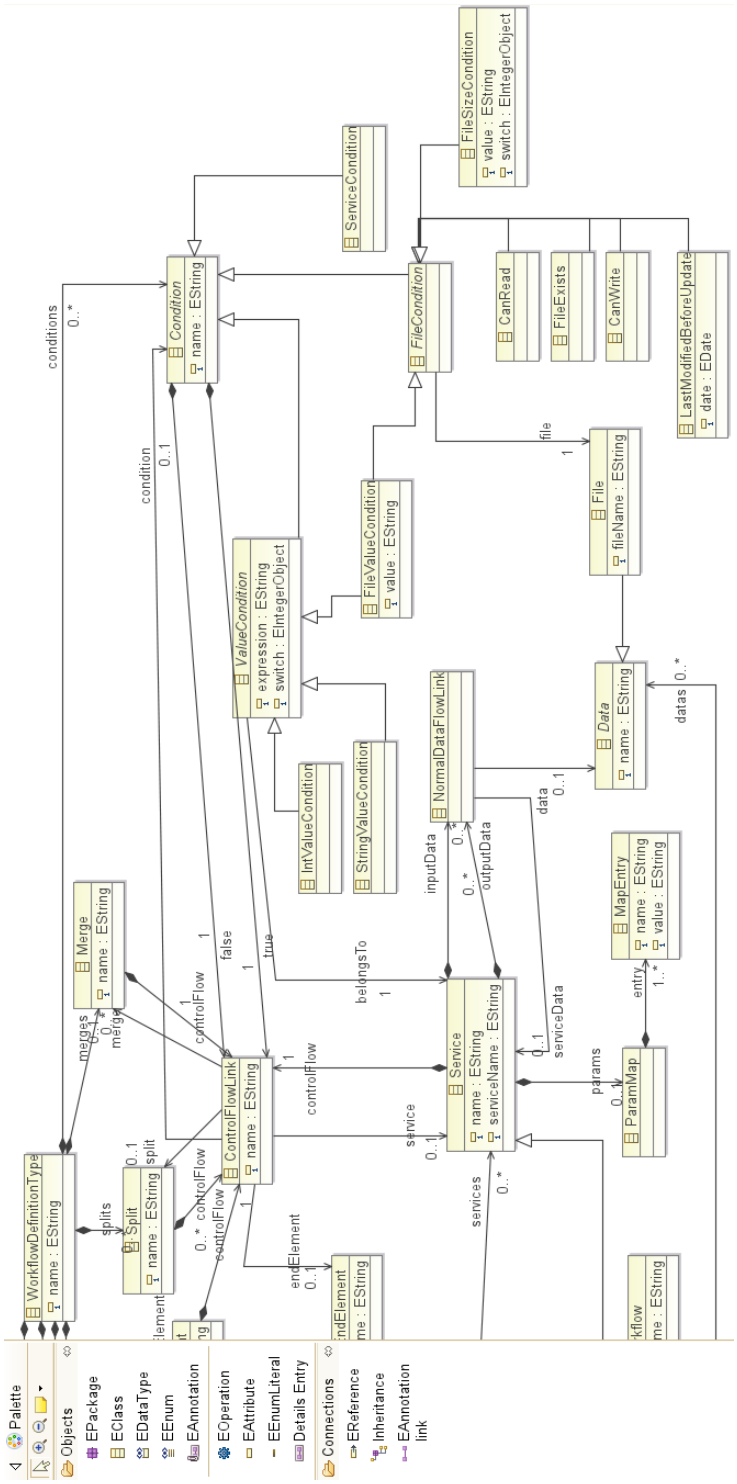

Abbildung 2.5.: Dieser Editor ermöglicht dem Entwickler eine grafische Manipulation des Ecore-Metamodells. Über die Elemente auf der linken Seite können Attribute und Klassen hinzugefügt werden. Des weiteren können Beziehungen definiert werden.

```
1  package de.tu_dortmund.cs.pg.eclipse4io.demo;
   import java.util.List;
3
   /**
5   * @model
    */
7  public interface Service {
     /**
9     * @model
      */
11   String getName();

13   /**
      * @model type="File" containment="true"
15    */
     List getFiles();
17 }

19 /**
    * @model
21  */
   public interface File {
23   /**
      * @model
25    */
     String getName();
27 }
```

Listing 2.1: Metamodell aus annotierten Java-Klassen

definiert alle wichtigen Komponenten, die für das Metamodell benötigt werden. Da in diesem Beispiel nur eine sehr einfache Variante beschrieben wird, wird der Namespace *ecore* nicht benutzt. Der *targetNamespace* muss definiert und als weiterer Namespace importiert werden (hier der Namespace *example*).

2.4. Erzeugen eines einfachen Metamodells

Im Folgenden wird ein einfaches Metamodell entwickelt. Um möglichst nah an der Thematik der Projektgruppe zu bleiben, soll es um die Entwicklung einer Workflowsprache gehen, die ein Start- und Ende-Element hat und aus einer Sequenz von Services bestehen kann. Das Metamodell in der Abbildung 2.6 entspricht dabei dem folgenden Ecore-Modell.

Das Metamodell definiert drei Komponenten. Alle drei Komponenten haben jeweils ein *name*-Attribut und unterschiedliche Kanten, die in der textuellen Darstellung durch den *eType* dargestellt werden.

```
<xsd:schema targetNamespace="http://www.cs.tu-dortmund.de/pg/eclipse4bio/
    example"
     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
     xmlns:example="http://www.cs.tu-dortmund.de/pg/eclipse4bio/example"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Service">
     <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="category" type="example:File"/>
     </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="File">
     <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
     </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Listing 2.2: Metamodell aus einem XML-Schema

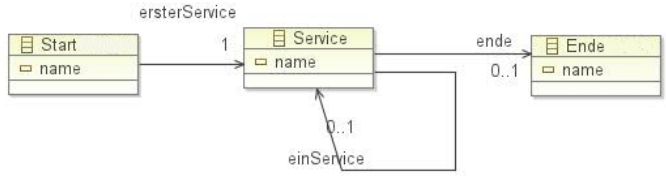

Abbildung 2.6.: Metamodell des Minibeispiels

Die Metamodelle sind in dieser Form vollständig modelliert. Alle wichtigen Aspekte (die Syntax) sind definiert. Weitere Informationen über das Metamodell (die Syntax) müssen nachträglich durch Validierungen hinzugefügt werden.

2.5. Service-Definition Metamodell

Im folgenden werden nun die konkreten Metamodelle beschrieben die in Eclipse4Bio entwickelt worden. Abbildung 2.7 zeigt das Metamodell für die Servicedefinition. Das Metamodell ist zugleich die Spezifikation für ein Service. Ein Service besteht aus den folgenden Komponenten:

ServiceContainer Der Container wird benutzt um verschiedene Servicedefinitionen zu speichern. Der Container muss einen Namen besitzen. Allerdings muss der Container nur dann definiert werden, wenn mehr als ein Service definiert wurde. Der Name wird benötigt um bei Mehrdeutigkeit immer den richtigen Service angeben zu können.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
        XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="">
  <eClassifiers xsi:type="ecore:EClass" name="StartKnoten">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ersterService"
        lowerBound="1"
        eType="#//Service"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Service">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ende" eType="#//
        EndKnoten"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="einService" eType
        ="#//Service"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="EndKnoten">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"/>
  </eClassifiers>
</ecore:EPackage>
```

Listing 2.3: Metamodell aus dem Minimodell

Service Der *Service* definiert die eigentliche Funktion, die ausgeführt werden soll. Dabei können Parameter (siehe *ParamMap*) und Aktionen, die ausgeführt werden sollen (siehe *BashDo*), angegeben werden.

Params Durch den Aufbau von Aufbau von EMF können Parameter Maps nur durch einen Kontainer erstellt werden, der die eigentlichen Werte enthält. Das *ParamMap* ist dabei der Kontainer und *MapEntry* enthält die eigentlichen Daten.

BashDo Das *BashDo* ist eine Form der Aktionsausführung. Dabei kann angegeben werden, welche Aktion für welches Betriebssystem ausgeführt werden soll.

Das Hauptelement im Metamodell ist das *Service*-Element. Es besteht aus der Angabe von Input- und Output-Daten. Des Weiteren kann eine Liste mit Parametern angegeben werden. Die Kernangabe in einem Service, der Befehl der ausgeführt werden soll, wird über *OperationSystemIndependentDo* oder *OperationSystemDependentDo* angegeben. An der Stelle können *JavaDo*s oder *BashDo*s definiert werden. Als letzte Angabe ist der Rückgabewert von dem Service wichtig. Dies erfolgt über das Element *ReturnType*, welches über den *FileReturn* auf eine Outputdatei zeigt.

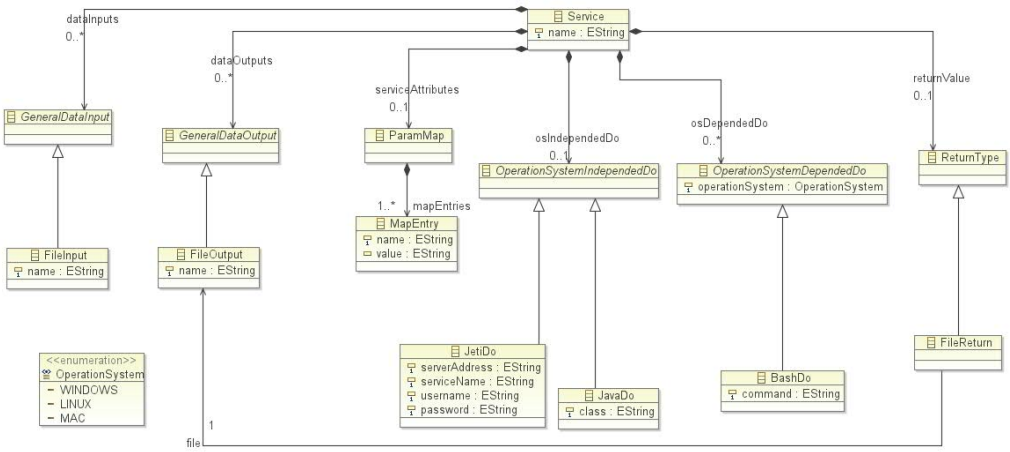

Abbildung 2.7.: Das Service-Metamodell

2.6. Workflow-Definition-Metamodell

Das entwickelte Metamodell stellt den Aufbau der Workflow-Definition dar. Die Abbildung 2.8 zeigt die grafische Repräsentation des Metamodells, welches im folgenden beschrieben wird.

WorkflowDefinitionType

Die wichtigste Komponente im Metamodell ist die *WorkflowDefinitionType*-Komponente. Sie wird benötigt, um die eigentliche Workflowinstanz zu verwalten und darzustellen. Aus diesem Grund gehen von dieser Instanz nur Aggregationsbeziehungen aus.

StartElement darf nur einmal existieren. Es definiert den Start des Workflows und enthält als einziges Attribut einen Namen. Des Weiteren hat das *StartElement* eine ausgehende Kante, die als *ControlFlowLink* definiert ist. Zu beachten ist, dass diese Kante wieder als *Containment* definiert ist. Damit verwaltet das *StartElement* diese Kante selbst. Da der Link an dieser Stelle erzeugt wird, ist eine Referenz von anderen Elementen nicht möglich.

Data wird benötigt, um allen Elementen, die Daten benötigen, eine Möglichkeit zu geben, auf diese zu referenzieren. Des Weiteren ist es möglich, auch Daten aus Datenbanken oder externen Services (wie zum Beispiel Web-Services) zu benutzen.

Service entspricht einem Service, der mit dem Service-Metamodell aus dem vorherigen Kapitel beschrieben wird. Der Service definiert dabei vor allem den eigentlichen Servicenamen. Dieser Servicename entspricht dem Namen, der in der Service-Definition angegeben wird (siehe 2.5). Um den richtigen Service bestimmen zu können, muss der

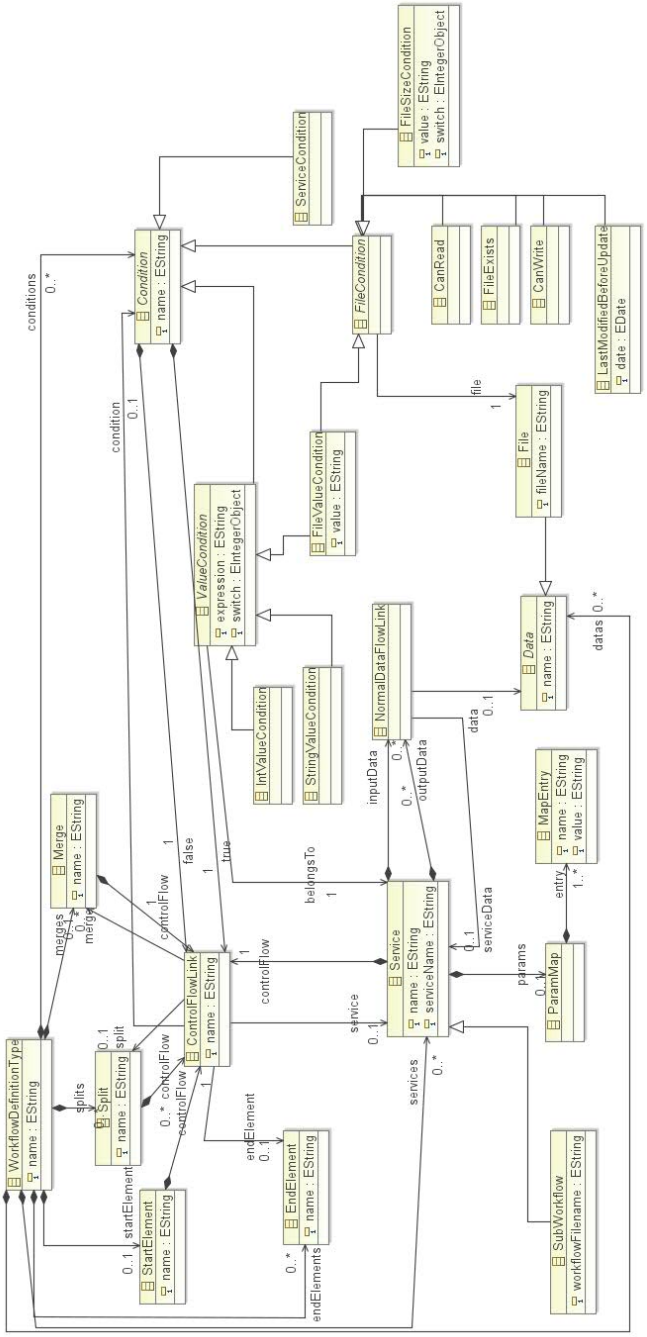

Abbildung 2.8.: Das Workflow-Metamodell

Name von dem Service mit dem Attribut *serviceName* definiert werden. Das Attribut *name* definiert nur den Namen der Komponente.

Condition -Elemente stellen die Bedingungen und Verzweigungen im Metamodell dar. Das Oberelement *Condition* unterteilt sich in die zwei Elemente *FileCondition* und *ValueCondition*. Das *FileCondition*-Element wird explizit für eine Datei definiert und auf dieser ausgeführt. Es unterteilt sich in die verschiedene Überprüfungen:

CanRead überprüft, ob die Datei gelesen werden kann.

CanWrite überprüft, ob die Datei geschrieben werden kann.

FileExists überprüft, ob die Datei existiert.

FileSizeCondition wählt über das Attribut *switch* verschiedene Modi aus. Diese Modi sind unter anderem die Definition ob der Wert größer, größer gleich, kleiner oder kleiner gleich sein soll. Durch das Attribut *value* kann angegeben werden, wie groß die Datei sein soll.

LastModifiedBeforeUpdate Dieses Element wird benutzt um anhand des Attributes *date* zu überprüfen, ob die Datei verändert worden ist.

Split teilt den Kontrollfluss in mehrere parallele Abläufe.

Merge führt die parallelen Kontrollflüsse wieder zusammen.

EndElement stellt das Ende des Kontrollflusses dar. In einem Modell können mehrere *EndElemente* existieren.

ControlFlowLink -Elemente zeigen auf alle Typen von Komponenten, die erreicht werden können. Dabei kann zu jedem *ControlFlowLink* der Name der Kante (als Beschreibung) und ein Wert angegeben werden. Die *ControlFlowLink*s werden benötigt, um die eigentlichen Kanten zwischen den Knoten abzubilden und um die Komponenten miteinander zu verbinden.

WorkflowDefinitionType -Elemente enthalten alle anderen Komponenten mit Hilfe der *Containments*. Es gibt zwei verschiedene Arten von Referenzen in einem *EMF*-Metamodell. Die normalen Kanten in dem Metamodell stellen ganz normale Referenzen auf die Komponenten dar. Dabei ist zu beachten, dass in diesem Fall die referenzierte Komponente in genau diesem Augenblick erzeugt wird. Eine *Containment*-Referenz bedeutet, dass diese Komponente schon irgendwann irgendwo erzeugt wurde und nur mit einem Zeiger auf diese Komponente gezeigt wird. Die Standardkante kann also mit einem Konstruktor aus der objektorientierten Programmierung verglichen werden.

Das Workflow-Metamodell wurde entwickelt um textuell und grafisch dargestellt zu werden. Im nächste Kapitel wird die Technologie für die textuelle Darstellung beschrieben. Anschließend folgt die der grafischen Repräsentation.

3. Textuelle Sprachen

Xtext [10] ist ein Framework für die Entwicklung von textbasierten domänenspezifischen Sprachen und kann zur Erstellung einer eigenen Programmiersprache verwendet werden. Es basiert auf dem Eclipse Modeling Framework, wodurch eine gute Integration in andere EMF-Frameworks möglich ist. Beispielsweise können Modelle aus Texten erzeugt oder Texteditoren generiert werden. Eclipse4Bio verwendet Xtext, um einen textuellen Editor für Service- und Workflow-Definitionen als Eclipse-Plug-in zu generieren. Der generierte Editor unterstützt folgende Funktionen:

- Hervorhebung von Syntaxkomponenten
- Code-Vervollständigung
- Anpassbares Outlining
- Validierungen zur Laufzeit
- QuickFixes

Die von den Metamodellen der Service- und den Workflow-Definition definierten verfügbaren Metainformationen werden von Xtext verwendet, um das syntaktische Modell der Sprachen genauer zu spezifizieren. Der Grund für die Benutzung des Xtext-Frameworks liegt in der schnellen und einfachen Entwicklung von domänenspezifischen Sprachen und der guten Integration in EMF.

3.1. Die Syntax

Es gibt in Xtext die Möglichkeit, im Workspace existierende Metamodelle zu importieren oder aus der Grammatik generieren zu lassen. In der Grammatik von Service- und Workflow-Definition wird jeweils ein Metamodell importiert. Des Weiteren ist die Xtext-Grammatik eine Sammlung von Regeln, die in Extended Backus–Naur Form (kurz EBNF) geschrieben werden [10] [11].

EBNF ist eine formale Metasprache, die dazu benutzt wird kontextfreie Grammatiken darzustellen.

EBNF-Ausdrücke können vier unterschiedliche Kardinalitäten haben, die bestimmen, wo und wie oft ein Element in der Sprache vorkommen darf:

- Operator | : Alternativen
- Operator ? : Optionalität
- Operator * : Optionalität oder Multiplizität
- Operator + : Multiplizität

Diese Kardinalitäten werden benötigt, um die folgenden Regeln zu definieren. Insgesamt gibt es nur zwei Arten von Regeln:

Terminal-Regeln beschreiben die Transformation einer Sequenz von Zeichen in einen Token, beispielsweise *terminal ID: ('a'..'z'|'A'..'Z'|'_ ')('a'..'z'|'A'..'Z'|'_ '|'0'..'9')*;*

Produktionsregeln beschreiben den Aufbau der Sprache. Dabei werden die Terminale und Nichtterminale zu komplexeren Statements zusammengefasst.

Die Regeln werden benötigt, um die Sprache zu definieren. Die Regeln beschreiben dabei die genaue Syntax und die Schlüsselwörter, die angegeben werden müssen. Xtext hat die Möglichkeit, Crosslinks zu deklarieren; dadurch entsteht eine Verknüpfung zu der Definition eines konkreten Objektes. Für die Crosslinks ist der Linker von Xtext zuständig. Er erkennt automatisch Angaben von Crosslinks, die auf Objekte zeigen, welche so nicht vorhanden sind (falscher Name, gelöschte Objekte und ähnliche Probleme). [10, Kapitel "Runtime Concepts"]. Xtext benutzt unter anderem die beiden Frameworks Xpand aus dem M2T Projekt [12] und die Modeling Workflow Engine 2 (kurz MWE2) aus dem Eclipse Modeling Projekt [13].

Xpand ist eine Templatesprache, die speziell auf einfache Textausgabe ausgerichtet ist. Die Syntax gleicht der Javasyntax mit zusätzlich eingeführten Befehlen zur Textausgabe, die während der Programmierzeit parallel in eine Java-Klasse kompiliert wird. Diese kann anschließend durch die Übergabe des Modells angesprochen werden und liefert den generierten Text zurück.

MWE2 beschreibt den Generator-Workflow, der von der *mwe2*-Datei im *generator*-Projekt repräsentiert wird. In dieser Datei wird bestimmt, wie und welche Modelle geladen werden müssen, wie sie validiert werden und wie genau der Code generiert wird.

3.2. Service-Definition Xtext-Editor

Der in Eclipse4Bio integrierte Editor, mit dem Services textuell beschrieben werden können, ist mit Hilfe von Xtext implementiert. Eine Servicebeschreibung definiert einen Service, der später für die Ausführung von Workflows verwendet werden kann. Diese Beschreibung basiert auf dem Service-Definition-Metamodell (siehe Kapitel 2.5 auf Seite 27).

In Listing 3.1 ist die Syntax-Definition eines Services zu sehen. Eine Servicebeschreibung fängt mit dem Schlüsselwort *Service* an und bekommt einen Namen zugewiesen. Anschließend können Angaben zu den Attributen und Dateien folgen. Die Sprache erlaubt danach entweder einen *OperationSystemIndependendDo* oder mehrere *OperationSystemDependedDo*s. Zum Schluss kann der Service optional ein Literal oder den Wert eines *FileOutput*-Elements zurückliefern.

```
'Service' name=EString
  (
  'attributes' serviceAttributes=ParamMap
    (
    'neededFile' '=' neededFile=[MapEntry] ';'?
    'generateFile' '=' generateFile=[MapEntry] ';'?
    )?
  )?
  (osIndependedDo=OperationSystemIndependedDo | osDependedDo+=
      OperationSystemDependedDo* )
  (fOutput=FileOutput)?
  ('returnValue' returnValue=ReturnType)?
```

Listing 3.1: Service-Definition in Xtext

3.3. Workflow-Definition-Xtext-Editor

Der ebenfalls auf Xtext basierende Editor für die textuelle Beschreibung von Workflows soll dazu dienen, die Workflows (welche später ausgeführt werden sollen) zu definieren. Dabei sollen die Services die vorher erzeugt worden sind orchestriert werden. Die daraus resultierende Definition soll dann als Grundlage dienen, damit die Engine die Services richtig ausführt. Dabei soll ein Workflow auf die Servicebeschreibungen zugreifen und konkrete Services referenzieren. Der Xtext-Editor für die Workflowbeschreibungen verwendet das Workflow-Metamodell.

Die textuelle Repräsentation eines Workflow (siehe Listing 3.2) besteht aus einem Namen und mehreren darauf folgenden Blöcken: *Data*, *attributes*, *Start*, *Node*s und *End*s. *Node* umfasst alle Elemente vom Typ *Service*, *Condition*, *Split*, *Merge*.

Zeilenumbrüche werden bei der Orchestrierung ignoriert und dienen nur der optischen Hervorhebung und Strukturierung von den Benutzer. Technisch gesehen ist auch die Definition in einer Zeile möglich. Eine solche Definition wäre jedoch nicht benutzerfreundlich, daher ist im Editor eine Autoformatierung implementiert. Ihr Ergebnis ist eine Formatierung der Komponenten, die je nach ihrer hierarchischen Position weiter nach rechts eingerückt werden.

```
'Workflow' name=STRING
  ('Data'
    (datas+=Data)+
  )?
  ('attributes'   attributes=AttributesMap)?
  startElement=StartElement
  (nodes+=Node)*
  (
  ( 'Ends' (endElements+=EndElement)*)   | endElements+=EndElement
  )
```

Listing 3.2: Workflow-Definition in Xtext

Das Element *Workflow* liefert ein Element vom Typ *WorkflowDefinitionType* aus dem Metamodell zurück. Ein Workflow benötigt einen Namen. Das Schlüsselwort *Data* erwartet mindestens ein Element vom Typ *Data*. Dies kann eine Input- oder Output-Datei sein. Des Weiteren benötigt der Workflow einen Start. Das *Start*-Element bestimmt den „ersten" *ControlFlowLink* des Kontrollflusses. Danach können eine Menge an Services, Conditions, Splits und Merges in beliebiger Reihenfolge definiert werden. Schließlich darf mehr als ein Ende des Workflows bestimmt werden. Das Erreichen mehrerer Endzustände kann durch eine Verzweigung im Workflow erreicht werden – mehr dazu im Kapitel 7 auf Seite 53.

3.4. Validierungen

Folgende Validierungen gelten sowohl für den textuellen, als auch für den grafischen Editor. Der Diagrammeditor verwendete in einer früheren Version für die Validierungen *EVL* [14]. Seine Validierungen basierten auf den gleichen Regeln wie die des Xtext-Editors, waren aber von ihnen getrennt implementiert. EVL konnte auf die Validierungen von Xtext nicht zugreifen. Gleiches galt für den umgekehrten Weg. Nachdem der grafische Editor neu implementiert wurde, wurden die Test mit dem entsprechenden Framework selbst umgesetzt (Graphiti, siehe Kapitel 4 auf Seite 38).

Xtext generiert für seine Sprache ein eigenes Package für die Validierungen. Dort befinden sich die angepassten Validierungmethoden. Diese Struktur ist beim Diagrammeditor nicht vorgegeben. Die aktuelle Version kann trotzdem auf diese Methoden zugreifen und sie manuell aufrufen. Xtext bietet auch automatische Validierungen für den eigenen Editor und sorgt bereits dafür, dass die Syntax gemäß der Grammatik korrekt ist.

Validierungen, welche über die von Xtext hinausgehen, erfolgen durch Eclipse4Bio-eigenen Java-Code. Die Validierungen werden meistens zusammen mit *QuickFixes* verwendet, durch deren Implementierung ein konkretes Problem behoben werden kann. Das Ergebnis einer durchgeführten Validierung kann einen Fehler oder eine Warnung zurückliefern.

Validierung einer Service-Definition

Folgende Validierungen werden für eine Service-Definition ausgeführt:

checkIfDoExists sichert, dass die Service-Definition entweder maximal ein *OperationSystemIndependentDo* oder mehrere *OperationSystemDependentDo*s besitzt.

checkBound gibt eine Warnung aus, wenn ein *OperationSystemDependentDo* nicht für alle unterstützten Betriebssysteme – Windows, Linux, Mac OSX – definiert ist.

checkCommand liest den Befehl in *BashDo* aus und überprüft ob alle vorkommenden Variablen definiert wurden.

checkBashCommandSingleOperationSystemEntry verbietet die mehrfache Definition für ein Betriebssystem.

Validierung einer Workflow-Definition

Die Validierungen einer Workflow-Definition erfolgen aus technischer Sicht ähnlich der Validierungen der Service-Definition. Im Folgenden werden Funktionen der wichtigsten Validierungen beschrieben:

checkDataFlowLink überprüft, ob jedes Datenflusselement genau ein Zielelement hat. Das Metamodell erlaubt einem *NormalDataFlowLink* auf *Service* und *Data* gleichzeitig zu zeigen, jedoch soll dies in einem Workflow verboten sein.

checkPathToElements überprüft per Floyd-Warshall-Algorithmus [17], ob ein Pfad zu jedem Knoten existiert und gibt im negativen Fall auch die Elemente, bei denen ein Erreichbarkeitsproblem vorliegt, aus. Dies ist wichtig, da jeder korrekt definierte Workflow terminieren soll, um am Ende ein Ergebnis zu liefern. Ein Workflow ist somit ungültig, wenn er – abgesehen von der Laufzeit – kein Endelement erreichen kann. Das kann passieren, wenn entweder kein Endelement definiert ist, oder wenn im aktuellen Pfad kein Ende-Element erreicht werden kann.

4. Diagrammeditor

Die texuelle Repräsentation sowie die Transformation wurden beschrieben. An dieser Stelle wird auf die grafische Darstellung eingegangen. Die GUI (Graphical User Interface) ist für die Darstellung und Interaktion der Komponenten innerhalb der Eclipse-Oberfläche zuständig. Eclipse4Bio-Workflows können mit Hilfe der Eclipse4Bio-Anwendung grafisch modelliert werden, wobei sich die Anwendung aus mehreren Teilkomponenten zusammensetzt. Zentrales Element ist der grafische Editor, mit dem die Workflows durch Drag-and-Drop von Knoten und Kanten aus einer Seitenleiste zusammengesetzt werden können. Die Komponenten „Arbeitsfläche“ und „Seitenleiste“, wurden in Eclipse4Bio zunächst mit Hilfe automatischer Codegenerierung aus den Metamodellen der Workflows erzeugt. Im Laufe der Projektarbeit wurden mehrere Ansätze zur Generierung bzw. Entwicklung von grafischen Workfloweditoren betrachtet und zunächst *EuGENia* [50] ausgewählt. Nach längerer Arbeit mit *EuGENia* stellte sich heraus, dass der generierte Code nur schwierig erweiterbar war. Aus diesem Grund wurde *Graphiti* [53] für die Entwicklung des grafischen Workfloweditors gewählt.

Dieser Abschnitt beschäftigt sich mit der Generierung und der Entwicklung der grafischen Oberfläche. Es werden die Tools vorgestellt, die für diesen Prozess zur Auswahl standen. Dabei handelt es sich einerseits um fertige *Workflow-Editoren*, andererseits um Tools zur Erzeugung grafischer Editoren im Eclipse-Kontext. Zuerst wird das *Java Workflow Tooling* (JWT) betrachtet [54], gefolgt von *EuGENia* [50], dem *Graphical Modeling Framework* [52] und *Graphiti* [53].

4.1. Java Workflow Tooling

JWT ist ein Workfloweditor Framework, der an der Universität Augsburg in Zusammenarbeit mit der *eMundo* GmbH entwickelt wird [15, 63]. Es bietet einen fertigen Workfloweditor, mit dem Workflows grafisch modelliert und validiert werden können. Hierbei fokussiert sich JWT nicht nur auf eine Beschreibungssprache und eine bestimmte Engine, sondern versucht eine Brücke zwischen verschiedenen bestehenden Workflowsystemen zu bauen. Unter anderem stehen Modellierungselemente wie *Rolle*, *Applikation* und *Daten* zur Verfügung.

Ein Ziel des Projekts ist es, die Möglichkeit zu bieten Workflows in verschiedene Definitionssprachen zu übersetzen. Um dies zu ermöglichen, bilden die *Transformations* ei-

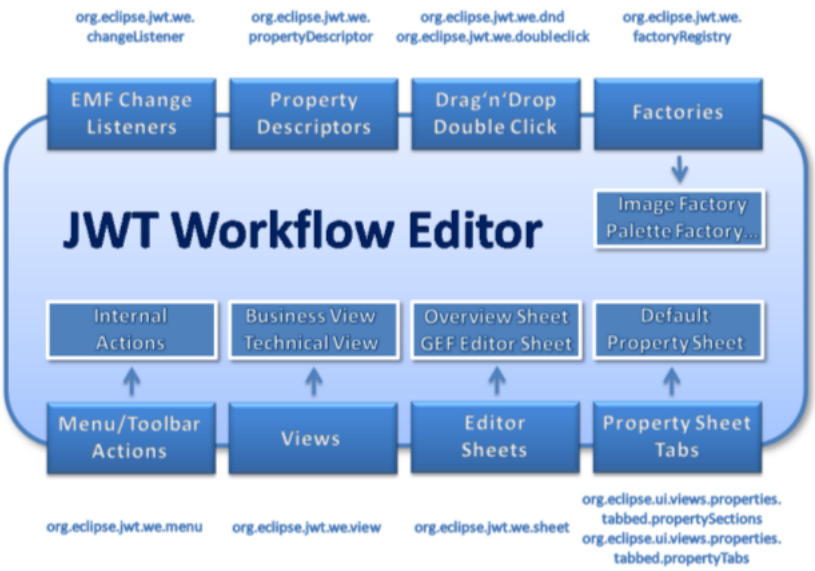

Abbildung 4.1.: Erweiterungspunkte des Java Workflow Tooling. Quelle: [16]

ne wichtige Komponente des Projekts. Diese ermöglichen den Export eines Workflows in *XPDL* (XML Process Definition Language) [67] und die Transformationen zu *BPMN* [28], *jPDL* (XML Prozessdefinitionssprache für die jBPM Engine) [21] und *STP-IM* (STP Intermediate Metamodel). Zur Überwachung der durch die Workflows definierten Prozesse bietet JWT die „Workflow Administration and Monitoring"-Komponente. Mit dieser können die Prozesse zur Laufzeit unter verschiedenen Engines beobachtet werden. Weiterhin stellt JWT einen systemunabhängigen Simulator bereit, der es ermöglicht, die Workflows auszuführen, ohne eine bestimmte Engine zu benutzen.

JWT besitzt Erweiterungspunkte, an denen das Tool erweitert werden kann. Viele davon beziehen sich auf die grafische Oberfläche. Unter anderem können verschiedene Ansichten erstellt und das Menü erweitert werden. Eine Übersicht über die Erweiterungsmöglichkeiten bietet Abbildung 4.1. Des Weiteren bietet JWT auch die Möglichkeit, das Metamodell zu erweitern.

Obwohl JWT auf den ersten Blick den Eindruck hinterlässt, dass es ein generisches Tool mit einfachen Erweiterungsmöglichkeiten ist, gibt es bei der Arbeit einige Probleme. Obwohl der von JWT angebotene Simulator plattformunabhängig sein soll, kann dieser nicht unter Mac OSX installiert werden. Ein weiteres Problem besteht in der eingeschränkten Erweiterbarkeit der *Applikationen*, die durch JWT zur Verfügung gestellt werden. JWT bietet zwar ein *Adapter Framework*, durch welches Programme als Applikation eingebunden werden können, jedoch wird in der Dokumentation angegeben, dass es sich dabei um Standardprogramme wie Word und Firefox handelt. Bei dem Versuch, eigene Applikationen zu implementieren, scheitert das Vorhaben daran, dass Komponenten des Adapterframeworks

bei der Implementierung nicht gefunden werden. Auf der SourceForge-Seite des Projektes gibt es einen Ordner „AgilPro Adapter“, der jedoch leer ist und in der Dokumentation zum Adapterframework lässt sich keine Lösung für dieses Problem finden [64].

4.2. Epsilon: EuGENia

EuGENia [50] ist ein Tool, das zum *Epsilon*-Framework [57] gehört. Epsilon bietet Programmiersprachen, die Interaktionen mit EMF-Modellen ermöglichen. Dabei soll die modellgetriebene Entwicklung unterstützt werden. Zu den Funktionalitäten des Epsilon - Frameworks gehören unter anderem Model-to-Model-Transformationen, Validierung von Modellen und Codegenerierung. EuGENia erlaubt es aus Metamodellen grafische Editoren zu generieren.

Es baut auf dem *Eclipse Graphical Modeling Framework* (GMF) auf [52] und erzeugt die für den Editor benötigten „.gmfgraph“-, „.gmftool“- und „.gmfmap“-Dateien. Dies stellt eine Arbeitserleichterung dar, da die Komplexität von GMF verdeckt und die Entwicklung eines Editors vereinfacht wird. Mit EuGENia wird somit Neueinsteigern geholfen, ihren ersten Editor zu entwickeln, aber auch Fortgeschrittenen genügend Funktionalität geboten, um ihre Ziele zu erreichen.

Um einen Editor aus einem Metamodell zu generieren, muss das Ecore-Modell, bzw. eine andere Darstellung des Modells, durch spezifische Annotationen erweitert werden. Eine übersichtliche Darstellung des Ecore-Modells bietet sich in Form der *Emfatic*-Repräsentation an [56]. Dies ist eine textuelle Repräsentation, die sich am Klassenkonzept orientiert. Dabei werden die Modellelemente als Klassen dargestellt. Ein Beispiel befindet sich im Anhang in Listing A.2. Dort ist ein kleiner Ausschnitt aus einem annotierten Ecore-Modell in der Emfatic-Notation dargestellt. Modellannotationen, die EuGENia nicht beachtet, sind aus dem Listing entfernt worden.

Im Folgenden werden Beispiele für die EuGENia Annotationen beschrieben:

@gmf.node(···) mit folgenden Parametern

- label=“name“
- figure=“ellipse“
- size=“50,50“
- color=“0,255,127“

sagt aus, dass die annotierte Klasse im Diagramm als Knoten dargestellt werden soll. Die Form des Knotens soll eine Ellipse der Größe „50,50“ Pixel sein und die Farbe haben, die durch den RGB-Wert „0,255,127“ kodiert wird.

@gmf.link(···) mit beispielweise folgenden Parametern:

- source=“controlFlowBiEnd“

- target=“controlFlowLinkEnd“
- target.decoration=“arrow“
- style=“solid“
- label=“value“

beschreibt eine Kante im späteren Diagramm. Durch „source“ und „target“ wird definiert von welchem Typ der Anfangs- und Endknoten der Kante ist. Die Darstellung der Kante wird hier durch „target.decoration“ und „style“ bestimmt. In diesem Fall wird die Kante durch einen Pfeil dargestellt.

Zusätzlich muss in dem Metamodell eine Klasse angegeben werden, die als übergeordnetes Objekt dient. Diese übergeordnete Klasse muss durch

@gmf.diagram(foo=“bar“) annotiert werden. Emfatic verlangt im momentanen Entwicklungsstadium Parameter, wodurch der Parameter „foo“ mit Wert „bar“ zustande kommt. Dieser hat jedoch keine weitere semantische Bedeutung. In diese Klasse werden die Klassen aufgenommen, die später im Diagrammeditor darstellbar sein sollen.

Diese unkomplizierte Art der Editorgenerierung hat jedoch auch Nachteile. EuGENia kann nicht alle Funktionen bereitstellen, die im GMF enthalten sind. Andernfalls könnte die hohe Komplexität von GMF nicht versteckt werden. Ein weiteres Problem von EuGENia ist die manuelle Bearbeitung von generiertem Code. Wird der Code bearbeitet und der Editor neu generiert, so werden die Änderungen nicht übernommen. Abhilfe verschafft in diesem Fall die *Epsilon Object Language* (EOL) [49]. In dieser Sprache können Skripte geschrieben werden, die nach jeder Generierung ausgeführt werden. Dies ist möglich, da EuGENia selbst auf Grundlage dieser Sprache arbeitet. Um den Workfloweditor nach der Generierung zu bearbeiten, werden zusätzliche Kenntnisse in EOL benötigt.

Ein anderes Problem ergibt sich dadurch, dass Kanten, die im späteren Editor dargestellt werden sollen, im Metamodell durch Klassen modelliert werden und bidirektionale Beziehungen angegeben werden müssen. Dadurch ist es schwierig das anfängliche Metamodell so zu annotieren, dass ein korrekter Diagrammeditor generiert werden kann. Mögliche Lösungen sind, das in EuGENia annotierbare Metamodell als grundlegendes Modell zu übernehmen bzw. zwei Metamodelle zu verwalten. Die Variante, ein Metamodell – das durch EuGENia annotierbar ist – als Grundlage zu nehmen, hätte eine reduzierte Benutzerfreundlichkeit zur Folge. Wie in Kapitel 3 beschrieben, wird zur Entwicklung des textuellen Editors ein Metamodell als Grundlage benutzt. Wäre dieses Metamodell das durch EuGENia annotierbare Modell, würden in der Syntax der textuellen Beschreibung die bidirektionalen Kantenbeziehungen abgebildet werden. Das hat zur Folge, dass in jedem Knoten, der eine ausgehende Transition besitzt, der Quell- und Zielknoten dieser Transition angegeben werden muss. Der Quellknoten ist jedoch immer der Knoten, der gerade definiert wird, sodass die zusätzliche Angabe des Quellknotens der Transition ein Overhead

ist. Daher verwaltete eine frühe Version von Eclipse4Bio zwei Metamodelle. Die Entwicklung von Transformationen zwischen beiden Modellen erwies sich jedoch als schwierig, wie in Abschnitt B.2.1 dargestellt.

Im Verlauf der Arbeit mit *EuGENia* ergaben sich weitere Probleme. Um den generierten Workfloweditor weiter anpassen zu können, musste der generierte Code verändert werden. Erweiterungen durch EOL sind eingeschränkt, da die Sprache an sich sehr eingeschränkt ist. Es können beispielsweise nur triviale Datentypen wie *String*, *Integer*, *Real* und *Boolean* nutzen. Möchte man aber unabhängig davon komplexe Datentypen nutzen, muss man diese extern, z.B. über Java definieren. Ein weiteres Problem war, dass sich die Ansicht für die Eigenschaften des übergeordneten Objekts nicht korrekt darstellen ließ.

4.3. Graphical Modeling Framework

Das *Graphical Modeling Framework* (GMF) ist ein Framework, das die Aufgabe besitzt, funktionsfähige grafische Editoren zu generieren [52]. Hierzu benötigt GMF zwei weitere Frameworks, das *Eclipse Modeling Framework* (EMF) und das *Graphical Editing Framework* (GEF). Das EMF stellt dem GMF das nötige Metamodell, das Ecore-Modell, zur Verfügung. Das durch GEF erzeugte Ecore-Modell ermöglicht andere Modelle zu generieren. Das GMF verlangt das Ecore-Modell, um daraus mit Hilfe des zweiten Frameworks, dem GEF, einen grafischen Editor erzeugen zu können, allerdings müssen vor der Entstehung des grafischen Editors weitere Modelle erzeugt werden. Es ist nicht auf direktem Weg möglich vom Ecore-Modell einen grafischen Editor, den Diagrammeditor, zu erstellen. Daher muss im ersten Schritt aus dem Ecore-Modell ein *EMF-Genmodel* generiert werden. Beide Modelle steuern dazu bei, den Java-Code zu erstellen.

Zwei weitere wichtige Modelle sind das *Graphical-Notation-Model* und das *Tool - Definition - Model*. Das Graphical-Notation-Model ist für die Beschreibung der Figuren, die im grafischen Diagrammeditor zu sehen sein werden, zuständig und das Tool-Definition-Model für die Beschreibung der Werkzeuge, denKnoten- und Kantentypen, im Diagramm. Zur Verknüpfung beider wird ein weiteres Modell benötigt, das *Mapping-Model*, mit dem das direkte Vorgängermodell, das *GMF-Gen-Model*, erstellt wird. Dieses hat die Endung „.diagram“. Aufgrund des Ecore-Models kann der grafische Editor erst nach dem Erzeugen von Zwischenmodellen generiert werden. Abbildung 4.2 zeigt die Abhängigkeiten zwischen den verschiedenen Modellen im Graphical Modeling Framework.

4.4. Graphiti

Mit Graphiti wird eine weitere Möglichkeit geschaffen, graphische Editoren für EMF-Modelle zu entwickeln. Im Gegensatz zu EuGENia, das einen Diagrammeditor aus den

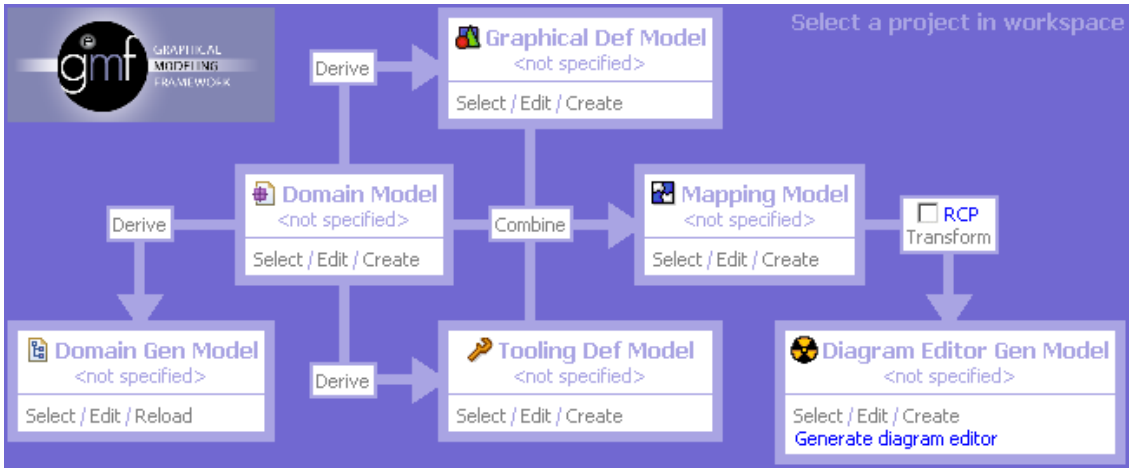

Abbildung 4.2.: Modelle im Graphical Modeling Framework [52]

Annotationen des Metamodells erstellt, ist Graphiti auf Java-Programmierung fokussiert. Graphiti bietet eine Java-API, mit der grafische Editoren erstellt werden können.

Ein Ziel des Graphiti-Frameworks ist es, Draw2D und GEF zu verstecken. Bei Draw2D handelt es sich um ein leichtgewichtiges Toolkit, mit dem grafische Komponenten im SWT-Stil dargestellt werden können. Dadurch, dass Graphiti die beiden Komponenten versteckt und der Benutzer hauptsächlich mit der Graphiti-API arbeitet, wird ihm der Aufwand, sich in Draw2D und GEF einzuarbeiten, abgenommen. Um einen grafischen Editor zu erstellen, wird nur Wissen über das Metamodell und Graphiti benötigt. Die Funktionalität der Editoren wird mit Graphiti durch sogenannte *Features* definiert. So gibt es beispielsweise *Create Features*, mit denen neue Modellobjekte erstellt werden und *Add Features*, die schon existierenden Modellobjekten eine grafische Repräsentation hinzufügen. Mit weiteren Features wird das Verhalten der Objekte beim Editieren, Bewegen und Löschen definiert. Weiterhin gibt es die Möglichkeit, weitere Features für verschiedene Eigenschaften bzw. Verhaltensweisen der Objekte zu implementieren [58].

Der Aufwand, einen Diagrammeditor mit Graphiti zu erstellen, ist aufgrund der Komplexität Graphitis im Vergleich zu EuGENia größer. Was in EuGENia mit kurzen Annotationen möglich ist, muss in Graphiti mit vielen Zeilen Code programmiert werden. Daher wurde Graphiti in Eclipse4Bio anfangs nicht verwendet. Nachdem es Probleme mit der Erweiterung des Workfloweditors gab, der mit EuGENia erstellt wurde, fand der Wechsel zu Graphiti statt. Durch den Einsatz von Graphiti ist das Verwalten von zwei Metamodellen nicht mehr nötig, da in Graphiti Kanten im Metamodell keine bidirektionale Beziehung benötigen.

Um einen Graphiti-Editor zu erzeugen, muss ein *DiagramTypeProvider* und ein *FeatureProvider* implementiert werden. Der DiagramTypeProvider stellt das Diagramm dar und der FeatureProvider liefert die Funktionalität der grafischen Objekte des Diagramms. Die Implementierung eines Features besteht grundsätzlich aus zwei Methoden. Die erste Me-

thode – die „canExecute“-Methode – muss definieren, wann das Feature ausgeführt werden kann. In der zweiten Methode – der „execute“-Methode – wird definiert, was ausgeführt werden soll. In Listing A.3 und A.4 ist das „create-feature“ und „add-feature“ eines Service dargestellt. Für jedes Metamodellobjekt, das im Editor dargestellt werden soll, muss ein Create- und Add-Feature definiert werden. Das Erstellen von Kanten folgt dem gleichen Prinzip. Im Create-Feature einer Kante muss sichergestellt werden, dass die Kombination aus Quell- und Zielknoten der neuen Kante dem Metamodell nicht widerspricht.

Je nach Feature müssen weitere Methoden implementiert werden. Im „Update-Feature“ muss beispielsweise die Methode „updateNeeded“ implementiert werden, die definiert, wann ein Update eines Objekts nötig ist.

5. Oberflächenkomponenten

Neben dem eigentlichen Arbeitsbereich für die Modellierung von Workflows existiert in Eclipse4Bio eine Seitenleiste zur Verwaltung der Projektressourcen sowie eine *Property-View*. Die Seitenleiste, Navigator genannt, zeigt die zu den Workflowprojekten gehörenden Dateien in baumförmiger Struktur an. Der Navigator erzeugt und verwaltet die Verzeichnisstruktur dieser Projekte. Zu einem Workflowprojekt gehören neben den eigentlichen Workflowdateien auch die von den Workflows genutzten Servicedateien. Die Property-View stellt die Eigenschaften der Knoten eines Workflows dar. Diese Eigenschaften können in der Property-View editert werden.

5.1. Navigator

Der Navigator ist ein Teil der grafischen Oberfläche, die dem Benutzer die Projektstruktur seiner aktuell existierenden Workflowprojekte anzeigt. Ebenfalls können hier Dateien, beispielsweise Workflowprojekte, Workflow- und Servicedateien, in den Projektordnern gelöscht oder hinzugefügt werden. Hierbei müssen folgende Eigenschaften eingehalten werden:

- Ein Workflowprojekt darf nicht innerhalb eines anderen Projektes erzeugt werden.
- Außerhalb von Workflowprojekten dürfen keine Workflow- und Servicedateien erstellt werden.
- Nach der Generierung eines Projektes darf die Grundstruktur des Projektes nicht verletzt werden, also kein Projektordner darf gelöscht werden .
- Das Löschen der Workflow- und Servicedateien soll möglich sein.
- Die Generierung der Workflow- und Servicedateien darf nur in dem für sie reservierten Projektordner stattfinden.
- Die Erzeugung der Projekte und Dateien ist auch über ein Popup-Fenster, das beim Anklicken auf ein Projekt erscheint, möglich.

Der Navigator ermöglicht dem Nutzer, innerhalb eines Workflowprojektes mehrere Workflowdateien und/oder mehrere Servicedateien zu verwalten. Dazu können in den entsprechenden Projektordnern Ordner hinzugefügt werden. Das kann zum Beispiel dazu dienen,

weitere Servicedateien eines Institutes in einem Ordner, zum Beispiel innerhalb des Projektordners „Services“, zusammenzufassen und die Übersicht über die vorhandenen Servicedateien zu vereinfachen. In Abbildung 5.1 wird ein Beispiel zur inneren Projektstruktur angezeigt.

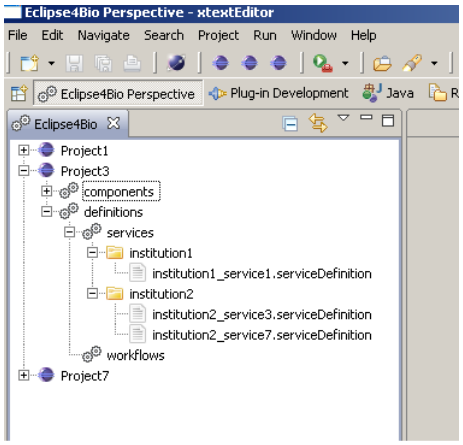

Abbildung 5.1.: Innere Projektstruktur

5.1.1. Bedienung

Das Navigator-Fenster kann durch das Anklicken der View „Eclipse4Bio“ aktiviert werden. Diese Perspektive-View befindet sich unter *Window/Show View/Others*.

Neben der Menüleiste der Anwendung kann auch über den Navigator ein Projekt, eine Workflowdatei oder auch eine Servicedatei erzeugt werden. Es besteht die Möglichkeit durch einen Rechtsklick im Navigator-Fenster ein Popup-Menü zu öffnen. Mit Hilfe der Funktion *New* werden Elemente angezeigt, die erzeugt werden können. Diese sind *Workflow-Project*, *Service-File*, *Workflow-File* und weitere auswählbare Elemente über *Others*. Abbildung 5.2 zeigt das Popup-Menü.

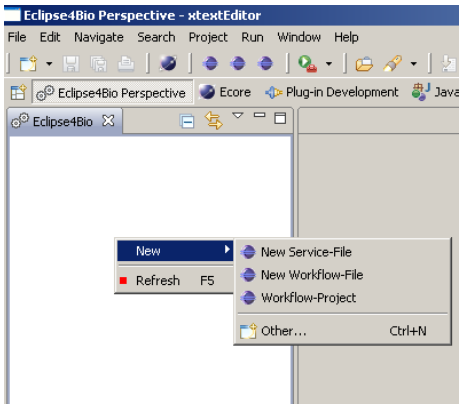

Abbildung 5.2.: Popup-Menü

Bei der Auswahl des Workflowprojekts erscheint ein Fenster, in das ein Name für das Projekt eingetragen und daraufhin bestätigt werden muss. Das Erzeugen eines Projektes innerhalb eines anderen Projektes und das Erzeugen einer Workflow- oder Servicedatei außerhalb eines Projektes ist nicht möglich.

5.1.2. Projekt exportieren

Die Export-Funktion ermöglicht dem Benutzer, ein Projekt aus Eclipse4Bio zu exportieren und in einer anderen Umgebung mit der Stand-Alone-Variante (siehe Kapitel 7 auf Seite 53) der Engine auszuführen. Beim Exportieren wird eine ZIP-Datei erstellt, die alle projektrelevanten Dateien beinhaltet. Eine Besonderheit der Export-Funktion ist das „Diff & Merge"-Prinzip, mit der die gloablen Services mit den lokalen Projekt-Services vermischt werden. Die lokalen Services erhalten eine höhere Priorität, so dass ein vorhandener Service aus dem Service-Repository mit den Projekt-Service überschrieben wird, sofern die Servicenamen identisch sind. Die generierte ZIP-Datei kann an einem belieben Ort entpackt und ausgeführt werden. Hierfür werden der Engine die benötigten Informationen als Parameter übergeben.

5.2. Property-View

Eclipse liefert standardmäßig eine Property-View, die – für einen mit *EuGENia* generierten Workfloweditor – alle Eigenschaften eines Knotens darstellt und editierbar macht. Dazu gehören auch Eigenschaften wie ausgehende Kanten eines Knotens oder der *serviceName* eines Service-Knotens. Die Anzeige der ausgehenden Kanten eines Knotens ist für den Benutzer des Editors redundant, da die Kanten im Editor grafisch dargestellt werden. Der *serviceName* eines Knotens darf angezeigt werden, soll aber nicht editierbar sein, nachdem der Service erstellt wurde. Ein mit Graphiti erstellter Workfloweditor stellt keine Property-View bereit. Die standardmäßige Property-View von Eclipse kann eingebunden werden, muss aber auf Grund der oben genannten Nachteile angepasst werden. Deswegen wurde für Eclipse4Bio eine Property-View implementiert, die den benötigten Anforderungen genügt.

Um im ersten Ansatz eine Property-View zu erstellen, wurde das *Extended Editing Framework* (EEF) [51] benutzt. EEF generiert auf Grundlage des Metamodells eine Property-View. Dabei wird aus der *.genmodel*-Datei eine *.components*- und *.eefgen*-Datei erzeugt. Durch die *.components*-Datei kann die später aus der *.eefgen*-Datei generierte Property-View angepasst werden. Standardmäßig wird für jedes Metamodellelement eine eigene View generiert, die die Eigenschaften dieses Objekts anzeigt. Hierbei entstehen zwei Probleme. Das erste Problem ist – wie schon angesprochen – dass alle Eigenschaften angezeigt und editierbar sind. Das zweite Problem ist, dass Eigenschaften von referenzierten Elementen in der View des referenzierenden Elements nicht angezeigt werden. Diese Probleme sollen durch Anpassen der *.components*-Datei gelöst werden können. Bei Eclipse4Bio ergaben sich

Abbildung 5.3.: Property View für den WorkflowDefinitionType

Abbildung 5.4.: Property View eines Service

aber die Probleme, dass nach dem Löschen von Views der Code nicht fehlerfrei generiert werden kann und Views, die auf andere Views referenzieren, die Werte der referenzierten Eigenschaften nicht aus dem Modell lesen bzw. in das Modell schreiben können.

Im zweiten Ansatz wurde für jedes Metamodellelement eine GUI-Darstellung der Eigenschaften erstellt, die in einer View angezeigt wird, wenn das korrespondierende Element fokussiert ist. Die GUI wurde mit Hilfe des *Google Web Toolkit* (GWT) erstellt [19]. GWT bietet einen grafischen Editor, mit dem durch *Drag & Drop* eine GUI erstellt werden kann. Der durch das GWT generierte Code wurde so erweitert, dass die Property-View die gewünschte Funktionalität bereitstellt. In den Abbildungen 5.3, 5.4 und 5.5 sind die property views des Containerelements, eines Service und einer Condition dargestellt.

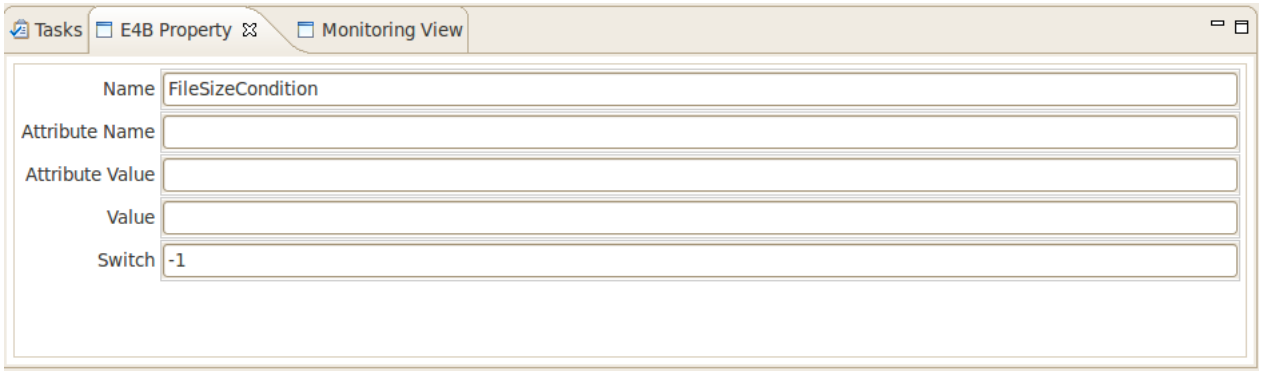

Abbildung 5.5.: Property View einer Condition

6. Build-Prozess

Zur Auslieferung von Eclipse4Bio ist es nötig, aus den Einzelkomponenten, möglichst automatisiert, eine lauffähige Anwendung – das Produkt – zu erstellen. Dieser sogenannte „Build-Prozess“ erzeugt aus den entwickelten Plugins, zusammen mit den benötigten Eclipse-Komponenten, ein möglichst kompaktes Produkt, um die Datenmenge des Programms gering zu halten. Eine weitere Anforderung ist die Kompatibilität mit gängigen Betriebssystemen. In diesem Kapitel werden drei Ansätze vorgestellt, mit denen sich diese Anforderungen erfüllen lassen: Das *Eclipse Plug-in Development Environment (PDE)*, *Maven Tycho* und eine *Eclipse Update-Site*. Die Betrachtung der Ansätze ergab, dass das Eclipse Plug-in Development Environment den Anforderungen der Projektgruppe am meisten gerecht wurde. Daher wird es für den Build-Prozess von Eclipse4Bio eingesetzt.

6.1. Eclipse PDE

Das Eclipse Plug-in Development Environment [60] ist eine Sammlung von Werkzeugen zum Erstellen, Entwickeln, Testen, Debuggen, Kompilieren und Ausliefern von Eclipse-Plug-ins, -Features, -Update-Sites und -RCP-Produkten. Diese Funktionen sind in drei weitgehend unabhängige Hauptkomponenten von Eclipse PDE eingeteilt: *Build*, *UI* und *API Tools*.

Im Folgenden wird näher auf die Verwendung von *PDE/Build* eingegangen, um ein lauffähiges Produkt für Eclipse4Bio zu erstellen. PDE/Build [66] ist ein auf Apache Ant basiertes Build-System für Ecplipse RCP-Anwendungen und Plugins. Es gehört zu den sogenannten *headless*-Systemen, da es zur Erstellung eines Produktes keine vollwertige Eclipse-Umgebung benötigt. Um das Produkt mit PDE/Build zu erstellen, muss eine „.product“-Datei für das Projekt erzeugt werden. In dieser Datei wird definiert, ob das Projekt auf *Plug-ins* oder *Features* basiert. Außerdem werden in der „.product“-Datei weitere Abhängigkeiten definiert, die für das Projekt benötigt werden. Diese Abhängigkeiten sind die Plugins, die zu dem Projekt gehören und weitere Abhängigkeiten, die wiederum für diese Plugins benötigt werden.

PDE/Build kann jedoch vereinzelt Abhängigkeiten nicht auflösen. Teilmengen der erstellten Plugins, die zum Projekt gehören – beispielsweise Plugins, die den Xtext-Teil des Projektes definieren –, können unregelmäßig beobachtet nur erzeugt werden, wenn sie auf Plugins aufgebaut waren. Bei anderen passiert dies wiederum, wenn Features als Grundlage

benutzt wurden. Ein weiteres Problem ergibt sich dadurch, dass das Projekt durch die benötigten SWT-Komponenten nicht mehr plattformunabhängig ist, wodurch die Erstellung des Build-Prozess für die drei Zielplattformen weiter erschwert wird.

Durch die Benutzung sogenannter „Deltapacks“ [66] ist es jedoch möglich, mit PDE/Build von einem Betriebssystem aus lauffähige Versionen für eine Vielzahl von Betriebssystemen zu erzeugen. Ein Deltapack enthält die plattformspezifischen Komponenten und Plugins, die benötigt werden, um ein Produkt für fremde Plattformen zu generieren. Dazu gehört auch die Unterstützung von betriebssystemabhängigen „Brandings“, wie dem Symbol für das Programm-Icon. Ausgabe eines Build-Prozesses mit PDE/Build und einem Deltapack sind mehrere getrennte Versionen von Eclipse4Bio für die Betriebssysteme Windows, Linux und Mac OSX – jeweils in einer 32-Bit- und einer 64-Bit-Version. Die Unterstützung weiterer Betriebssysteme, etwa Solaris, lässt sich bei Bedarf dem Projekt hinzufügen.

6.2. Maven Tycho

Maven Tycho [62] ist eine Sammlung von Maven [46] Plug-ins und Erweiterungen mit dem Ziel, Eclipse Plug-ins und sogenannte OSGi-Bundles zu erstellen. Es nutzt die in Eclipse Plug-ins und OSGi-Bundles enthaltenen nativen Metadaten während des Buildprozesses und unterstützt – ähnlich PDE/Build – Bundles, Features, Fragments, Update-Sites und RCP-Anwendungen.

Maven Tycho kann Projekte für verschiedene Plattformen erstellen und Abhängigkeiten automatisch auflösen. Dafür werden für die Plugins sogenannte „.pom“-Dateien (Project Object Model) erzeugt. Jedes Plugin enthält eine .pom-Datei, die auf eine „Parent pom“ verweist. In der „Parent pom“ sind die wichtigsten Informationen eingetragen. Beispielsweise wurde in früheren Tycho-Versionen dort hinterlegt, für welche Betriebssysteme das Projekt gebaut werden soll. Die aktuelle Version von Maven Tycho ist dazu übergegangen, die Zielsysteme über eine Eclipse-spezifische „.target“-Datei zu definieren. Tycho kann jedoch auch diverse Abhängigkeiten nicht auflösen und ein Entfernen dieser Abhängigkeiten führt zur Erstellung fehlerhafter und nicht ausführbarer Programme.

6.3. Eclipse Update-Site

Eclipse Update-Site ist eine Sammlung von Eclipse Plug-ins [48]. Eine Update-Site kopiert alle notwendigen Plugins in JAR-Dateien. Das Produkt kann daraufhin unter Eclipse installiert werden, indem die Dateien aufgerufen werden. Dafür muss nur der Pfad zu den Dateien bekannt sein.

Für die Eclipse4Bio Update-Site existieren zwei Plugins:

de.tu_dortmund.cs.pg.eclipse4bio.xtextEditor.feature enthält alle Plugins, in denen nur der Code für die Xtext- und Diagramm-Editoren liegt.

de.tu_dortmund.cs.pg.eclipse4bio.xtextEditor.platform.feature enthält alle Plugins, die benötigt werden um eine DSL auf einer RCP-Plattform auszuführen.

Nachteile dieser Variante sind, dass die Update-Site nicht selbstständig ausgeführt werden kann und immer die Eclipse-IDE als Grundlage benötigt. Außerdem ist eine manuelle Bestimmung der minimalen Abhängigkeiten für die Installation schwierig. Daher kann es passieren, dass für eine bestimmte Eclipse-Umgebung zu viele oder zu wenige Plugins in den Features definiert sind.

7. Anforderungen an die Workflowausführung

Die Workflowmodelle, die im Eclipse4Bio-Projekt erstellt werden, wurden in den vorherigen Kapiteln ausreichend beschrieben. Außerdem wurde erläutert, wie ein Workflow grafisch dargestellt und erzeugt werden kann. Diese Workflows müssen nun noch ausgeführt werden. Die Hauptaufgabe der Eclipse4Bio-Engine ist es deshalb, einen Workflow auszuführen, wie er in Kapitel 2.6 definiert und vorgestellt wird. Neben der offensichtlichen Aufgabe der reinen Ausführung stellt sich die Frage, in welcher Form die Ausführung stattfindet. Es sollen drei Varianten zum Einsatz kommen, die im folgenden beschrieben werden.

Die RCP-Integration ist eine direkte Integration in das Endprodukt. Der Benutzer soll die Möglichkeit haben, seine Workflows direkt aus der Applikation heraus zu starten. Dies ist insbesondere dann sehr hilfreich, wenn man kleinere Workflows lokal ausführen möchte. Zusätzliche Schnittstellen ermöglichen die Interaktion der GUI mit den Komponenten der Engine, dazu gehören insbesondere das Starten und Stoppen der Ausführung. Eine weitere wichtige Komponente ist das Monitoring, womit die Ausführung überwacht werden kann. Die Monitoringfunktionen fungiert nach dem Observer-Pattern-Design [18] und kann damit auch von außerhalb angesprochen werden.

Die Stand-Alone-Variante beinhaltet lediglich die Engine, sowie alle benötigten Bibliotheken, die für ihre Ausführung erforderlich sind. Der Vorteil dieser Stand-Alone-Variante ist, dass man seinen zuvor definierten Workflow in einer anderen Umgebung ausführen kann.

Um die Rechenzeit bzw. die Abarbeitungszeit der Workflows zu minimieren, kann die neue Umgebung ein rechenstarker Server oder Compute-Cluster sein. Die Stand-Alone-Variante wird als eigene Distribution zur Verfügung gestellt. Der Benutzer kann somit diese Version an einem beliebigen Ort entpacken und ausführen. Hierfür beinhaltet die Stand-Alone-Variante alle notwendigen Bibliotheken und Konfigurationen um einen Workflow ohne die Eclipse4Bio-GUI auszuführen. Für eine Steuerung der Ausführung wurde die Stand-Alone-Variante um einen *Command-Line-Interpreter* erweitert. Insbesondere für das Ausführen von exportierten Projekten (Kapitel 5.1.2 auf Seite 47) wird die Stand-Alone-Variante benötigt.

Die Server-Version ist vom Funktionsumfang identisch mit der Stand-Alone-Variante. Sie läuft im Hintergrund, wird automatisch als Dienst gestartet und kann auf einem festgelegten Netzwerk-Port auf Anfragen reagieren und *Workflow-Jobs* entgegennehmen. Diese Workflow-Jobs verwaltet der Server mit Hilfe eines Job-Managements in einer FIFO-Queue. Die Konfiguration des Servers soll mit einer externen Datei möglich sein. Für die Kommunikation zwischen dem Client und dem Server muss ein Protokoll definiert sein. Die Abbildung 7.1 veranschaulicht den Aufbau eines solchen Serversystems. Die Server Variante gehört nicht zum Funktionsumfang von Eclipse4Bio, ist jedoch eine mögliche Erweiterung.

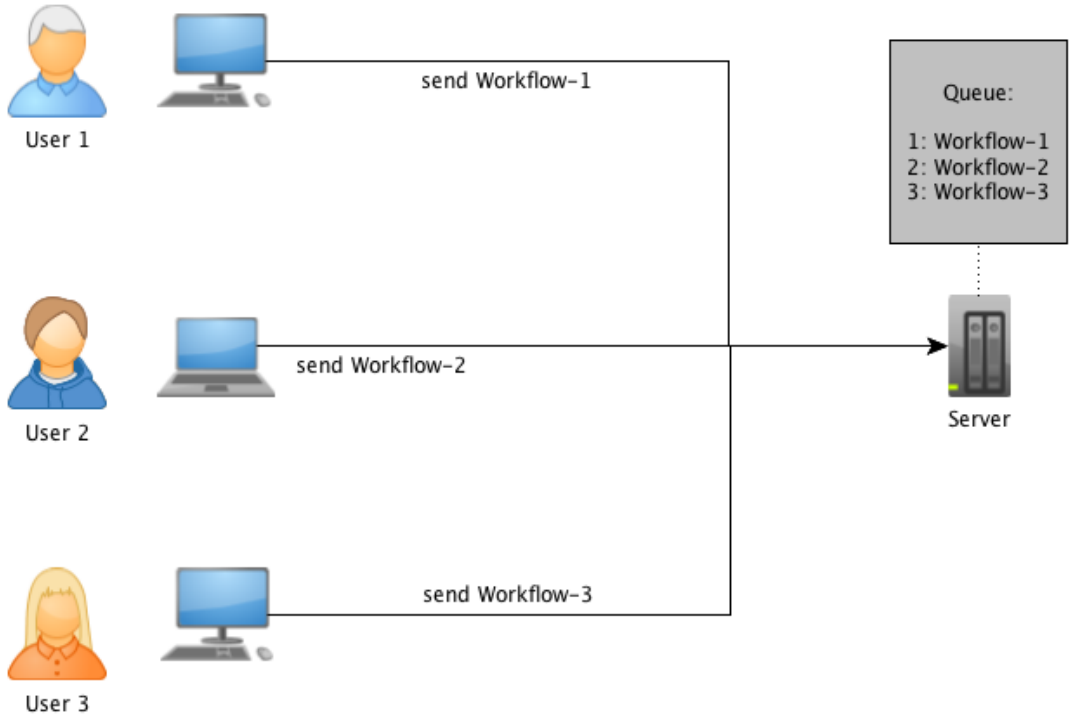

Abbildung 7.1.: Engine Client-Server Architektur

In einer informellen Expertenbefragung mit Bioinformatikern der TU-Dortmund und den Projektgruppenteilnehmern wurden weitere wichtige funktionale Anforderungen definiert. Diese funktionalen Anforderungen gelten für alle zuvor definierten Engine-Varianten und können somit als Basis für jede Engine-Variante betrachtet werden. Die Engine muss ein detailreiches Logging besitzen, das die Abarbeitung der Workflows detailliert protokolliert und diverse wichtige Informationen beinhaltet. Zu den einzelnen Log-Einträgen wird laut Aussage der Experten eine bis auf die Millisekunde genaue Zeitangabe vorausgesetzt und die Log-Einträge müssen aussagekräftig genug sein, um zu beschreiben was gerade passiert und mit welchen zusätzlichen Parametern gearbeitet wird.

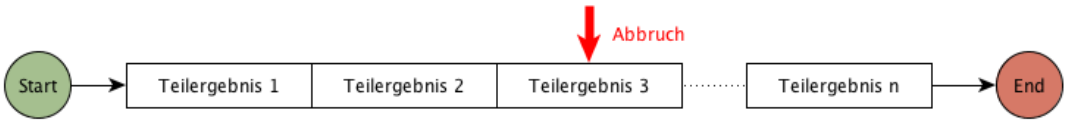

Abbildung 7.2.: Engine ohne die Stop & Resume Funktion. Nach dem Abbruch bei Teilergebnis 3, müssen alle zuvor berechnetet Teilergebnisse neu berechnet werden.

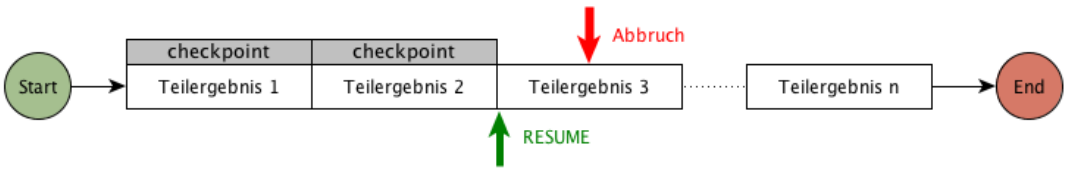

Abbildung 7.3.: Engine mit der Stop & Resume Funktion. Nach dem Abbruch bei Teilergebnis 3, kann die Engine wieder bei der Berechnung von Teilergebnis 3 anfangen (Resume) und auf die zuvor gespeicherten Teilergebnissee 1 und 2 zugreifen. Dadurch entfällt die Neuberechnung von den Teilergebnisse 1 und 2.

Es müssen alle relevanten Ereignisse protokolliert werden, wie zum Beispiel der Start, die Abarbeitung und das Beenden eines Services. Das ausführliche Logging ist für die Fehleranalysen notwendig, um potenzielle Fehlerquellen zu identifizieren.

Ein *Service* kann für verschiedene Betriebssysteme definiert werden (siehe Kapitel 2.6). Für die Engine ist es wichtig, dass die in dem Workflow definierten Services zu der Plattform kompatibel sind, auf der der Workflow ausgeführt werden soll. Sie kann somit nur „homogene Services“ ausführen. Bei heterogenen Services, die in einem Workflow definiert sind, wird die Ausführung mit einer Fehlermeldung beendet.

Die Ausführung von realen Workflows kann unter Umständen mehrere Stunden bis hin zu mehreren Wochen dauern. Während der Berechnung entstehen in der Regel Zwischenergebnisse, die für den nächsten Berechnungsschritt benötigt werden. Diese werden im Normalfall nicht zwischengespeichert, so dass im Falle eines Abbruches oder Absturzes die gesamte Berechnung neugestartet werden muss (Abbildung 7.2). Wünschenswert ist eine sogenannte *Stop & Resume*-Funktion, mit den die Engine in der Lage sein soll, im Falle eines Abbruches wieder an der unterbrochenen Stelle weiterzuarbeiten und nicht die zuvor berechneten Teilergebnisse erneut berechnen zu müssen. Mit Hilfe dieser Funktion soll die Produktivität und die Effizienz gesteigert werden. Abbildung 7.3 zeigt die Ausführung eines sequenziellen Workflows mit einem Abbruch udn Resume.

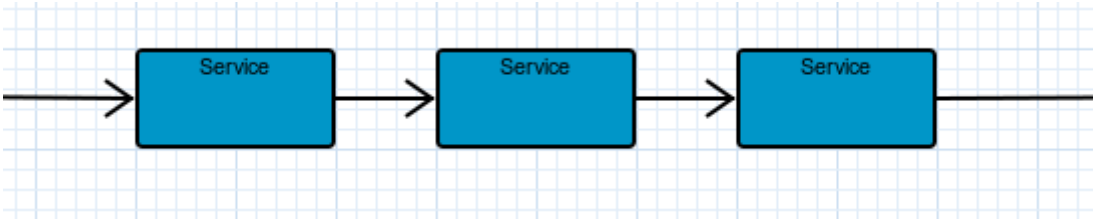

Abbildung 7.4.: Sequence Workflow-Element

Die Engine muss in der Lage sein die *Service*- (Abbildung 7.4), *Condition*- (Abbildung 7.5) und *Split*- (Abbildung 7.6) Workflow-Elemente zu interpretieren und auszuführen. Das *Service*-Element ist das Standardelement und wird sequenziell nacheinander abgearbeitet.

Beim *Condition*-Element muss die Engine die Condition auswerten und dem korrekten Pfad im Workflow folgen. Das *Split*-Element wird für eine parallele Abarbeitung verwendet, indem mehrere Verzweigungen erstellt werden. Jeder dieser einzelnen Pfade wird von der Engine parallel abgearbeitet. Zu jedem Split muss ein *Merge*-Element (Abbildung 7.6) existieren. Die Engine führt den Workflow erst dann weiter aus, sobald alle ausgehenden Kanten von dem Split-Element in dem Merge-Element angekommen sind.

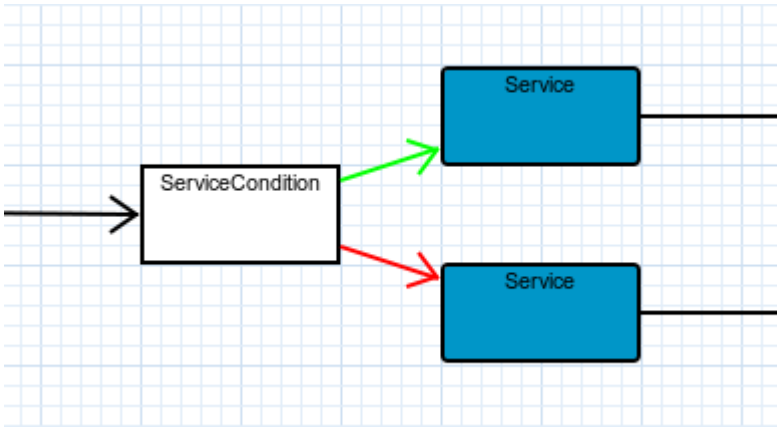

Abbildung 7.5.: Condition Workflow-Element

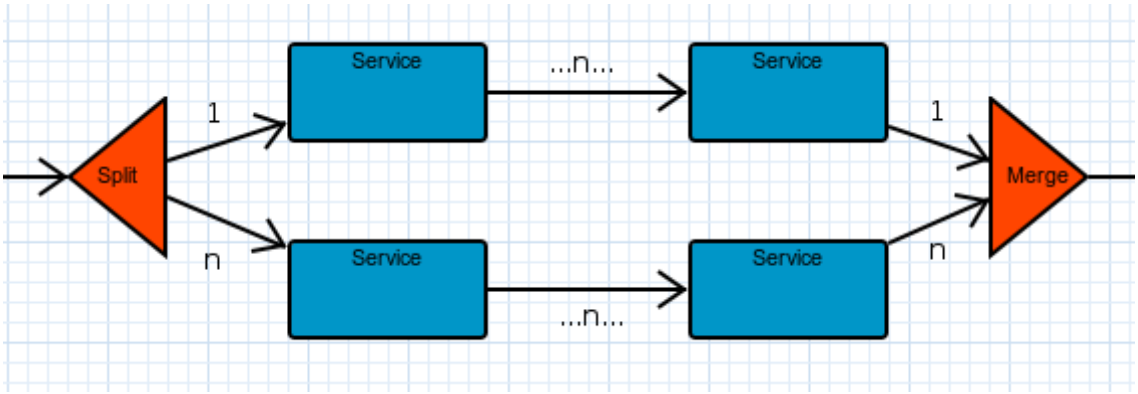

Abbildung 7.6.: Split Workflow-Element

8. SpringBatch

Um den Anforderung der Ausführung eines Workflows gerecht zu werden, wurde SpringBatch eingesetzt. Es ist ein Framework für Stapelverarbeitungen, das wiederverwendbare Funktionen zur Verfügung stellt, die essentiell für die Verarbeitung von großen Datenmengen sind. Darunter fallen zum Beispiel Logging/Tracing, Transaktionsmanagement, Statistiken über die Jobverarbeitung, Jobneustart, Überspringen und Ressourcenmanagement. Zusätzlich bietet es erweiterte technische Möglichkeiten für die Stapelverarbeitung hoher Datenmengen mittels Optimierungs- und Partionierungstechniken [44]. Dies entspricht den Anforderungen der betrachteten Bioinformatikanwendungen.

SpringBatch ist in Java implementiert, eine wünschenswerte Eigenschaft, um die Integration und Kommunikation mit anderen Komponenten von Eclipse4Bio zu erleichtern und Ausführung auf jedem unterstützten Betriebssystem zu gewährleisten.

8.1. Die Job-Definition

Damit der Eclipse4Bio-Workflow von SpringBatch ausgeführt werden kann, muss er in eine für SpringBatch lesbare Sprache übersetzt werden. Dies ist die Job-Definition, ein XML-Dokument, das die gesamte Beschreibung des Workflows enthält. Zur Laufzeit bilden *Java-Tasklets* die Basis der Ausführung. Daher besteht der Inhalt der Job-Definition im Kern aus Definitionen für die *Tasklets*, sowie weiteren Steuerungs-Tags namens *Step*, *Decision* und *Split*.

Step ist das Standardelement einer SpringBatch-Jobkonfiguration. Er beinhaltet alle Informationen, die notwendig sind, um eine Stapelverarbeitung zu definieren und zu kontrollieren.

Eclipse4Bio benutzt drei Eigenschaften der Steps: *id* und *next*, die zur Steuerung des Flusses dienen und *parent* für den Aufruf eines übergeordneten Steps als Monitoring-Funktion. Jeder Step ist nach dem Element benannt, welches es in der Workflowdefinition repräsentiert. Um eine Verwechslung mit den Taskletnamen zu vermeiden, werden die Namen mit dem Zusatz „Step“ versehen. Durch diese Id kann der Step von einem anderen Element aufgerufen und damit ausgeführt werden. Der Wert der Eigenschaft *next* ist die Id des nächsten Elementes, das ausgeführt werden soll, sobald der Step beendet ist. Es existiert die Möglichkeit, eine Art Vererbung in den Steps über das Attribut *parent*

zu definieren. Dieser wird zusätzlich zum eigentlichen Step ausgeführt und benutzt, um einen Listener auf den Step zu setzen. Wie in Abbildung 8.1 zu sehen ist, führt der Step zur StepExecution. In ihm wird der Ausführungsstatus verwaltet, der über einen Listener beobachtet werden kann. Ändert sich der Status, so wird diese Information über eine Schnittstelle an die GUI weitergeleitet.

Der Kern des Steps liegt im Inneren, im Tasklet-Tag. Hier wird das ebenfalls in der Job-Definition enthaltene Tasklet angesprochen, in dem die eigentliche Aufgabe definiert ist. Im Beispiel-Listing 8.1 ist ein Step namens „aStep“ mit einem nachfolgenden Element namens „bStep“ zu erkennen; zusätzlich wird „abstractStep“ ausgeführt. Seine Aufgabe ist im Tasklet „initer“ definiert.

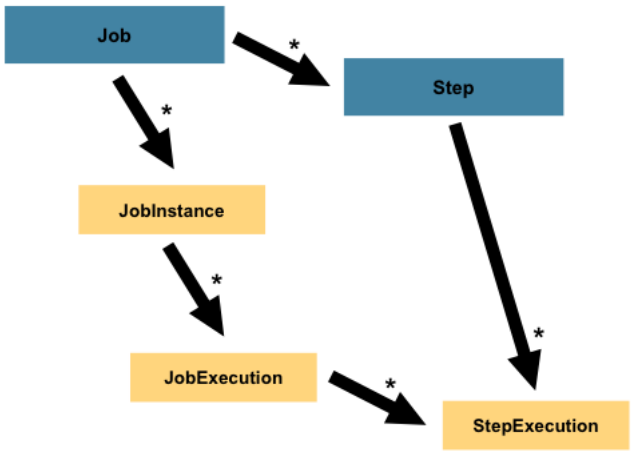

Abbildung 8.1.: Die Ausführung eines Steps

Tasklets definieren die eigentliche Aufgabe, die in einem Step ausgeführt werden soll. In Eclipse4Bio sind dies in der Regel Bash-Aufrufe, die aus einer speziell dafür geschriebenen Java-Klasse namens *BashExecuterTasklet* aufgerufen werden. Jedes Tasklet erhält, genau wie jedes andere Element, eine eindeutige Identifikation *id* über die es angesprochen werden kann. Zusätzlich enthält es das Attribut *class*, in unserem Fall den *BashExecuterTasklet*. Es handelt sich um eine JavaBean, deren *Properties* durch XML-Definition gefüllt werden.

Folgende Eigenschaften werden gesetzt:

command wird als Kommando ausgeführt, wenn das Betriebssystem nicht erkannt wird.

timeout legt einen Timeout für das Kommando fest.

interruptOnCancel legt fest, ob der Thread, mit dem das Kommando verknüpft ist, gelöscht wird, wenn der dazugehörige Job abgebrochen wird.

```
<batch:step id="aStep" next="bStep" parent="'abstractStep">
    <batch:tasklet ref="initer" transaction-manager="'transactionManager"/>
</batch:step>
```

Listing 8.1: XML-Beispielcode für einen Step

```
<batch:decision id="conditionOneStep" decider="myCondition">
  <batch:next on="COMPLETED" to="aStep" />
  <batch:next on="FAILED" to="bStep" />
</batch:decision>

<bean id="myCondition" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
    decider.FileExistDecider">
  <property name="file" value="thirdFileName" />
</bean>
```

Listing 8.2: XML-Beispielcode für eine Decision

terminationCheckInterval stellt den Parameter ein, wie oft das System überprüfen soll, ob das Kommando abgeschlossen ist. Dies wird bei der parallelen Ausführung benötigt.

windowsCommand wird als Kommando ausgeführt, wenn der Job mit einem Windows-System läuft.

linuxCommand wird als Kommando ausgeführt, wenn der Job mit einem Linux-System läuft.

macCommand wird als Kommando ausgeführt, wenn der Job mit einem Mac-OSX-System läuft.

Durch das Setzen der Eigenschaften – besonders die der Kommandos – wird das Verhalten der Klasse festgelegt, zum Beispiel welcher Code mit welchem System ausgeführt werden soll. Die Bestimmung des aktuellen Systems, auf dem der Job ausgeführt wird, erfolgt zur Laufzeit.

Listing 8.3 zeigt ein Beispielcode für ein Tasklet mit den drei Kommandos für die verschiedenen Betriebssysteme. Die Kommandos sind jeweils optional, so dass auch nur ein Befehl für ein bestimmtes System oder prinzipiell sogar kein Befehl möglich wäre.

Zusätzlich zum BashExecuterTasklet hat Eclipse4Bio aktuell noch drei weitere Tasklets implementiert. Diese erfüllen zur Zeit keine Funktion, können aber in Zukunft für Aufgaben genutzt werden.

EnderTasklet wird von den Endelementen in einem Workflow ausgeführt.

IniterTasklet wird vom Startelement ausgeführt.

```
12 <bean id="aService" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.tasklets.
       BashExecuterTasklet">
     <property name="command" value="" />
14   <property name="timeout" value="9223372036854775807" />
     <property name="interruptOnCancel" value="true" />
16   <property name="terminationCheckInterval" value="1000" />

18   <property name="windowsCommand" value="notepad d:\s1.txt" />
     <property name="linuxCommand" value="sleep 6" />
20   <property name="macCommand" value="java -jar simpleTester.jar test.txt
         test1.txt" />
   </bean>
```

Listing 8.3: XML-Beispielcode für ein Tasklet

Decisions bieten die Möglichkeit einen nichtlinearen Fluss zu modellieren und dienen dazu Entscheidungen zu treffen. In SpringBatch ist es möglich, von einer Klasse namens JobExecutionDecider zu erben, die bei einem Aufruf einen von vier verschiedenen Rückgabewerten liefert. Diese sind grundsätzlich frei interpretierbar. In Eclipse4Bio wurde folgende Interpretation gewählt, da sie intuitiv mit der Semantik der Workflows einhergeht:

- COMPLETED - Die Entscheidung im Decider fiel positiv aus.
- FAILED - Die Entscheidung im Decider fiel negativ aus.
- STOPPED - Der Flow wurde gestoppt, als er sich gerade im Entscheidungsprozess befand.
- UNKNOWN - Das Ergebnis ist nicht bekannt, da beispielsweise ein Fehler bei der Bearbeitung auftrat, der eine Entscheidung nicht möglich macht.

Aktuell werden jedoch nur die beiden Status *COMPLETE* und *FAILED* von Eclipse4Bio benutzt, da weder eine Pausierung der Ausführung noch eine detaillierte Fehlerbehandlung implementiert ist.

Die *id* eines Deciders besteht aus dem Namen der *Condition* der Workflowdefinition mit dem Zusatz „Step“; analog zum *step*, der dazu dient, die Identifikation der eigentlichen Flusselemente von denen der Metaelemente, zum Beispiel der Tasklets, abzugrenzen. Da kein fester Nachfolger existiert, fehlt das Attribut *next*. Es wird durch den Tag *batch:next* ersetzt, der beschreibt, zu welchem Folgeelement der Fluss bei einem der vier Status geleitet wird. Zusätzlich braucht jeder Decisionstep die Angabe der erstellten Java-Klasse, die die Entscheidung trifft. Dies geschieht im Attribut *decider*. Listing 8.2 zeigt ein Beispiel für eine Decision mit dem Namen „conditionOneStep“, die – sobald sie im Fluss aufgerufen wird – ihre angegebene Klasse *FileExistDecider* aufruft, deren Rückgabewert auswertet

und bei *COMPLETED* zu aStep und bei *FAILED* zu bStep weiterleitet. Der Decider wird völlig analog zum Tasklet außerhalb des Jobs definiert.

Zur Zeit sind 5 Standard-Decider implementiert:

1. FileCanReadDecider - prüft auf Leserechte für eine Datei.
2. FileCanWriteDecider - prüft auf Schreibrechte für eine Datei.
3. FileExistDecider - prüft, ob eine Datei vorhanden ist.
4. FileLongValueDecider - öffnet eine Datei und vergleicht ihren Inhalt mit einem Wert.
5. FileSizeDecider - vergleicht die Größe einer Datei mit einem Wert.

Split geben dem Benutzer die Möglichkeit Parallelität zu modellieren. Gerade bei großen Datenmengen kann es vorkommen, dass Daten parallel abgearbeitet werden sollen. Zu diesem Zweck muss eine parallele Ausführung in der Job-Definition festgelegt werden können, wofür SpringBatch das Element *split* zur Verfügung stellt. Es ist ein abgeschlossener Block innerhalb der Festlegung der Ausführung mit *id* und *next* für die Festlegung der eigenen Identifikation und des Nachfolgers. Ein *split* kann folglich angesprochen und behandelt werden wie jedes andere Element zur Flusskontrolle auch; der Unterschied besteht lediglich aus den inneren Abläufen, dem *flow*. Ein *split* kann beliebig viele *flows* besitzen, wobei jedes davon einen parallelen Fluss erstellt. Die Flussdefinition innerhalb eines *flow* ist analog zu der des globalen. Es können die gleichen Elemente auf identische Weise verwendet werden. Jeder innere Fluss muss beendet werden, bevor der Hauptfluss nach dem *split* fortgesetzt wird. Listing 8.4 zeigt ein Beispielcode für ein *split* bestehend aus zwei *flows*.

8.2. Der CommandLineRunner

Eines der Ziele ist es, per grafischem Editor einen Workflow zu erstellen und diesen in einem Schritt in eine Job-Definition umwandeln und starten zu können. Dies wird durch den CommandLineRunner ermöglicht, der bereits im SpringBatch-Framework vorhanden ist. Durch ihn ist es möglich, auf einfachem Wege mittels Kommandozeile die SpringBatch-Ausführung zu starten. Er ist ebenfalls über die Konsole bedienbar, eine Eigenschaft, welche für ein eventuelles Ansprechen ohne grafische Benutzeroberfläche nötig ist. Leider erwartet er, dass das XML-Dokument bereits im Classpath und damit vor dem Start vorhanden ist. Dies stellte in der Anfangszeit bei der Arbeit mit SpringBatch das Hauptproblem dar, da das XML-Dokument erst zur Laufzeit aus der Workflowdefinition generiert wird. Experimente mit anderen, komplexeren angebotenen Runnern führten zu keinem benutzbaren Ergebnis. Die Lösung wurde schließlich im offenen Quelltext des

```
22 <batch:split id="Split1Step" task-executor="taskExecutor" next="Split1After"
     >
   <batch:flow>
24   <batch:step id="Service2Step">
       <batch:tasklet ref="Service2"/>
26   </batch:step>
   </batch:flow>
28
   <batch:flow>
30   <batch:step id="Service3Step" next="Service4Step">
       <batch:tasklet ref="Service3"/>
32   </batch:step>
     <batch:step id="Service4Step">
34     <batch:tasklet ref="Service4"/>
     </batch:step>
36 </batch:flow>
</batch:split>
```

Listing 8.4: XML-Beispielcode für einen Split

CommandLineRunners ersichtlich. Er verwendet in seiner Hauptmethode eine Klasse namens *ClassPathXmlApplicationContext*, die lediglich Dokumente im Classpath verarbeiten kann. Da SpringBatch nicht darauf ausgelegt ist in ein größeres Projekt implementiert zu werden, existiert keine Konfigurationsmöglichkeit, um den Classpath zu umgeben. Durch das Duplizieren des gesamten Codes in eine eigene Runner-Klasse und dem Austauschen der *ClassPathXmlApplicationContext* durch die ebenfalls vorhandene *FileSystemXmlApplicationContext*, ist es möglich zur Laufzeit generierte Job-Definitionen ausführen zu lassen.

9. Die Ausführung eines Workflows

Um einen Workflow ausführen zu können, muss er zunächst in eine passende SpringBatch Definition übersetzt werden. Diese Transformation und die verwendete Templatesprache werden in diesem Kapitel erläutert. Weiterhin wird ein Überblick über die von der Engine angebotenen Funktionalitäten gegeben. Somit wird in diesem Kapitel ersichtlich, dass die Anforderungen, die an die Engine gestellt worden sind, alle erfüllt werden konnten.

9.1. Transformation

Die Umwandlung aus Service- und Workflowdefinitionen in eine ausführbaren Job-Definition bildet den ersten Schritt der Ausführung. Mit Hilfe einer rekursiven DFS-Traversierung erstellt Eclipse4Bio die XML-Job-Definition sequenziell. Die Generierung kann sequenziell geschehen, mit Ausnahme der Split Knoten. Hier ist in der Job-Definition eine Verschachtelung der Element notwendig. Dadurch ist es notwendig die Traversierung eines Merge-Knoten als Rekursionsende anzusehen und nachdem alle zuvor gesplitteten Flüsse den Merge-Knoten erreicht haben, hinter ihm fortgesetzt. Dies ist, bei einem sequenziellen Aufbau, zwingend erforderlich, da die Verschachtelung des vorangegangenen Splits erst geschlossen werden muss, bevor nachfolgende Elemente ausgegeben werden können.

Der Nachfolger eines Splits muss bereits zu Beginn, beim Erzeugen des Split-Knotens, bekannt sein, da hier die Eigenschaft *next* festgelegt wird. Dies ist bei einer Traversierung durch den Graphen vom Start bis zum Ende nicht möglich. Um dieses Problem zu lösen ersetzt die Umwandlung den Namen des Merge-Steps, der nach jedem Split zwangsläufig erfolgt, durch den Namen des Split-Step und dem Zusatz „After“, um ihn eindeutig zu machen.

Während der Erstellung werden alle In- und Outputvariablen in allen Elementen durch ihren entsprechend angegebenen Wert ersetzt. Abbildung 9.1 zeigt einen Beispielgraphen, der aus drei Service-, einem Split-, einem Merge- und den beiden obligatorischen Start- und Endknoten besteht. Zugehörig zu diesem Graphen ist die vollständige SpringBatch-XML-Job-Definition in Listing A.1 im Anhang.

Die Servicedefinitionen sind hier der Übersicht halber nicht ausführlich angegeben. Jeder der drei Services führt unter Windows Notepad aus und öffnet eine Datei namens s1.txt, s2.txt oder s3.txt auf Festplatte „d“.

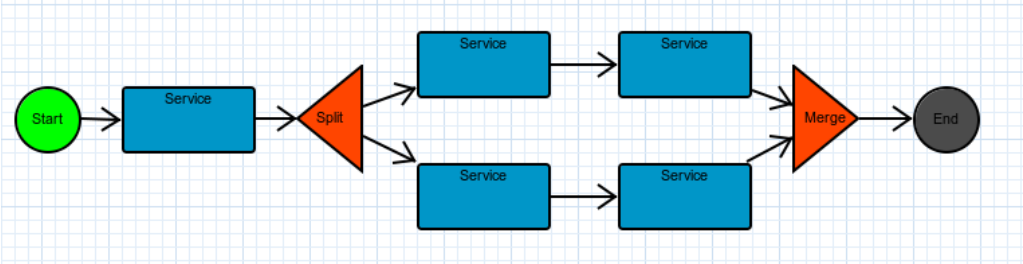

Abbildung 9.1.: Beispiel: Workflow für ein Split

9.2. Templatesprachen

Für die Aufgabe der Erstellung der Job-Definition, ist grundsätzlich jede Sprache geeignet, die fähig ist Text in eine Datei zu schreiben, insbesondere also auch Java. Durch die Java-Syntax ist eine Erzeugung mittels Java jedoch nicht wünschenswert. Der Code wäre schwer lesbar und damit auch schwer zu debuggen. Hierzu bieten sich in Java benutzbare Templatesprache an, die genau zu diesem Zweck entwickelt wurden. Dabei kann in der Regel Code in der entsprechenden Templatespache erstellt werden, der anschließend in eine importierbare Java-Klasse umgewandelt wird. Diese ist danach durch die zusätzliche Syntax eines erstellten Codes noch schwieriger lesbar. Da Änderungen direkt im Template gemacht werden können ist eine direkte Bearbeitung aber nicht nötig. Für Java und Eclipse sind viele Templatesprachen verfügbar.

xPand2 war zunächst die für Eclipse4Bio favorisierte Templatesprache. Die Installation von xPand2 in Eclipse „EMF 3.6 Helios x86“ war jedoch nicht möglich, weswegen die Sprache vorerst nicht weiter untersucht wurde. Zeitgleich erschien Eclipse Version 3.7 Indigo, in der das Plugin bereits integriert ist. Der vorhandene Editor war jedoch unbenutzbar; beim Erstellen einer Schleife gab Eclipse eine Nullpointer-Exception in einer MessageBox aus. Obwohl es prinzipiell funktionierte, war ein flüssiges Arbeiten nicht möglich. Zusätzlich hätten alle Entwickler die Version auf Eclipse „3.7 Indigo“ umstellen müssen, was mit großer Wahrscheinlichkeit weitere Fehler zur Folge gehabt hätte.

StringTemplate ist eine Java-Template-Engine, die sehr einfach in ein bestehendes Produkt integriert wird. Sie benötigt kaum Konfigurationsaufwand oder weitere Abhängigkeiten und wird direkt im Java-Code ausgeführt. Die Engine ist vor allem darauf ausgelegt Quellcode zu generieren und wird auch in anderen Bibliotheken z.B. in SpringBatch eingesetzt. [45] Mit StringTemplate ist es möglich die Templates für die einzelnen Komponenten aufzuteilen und so einen modularen Aufbau zu erreichen. Verschiedene Konstrukte der Workflow Sprache (Schleifen, Bedingungen, etc.) können mit der Engine ohne Probleme ausgewertet werden. Zwar erfüllt StringTemplate so unsere technischen Anforderungen, jedoch gab es zum Zeitpunkt der Evaluation

```
    // Generate workflow model
    Workflow2SBXML w2sb = new Workflow2SBXML();
    JET2Context context = new JET2Context(null);
    context.setVariable("workflow", workflow);

    // Write SpringBatch XML with JET
    MyJetWriter jetWriter = new MyJetWriter(filename);
    w2sb.generate(context, jetWriter);
    jetWriter.close();
```

Listing 9.1: Auschnitt aus der Engine Klasse (Engine.java)

für unsere Entwicklungsumgebung (Eclipse 3.6) keine Unterstützung bei der Erstellung der Templates. Diese müssen als einfache Textdokumente erstellt werden. Durch das Fehlen der automatischen Codevervollständigung und einer automatischen Syntaxanalyse ist die Erstellung der Templates zu zeitaufwändig. Änderungen von Teilen des Modells während der Entwicklung der Workflowsprache führen zu Fehlern, die Aufgrund der fehlenden IDE-Unterstützung erst zur Laufzeit bemerkt werden.

JET (Java Emitter Templates) ist ein Teilprojekt des „Model To Text"(M2T)-Projektes der Eclipse Foundation. Im Kontext dieses Projektes ist JET eine Komponente der modellgetriebenenen Entwicklung. JET wird dabei typischerweise als Code-Generator eingesetzt. Die Transformation von abstrakten Modellen (z.B. EMF/ECORE Modelle) in Code kann mit Hilfe von JET definiert und ausgeführt werden [8]. Ein JET-Converter wird als Eclipse Plugin angelegt. Der Editor, der für Eclipse verfügbaren JET-Erweiterung unterstützt sowohl eine automatische Codevervollständigung, als auch Syntaxanalyse. Damit ist es möglich schnell die Templates für die verschiedenen Fragmente der Workflowsprache zu erstellen oder nachträglich zu editieren. Der enthaltene JET-Compiler generiert bei jeder Änderung der Templates automatisch die entsprechenden Java-Klassen, die die eigentliche Umwandlung durchführen. Durch die Möglichkeit auch nativen Java Code in den Templates verwenden zu können, existieren keine Einschränkungen bezüglich der Mächtigkeit der Umwandlungssprache.

Die durch Jet generierten Klassen werden dazu verwendet, aus einer Workflow-Definition die Job-Definition in XML für SpringBatch zu erzeugen. Listing 9.1 zeigt den Hauptaufruf zur Umwandlung. Zunächst wird eine Instanz der generierten Klasse *Workflow2SBXML* erzeugt und in der Klasse *JET2Context* wird das Objekt-Modell des Workflows abgelegt. Mit der Implementierung *MyJetWriter* des Interfaces JET2Writer, die im Wesentlichen auf einem einfachen FileWriter basiert, wird die Umwandlung gestartet und abschließend der Writer geschlossen.

```
<%@ jet package="de.tu_dortmund.cs.pg.eclipse4bio.engine.jetconverter"
  imports="WorkflowDefinition.* de.tu_dortmund.cs.pg.eclipse4bio.engine.
      jetconverter.tools.*"
  class="ConditionOut" %>
<% Condition condition = (Condition)context.getVariable("condition");%>
    <batch:decision id="<%=condition.getName()%>Step" decider="<%=condition.
        getName()%>">
      <batch:next on="COMPLETED" to="<%=WorkflowConverterTools.
          getControlFlowName(condition.getTrue())%>Step" />
      <batch:next on="FAILED" to="<%=WorkflowConverterTools.
          getControlFlowName(condition.getFalse())%>Step" />
    </batch:decision>
```

Listing 9.2: Beispiel für einen JET-Template-File (ConditionOut.jet)

In Listing 9.2 ist der Aufbau des Kopfes einer JET Datei exemplarisch aufgezeigt. *Package* und *class* definieren das Package und den Namen der generierten Klassen. Über das Attribut *import* werden, die im Template benötigten Klassen importiert. Mittels *<% ... %>* lässt sich nativer Java-Code in das Template einbinden.

9.3. Features

Die Ausführung bietet verschiedene Optionen, die vor der Laufzeit gesetzt werden können oder während der Laufzeit die Ausführung beeinflussen.

Monitoring dient der Überwachung der Ausführung. Die Oberfläche greift dabei auf ein Interface zu, um die Ausführung grafisch darzustellen. Hierbei wird eine Statusänderung der SpringBatch Ausführung von der Engine registriert und an das Interface weitergeleitet.

Resume gibt dem User die Möglichkeit, eine abgebrochenen oder auf Grund eines Fehlers fehlgeschlagene Ausführung wieder Aufzunehmen. Hierzu schreibt Eclipse4Bio während der Ausführung eine Log-Datei. Diese dient dazu persistent den Status der abgeschlossenen Steps festzuhalten, mit deren Hilfe der aktuelle Zustand beim Abbruch wiederhergestellt wird. Dies geschieht durch überspringen bereits vor dem Abbruch vollendeter Ausführungen, bzw. bei Conditions die Weiterleitung des Flusses entsprechend der beim vorherigen Mal getroffenen Entscheidung. Dementsprechend speichert diese Log-Datei bei Conditions ebenfalls wie sie sich entschieden haben.

Stop ist eine Funktion für die Grafische Oberfläche, die es ermöglicht die Ausführung zu beenden, nachdem der aktuell laufende Step abgeschlossen ist. Dies ist nützlich, wenn der User zu einem späteren Zeitpunkt die Ausführung wieder aufnehmen möchte und nur temporär Ressourcen freigeben möchte.

9.4. Ausführung von Webservices

In den letzten Kapitel wurde beschrieben, welche Anforderungen an die Engine gestellt worden sind. Um dieses Anforderungen gerecht zu werden, wurde das Framework Spring-Batch verwendet, das hinreichend beschrieben worden ist. Anschließend wurde erläutert, wie ein Workflow mit Hilfe der so entwickelten Engine ausgeführt werden kann. Dabei haben sich die Anforderungen im Laufe der Zeit stark verändert. Nach mehreren Gesprächen mit den Bioinformatikern der TU Dortmund, wurden die Anforderungen so modifiziert, dass die geäußerten Wünsche realisiert werden konnten. Die Anforderungen sind dabei von den Webservices hin zu den lokalen Diensten gewechselt. Ursprünglich bestand die Aufgabenstellung darin, die Workflows und deren Ausführung auf Webservices zu spezialisieren. Zu diesem Zweck bestand eine Primärfunktion, einen Wizard bereitzustellen, der einfache Kommunikation mit BioCatalogue ermöglicht.

BioCatalogue ist ein Open-Source-Verzeichnis von biologischen Webservices. Die BioCatalogue-Webseite ist frei zugänglich für die Registrierung, Kommentierung und Kontrolle von Webservices des Faches Biologie. Er umfasst insgesamt 2065 Services von 146 Providern und bietet eine REST-API an, mit der man sämtliche Informationen des Systems abrufen kann. Durch die vergebenen Rechte kann man das System verändern und als registrierter Benutzer auch neue Services hinzufügen. Die Ziele der Platform sind laut eigenen Angaben (BioCatalogue, Biocatalogue Wiki, 2011):

- die Zurverfügungstellung eines zentralen Registrierungspunktes für Webservices und einer einzigen Suchseite für Wissenschaftler und Entwickler.
- einen verwalteten Katalog von biowissenschaftlichen Webservices anzubieten.
- der Katalog wird von Anbieter, Experten und Benutzer beaufsichtigt und überwacht und liefert hochqualitative Beschreibungen für die Services.
- eine Ermöglichung des BioCatalogues, die Community der Experten und Verwalter bestimmten Services zu finden und zu kontaktieren.

Ein wichtiger Punkt war die zentrale Registrierung. Mit BioCatalogue ist es möglich, praktisch alle relevanten Services für BioInformatik an einem Ort zu finden. Zusätzlich sind die Services in Kategorien eingeteilt und es werden verschiedene Suchfunktionen angeboten, die es dem Anwender ermöglichen, schnell den passenden Service zu finden. Der Großteil der Kommunikation mit den angebotenen Services basiert auf SOAP, dessen jeweilige Interfaces bzw. Endpoints bei BioCatalogue in einem WSDL-Dokument hinterlegt sind und abgerufen werden können. In diesem Dokument sind alle technischen Informationen enthalten, die eine automatische Erstellung eines speziellen Workflow-Service-Elements für diesen Service generieren zu lassen. Der Benutzer von Eclipse4Bio hätte neben der graphischen Modellierung seines Workflows die Option gehabt, durch einen integrierten Navigator einen

Webservice, der auf BioCatalogue angeboten wird, zu suchen und auszuwählen. Anschließend würde die WSDL-Definition geladen und ein benutzbares Workflow-Element generiert. Diese Workflow-Elemente hätten Ein- und Ausgänge und könnten auch wie andere Elemente des Workflows verwendet werden. Die Erstellung der Verbindung ist dabei nicht direkt auf einfachem Wege in Java realisierbar, daher bieten sich mehrere Frameworks, unter anderem ApacheODE.

ApacheODE [4] (Orchestration Director Engine) ist für die Ausführung von Webservices konzipiert. Es unterstützt WS-BPEL 2.0 [59] und ist somit auf dem neusten Stand der Spezifikation.

Dank der „Integration-API“ ist es möglich, das ApacheODE-Framework in eine Java-Applikationen einzubetten. Ein weiterer Grund für die Verwendung für Apache ODE ist, dass viele andere Open-Source-Projekte ebenfalls Apache ODE als Basis-Framework benutzen. Ein prominentes Beispiel für den Apache ODE Einsatz, ist das **RiftSaw** [27] Projekt von der Firma JBoss. Das *RiftSaw*-Projekt von JBoss, ist eine WS-BPEL-2.0-konforme Ausführungsengine. Apache ODE ist jedoch im Gegensatz zu RiftSaw und anderen Derivaten leichtgewichtig und kann besser den Anforderungen an das Eclipse4Bio-Projekt angepasst werden, was es für einen Einsatz qualifiziert. Durch die Änderungen der Zielsetzung müssen keine Webservices mehr angesprochen werden, so dass diese Funktionalität nicht mehr im Kern verfügbar sein muss. Dadurch wird kein Interface mehr für die Kommunkation mit BioCatalogue benötigt und als Resultat auch keine Engine, die diese Aufgabe übernimmt. Ein Anwender kann trotzdem Webservices ansprechen, indem er ein eigenes Programm schreibt, das diese Aufgabe erledigt und er dieses wiederum per Bash-Befehl aus dem Workflow heraus startet.

10. Zusammenfassung

In diesem Bericht wurde beschrieben, wie das Open-Source-Workflowmanagementsystem Eclipse4Bio entwickelt wurde, mit dem man Workflows, die speziell für die Anwendung in der Bioinformatik ausgelegt sind, erstellen kann.

Dazu wurde als Motivation in die Problemstellung der Bioinformatik eingeführt. Dabei zeigte sich, dass insbesondere aufgrund der modernen Hochdurchsatztechnologien Analyse-Workflows in Zukunft von großer Bedeutung sein werden. Es wurde ein Framework entwickelt, mit denen Workflows textuell und grafisch dargestellt können. Ein Workflow kann mit einem grafischen oder einem textuellen Editor vom Benutzer in Eclipse erstellt und ausgeführt werden. An die Ausführung eines Workflows sind besondere Anforderungen, die aus dem Bereich der Bioinformatik kommen, gestellt worden. Diese Anforderungen konnten umgesetzt werden.

Folgende Erweiterungen für Eclipse4Bio sind denkbar. Aufgrund der großen Datenmengen kann die Ausführung eines Workflows auf einem Cluster hilfreich sein. Zu diesem Zweck müssten zunächst passende Konzepte erarbeitet werden, da das Ausführen eines Workflows auf Clustern nicht trivial ist. Weiterhin ist der Aufbau einer großen Menge von Services denkbar. Diese Menge an Services kann das Erstellen eines individuellen Workflows sehr vereinfachen. Dabei kann zum Beispiel auf Services anderer Forschungsgruppen zurückgegriffen werden, um sich so Arbeit zu ersparen. Durch steigende Wiederverwendbarkeit, fielen auch die Kosten der Entwicklung.

Zusammenfassend wurde mit Eclipse4Bio ein Workflowmanagementsystem entwickelt, das der besonderen Herausforderung im Bereich der Bioinformatik gerecht wird. Dazu wird ein Workflow textuell und grafisch repräsentiert. Diese Flexibilität kann die Arbeit mit Eclipse4Bio sehr vereinfachen.

A. Anhang

Listing A.1: Beispiel: vollständiges Spring Batch XML Dokument nach Umwandlung

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:batch="http://www.springframework.org/schema/batch"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.1.xsd
      http://www.springframework.org/schema/batch
   http://www.springframework.org/schema/batch/spring-batch-2.1.xsd
      http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-2.1.xsd
      http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-2.1.xsd">

  <import resource="../config/jobrepository-memory.xml"/>

  <bean id="initer" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.tasklets.
      IniterTasklet"/>
  <bean id="ender" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.tasklets.
      EnderTasklet"/>
  <bean id="nothing" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.tasklets
      .NothingTasklet"/>
  <bean id="taskExecutor" class="org.springframework.core.task.
      SimpleAsyncTaskExecutor" />
  <bean id="stepListener" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
      listener.StepListener"/>
  <bean id="jobListener" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
      listener.JobListener"/>

  <batch:step id="abstractStep" abstract="true">
    <batch:listeners>
      <batch:listener ref="stepListener" />
    </batch:listeners>
  </batch:step>
```

```
70  <bean id="Service1" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
        tasklets.BashExecuterTasklet">
      <property name="command" value="echo 'no command'" />
72    <property name="timeout" value="9223372036854775807" />
      <property name="interruptOnCancel" value="true" />
74    <property name="terminationCheckInterval" value="1000" />
      <property name="windowsCommand" value="notepad d:\s1.txt" />
76    <property name="linuxCommand" value="sleep 6" />
      <property name="macCommand" value="java -jar simpleTester.jar test.txt
          test1.txt" />
78  </bean>

80  <bean id="Service2" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
        tasklets.BashExecuterTasklet">
      <property name="command" value="echo 'no command'" />
82    <property name="timeout" value="9223372036854775807" />
      <property name="interruptOnCancel" value="true" />
84    <property name="terminationCheckInterval" value="1000" />
      <property name="linuxCommand" value="sleep 4" />
86    <property name="windowsCommand" value="notepad d:\s2.txt" />
      <property name="macCommand" value="java -jar simpleTester.jar test1.txt
          test2.txt" />
88  </bean>

90  <bean id="Service3" class="de.tu_dortmund.cs.pg.eclipse4bio.engine.
        tasklets.BashExecuterTasklet">
      <property name="command" value="echo 'no command'" />
92    <property name="timeout" value="9223372036854775807" />
      <property name="interruptOnCancel" value="true" />
94    <property name="terminationCheckInterval" value="1000" />
      <property name="windowsCommand" value="notepad d:\s3.txt" />
96    <property name="linuxCommand" value="test" />
    </bean>
98
    <batch:job id="split-workflow" job-repository="jobRepository">
100   <batch:step id="Service1Step" next="Split1Step">
        <batch:tasklet ref="Service1"/>
102   </batch:step>
      <batch:split id="Split1Step" task-executor="taskExecutor" next="
          Split1After">
104     <batch:flow>
          <batch:step id="Service2Step">
106         <batch:tasklet ref="Service2"/>
          </batch:step>
108     </batch:flow>

110   <batch:flow>
        <batch:step id="Service3Step">
```

```
        <batch:tasklet ref="Service3"/>
      </batch:step>
    </batch:flow>
      </batch:split>
        <batch:step id="Split1After" next="EndeStep">
          <batch:tasklet ref="nothing"/>
        </batch:step>
        <batch:step id="EndeStep">
          <batch:tasklet ref="ender" transaction-manager="transactionManager
              " />
        </batch:step>
  </batch:job>
</beans>
```

```
@gmf(foo="bar")
@namespace(uri="http://www.cs.tu-dortmund.de/pg/eclipse4bio/
    workflowDefinition", prefix="WorkflowDefinition")
package WorkflowDefinition;

@gmf.diagram(foo="bar")
class WorkflowDefinitionType {
  id attr String[1] name;
  val StartElement startElement;
  val EndElement[*] endElements;
  val Condition[*] conditions;
  val Service[*] services;
  val Data[*] datas;
  val ControlFlowLink[*] controlFlowLinks;
  val NormalDataFlowLink[*] normalDataFlowLink;
  val Split[*] split;
  val Merge[*] merge;
}

class ControlFlowLinkStart {
  ref ControlFlowLink[+]#controlFlowBiEnd controlFlows;
}

@gmf.link(source="controlFlowBiEnd", target="controlFlowLinkEnd", target.
    decoration="arrow", style="solid", label="value")
class ControlFlowLink {

  id attr String[1] name;
  attr String value;
  ref EndElement endElement;
  ref Service service;
  ref Condition condition;
  ref ControlFlowLinkStart[1]#controlFlows controlFlowBiEnd;
  ref ControlFlowLinkEnd[1]#controlFlowEndBiEnd controlFlowLinkEnd;
```

```
}

class ControlFlowLinkEnd {
  ref ControlFlowLink[+]#controlFlowLinkEnd controlFlowEndBiEnd;
}

@gmf.node(label="name", figure="ellipse", size="50,50", color="0,255,127")
class StartElement extends ControlFlowLinkStart {

  @GenModel(documentation="The name of the element. It doesn't contain any
      semantic informations, it's just a uniq name.")
  id attr String[1] name;
}
```

Listing A.2: Erweitertes Ecore-Modell durch EuGENia-Annotationen repräsentiert in Emfatic-Notation. Das *StartElement* erbt von der Klasse „ControlFlowLink“, die durch eine Kante im Diagramm dargestellt werden soll. Das wird durch die Annotation „@gmf.link(source=“controlFlowBiEnd“, target=“controlFlowLinkEnd“, target.decoration =“arrow“, style=“solid“, label=“value“)“ deutlich. Die Attribute „source“ und „target“ beschreiben, von welchem Typ die Anfangs- und Endknoten dieser Kante sein dürfen. Wie an diesem Beispiel zu erkennen ist, müssen die Kantenbeziehungen bidirektional angegeben werden. Die Referenz der Klasse „ControlFlowLink“ auf eine Klasse des Typs „ControlFlowLinkEnd“ zeigt auf „controlFlowLinkEnd“ der gleichnamigen Klasse. In der Klasse „ControlFlowLinkEnd“ zeigt die Referenz wiederum auf „ControlFlowEndBiEnd“ der Klasse „ControlFlowLink“.

```
public class ServiceCreate extends AbstractCreateFeature {

  public ServiceCreate(IFeatureProvider fp) {
    super(fp, "Service", "Create Service");
  }

  @Override
  public boolean canCreate(ICreateContext context) {
    return context.getTargetContainer() instanceof Diagram;
  }

  @Override
  public Object[] create(ICreateContext context) {

    Service newClass = TwdFactory.eINSTANCE.createService();
    getDiagram().eResource().getContents().add(newClass);
    newClass.setName("Service");

    Object bo = getBusinessObjectForPictogramElement(getDiagram());
    if (bo instanceof WorkflowDefinitionType) {
```

```
      ((WorkflowDefinitionType) bo).getServices().add(newClass);
    }

    addGraphicalRepresentation(context, newClass);

    return new Object[] { newClass };
  }

}
```

Listing A.3: Create-feature eines Service

```
public class ServiceAdd extends AbstractAddShapeFeature {

  public ServiceAdd(IFeatureProvider fp) {
    super(fp);
    // TODO Auto-generated constructor stub
  }

  @Override
  public boolean canAdd(IAddContext context) {
    if (context.getNewObject() instanceof Node) {
      if (context.getTargetContainer() instanceof Diagram)
        return true;
    }
    return false;
  }

  @Override
  public PictogramElement add(IAddContext context) {

    ContainerShape shape = PictogramElementAddFactory.getInstance()
        .createPictogramElement(context, "Service", NodeShapes.RECT_ROUNDED,
            manageColor(0, 0, 0), manageColor(0, 150, 200),
            manageColor(0, 0, 0));


    link(shape, context.getNewObject());

    return shape;
  }
}
```

Listing A.4: Add-feature eines Service. Die Methode „link(shape, context.getNewObject)“ verknüpft die grafische Repräsentation des Service mit dem zugehörigen Modellobjekt. „PictogramElementFactory“ ist eine selbstdefinierte Hilfsklasse, die die grafische Darstellung erzeugt.

B. Modelltransformation

In diesem Kapitel wird dargestellt, wie die Transformation von Xtext zu GMF und von GMF zu Xtext aufgebaut ist. Aufgrund gewisser Einschränkungen beim Einsatz von EuGENia war es nötig, zwei Metamodelle – eines für Xtext und eines für GMF – für die Workflow-Beschreibung zu entwickeln. Diese Metamodelle unterschieden sich in ihrer Struktur nur in geringem Maße, sodass eine Transformation von einem Modell zum anderen sinnvoll und umsetzbar erschien. Für eine einwandfreie Interaktion zwischen den beiden Metamodellen wurde daher eine Transformationmöglichkeit angeboten. Im Folgenden werden die für Eclipse4Bio betrachteten Transformationssprachen *ATL*, und *XPAND2* vorgestellt. Es stellte sich heraus, dass keine dieser Sprachen den Anforderungen von Eclipse4Bio genügen konnte. Daher wurde eine angepasste Transformation in Java implementiert.

Im Verlauf der Projektarbeit zeigte sich jedoch, dass die Verwendung von EuGENia zu viele Nachteile mit sich bringt und vor allem kontinuierliche Änderungen an der Transformation benötigt. Daher wurde (wie in 4.4 auf Seite 42 beschrieben) EuGENia durch Graphiti ersetzt. Eine Verwaltung zweier Metamodelle wird dadurch – ebenso wie eine Transformation – hinfällig.

B.1. Das Transformationskonzept

Bei einer Transformation handelt es sich um die Erstellung eines oder mehrere Zielmodelle aus einem oder mehreren gegebenen Quellmodellen. Ausgehend von den beiden entwickelten Metamodellen in Eclipse4Bio ist eine Abbildung zwischen den Modellen definiert worden. Daraus wurde eine konkrete Transformation auf der M2-Ebene (vergleiche Abbildung 2.1 auf Seite 19) zwischen den Metamodellen implementiert. Die Transformation auf der M2-Ebene wandelt eine Instanz des GUI-Metamodells in eine Instanz des Modellierungs-Metamodells sowie umgekehrt. Hierzu stellt Eclipse bereits zwei wichtige Modelling-Projekte zur Verfügung:

M2M-Projekt Eine Transformation zur Verfeinerung oder Anreicherung von Modellen setzt auf einem Quellmodell auf und erzeugt ein Zielmodell, welches wiederum als Ausgangsmodell für weitere Transformationen dienen kann. Solche Transformationen heißen Model-to-Model-Transformationen (M2M-Projekt [5]). Transformationen werden durch die in die Infrastruktur integrierten Transformation-Engines ausgeführt.

```
2  -- @path xTextModel=/workflowDefinitionLanguage/model/WorkflowDefinition.
       ecore
   -- @path guiMetaModell=/workflowDefinitionLanguageForGMF/model/
       WorkflowDefinitionWithXORProblem.ecore
4
   module GMFModelToXTextTransformation;
6  create xTextInput : xTextModel from guiInput : guiMetaModell;
```

Listing B.1: Erstellung xTextModel

M2T-Projekt Das M2T-Projekt [6] (Model-to-Text) beschäftigt sich mit der Generierung eines textuellen Artefakts aus den gegebenen Modellen. Diese Modelle werden in einer domänenspäzifischen Sprache erstellt.

Eclipse4Bio benutzt die M2T-Transformation, da die M2M-Transformation aufgrund der Komplexität der Transformationssprache technisch schwer umsetzbar ist.

B.2. M2T-Transformation

Die im XMI-Format (XML Metadata Interchange) dargestellten Modelle werden durch die M2T-Transformationen bearbeitet. Um diese Transformation durchzuführen stehen verschiedene Transformationssprachen zur Verfügung. Eine Auswahl dieser Sprachen – *ATL* und *XPAND2* – wird im Folgenden betrachtet.

B.2.1. Atlas Transformation Language

Die *Atlas Transformation Language* (ATL) [47] ist eine mögliche Technik, um Transformationen auf Modellen durchzuführen. Sie ist eine Modelltransformationssprache, die sowohl für Modelle als auch für textuelle, konkrete Syntax verwendet werden kann. Im Bereich des Model-Driven Engineering (MDE) ermöglicht ATL Entwicklern aus gegebenen Modellen andere Modelle zu erzeugen. Entwickelt für die Eclipse-Plattform, bietet die in ATL integrierte Entwicklungsumgebung eine Reihe von Standardentwicklungstools (Syntax-Highlighting, Debugger, etc.), welche die Entwicklung von ATL-Transformationen vereinfachen sollen. Eine ATL-Datei besteht aus mehreren Teilen:

header definiert den Namen des Moduls und die entsprechenden Quell- und Zielmodelle. Die Syntax für den Header-Bereich ist in Eclipse4Bio definiert und wird in Listing B.1 veranschaulicht.

- *module* definiert den Namen des Moduls. Dieser entspricht in Eclipse4Bio-Projekten dem Dateinamen.

```
    helper def : xTextStartElementControlFlowLinkName(startElement :
        xTextModel!StartElement) : String =
2     if(startElement.controlFlow.size() = 1) then
        startElement.controlFlow.at(0).name
4     else
        ''
6     endif
    ;
```

Listing B.2: helper Definition

- *create ... from ...* gibt an, von welchem Modell in welches Modell transformiert werden soll.

import ist ein optionales Element, um eine oder mehrere bereits existierende ATL-Bibliotheken zu importieren. Diese optionale Funktion kommt in Eclipse4Bio nicht zum Einsatz.

helpers können als Java-Methoden betrachtet werden. Sie ermöglichen ATL, Codeabschnitte an verschiedenen Stellen aufzurufen. Die Definition der ATL-helpers wird in Listing B.2 dargestellt.

- In der ersten Zeile werden der Name der Methode, alle benötigten Parameter und der Rückgabewert definiert.
- Nachfolgende Zeilen definieren den Code der Methode.
- Jedes Statement muss einen Wert zurückliefern, da alle Statements automatisch als Rückgabewert interpretiert werden.

rules definieren den Ablauf der Transformation. Bei ATL wird zwischen drei verschiedenen Arten von Regeln unterschieden, die für die beiden unterschiedlichen Programmiermodi – deklarative und imperative Programmierung – von ATL zur Verfügung gestellt werden. Für die Transformation kommt die deklarative Variante zum Einsatz.

Matched rules stellen den Hauptbestandteil der deklarativen Transformation dar. Es ist möglich zu spezifizieren:

- für welche Quellelemente Zielelemente erzeugt werden müssen,
- wie die erzeugten Zielelemente initialisiert werden müssen.

Die Implementierung der Transformation von Xtext zu GMF führte zu einem Problem von ATL: Durch die *helper* kann nicht auf die bidirektionale Struktur zugegriffen werden. Es ist nicht möglich, die benötigten Daten auszulesen und so das zweite Modell zu erzeugen. Aus diesem Grund war ATL für die Anforderungen von Eclipse4Bio nicht geeignet.

```
module workflow.WorkflowDefinitonGenerator
import org.eclipse.emf.mwe.utils.*

var targetDir = "src-gen"
var fileEncoding = "UTF-8"
var modelPath = "src/model"

Workflow {
  component = org.eclipse.xtext.mwe.Reader {
    path = modelPath
    register = de.tu_dortmund.cs.pg.eclipse4bio.
        WorkflowDefinitonStandaloneSetup {}
    load = {
      slot = "workflowDefinitionType"
      type = "WorkflowDefinitionType"
    }
  }

  component = org.eclipse.xpand2.Generator {
    metaModel = org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel{}
    expand = "templates::Template::main FOREACH workflowDefinitionType"
    outlet = {
      path = targetDir
    }
    fileEncoding = fileEncoding
  }
}
```

Listing B.3: Definition eines WorkflowDefinitionGenerators

B.2.2. XPAND2

Als eine weitere Möglichkeit, die Transformation zwischen den beiden Modellen in Eclipse-4Bio umzusetzen, wurde XPAND2 [55] in Betracht gezogen. XPAND2 ist eine Templatesprache, die Konzepte wie Polymorphie unterstützt und eine generische Transformation von einem gegebenen Model in ein anderes Format bietet. Beim Durchlauf vom Quellformat wird ein neues Zielformat erzeugt.

Der Generator für die Eclipse4Bio Workflow-Definition, der ein gegebenes Modell mit der Hilfe eines in XPAND2 erstellten Templates in eine XML-Datei transformieren kann, ist vom Typ *MWE2* und ist in Listing B.3 definiert.

Es kommen folgende Schlüsselwörter zum Einsatz:

module muss einen referenzierbaren Namen haben. Der Typ des Moduls wird von der Wurzel abgeleitet.

import importiert die für die Generierung benötigten Pakete.

var ermöglicht beliebige Informationen in Form von Eigenschaften zu extrahieren, um den Workflow übersichtlich zu halten und eine einfachere externe Anpassung zu gewährleisten.

component unterteilt sich in:

Reader wird gebraucht um die durch Xtext definierte Sprache zu initialisieren, Xtext-Dateien zu lesen und bestimmte Elemente aus den Modellen zu extrahieren.

Generator ist für die XPAND2-Templatesprache zuständig. Der Generator muss das richtige Template und das passende Metamodell aufrufen.

path legt den Pfad für das Modell fest.

register dient der Xtext-Sprachinitialisierung. Diese wird durch die Angabe einer beliebigen Anzahl von Setup-Implementierungen durchgeführt. In Eclipse4Bio ist die generierte WorkflowDefinitonStandaloneSetup registriert. So wird sichergestellt, dass die Infrastruktur der Domain-Modellsprache richtig eingestellt ist.

load gibt an, welche Elemente aus den geladenen Ressourcen bereitgestellt werden müssen.

slot ist der Name, mit dem andere Workflowkomponenten auf die gespeicherten Elemente zugreifen können.

outlet beschreibt, wo die generierte Datei gespeichert werden kann.

Nachdem Änderungen an den beiden Metamodellen durchgeführt wurden, stellte sich heraus, dass die Ausführung der mwe2-Datei keine XMI-Datei erzeugte. Trotz expliziter Registrierung der Metamodelle konnte diese nicht mehr richtig initialisiert werden. Aus diesem Grund war die XPAND2-Transformation für Eclipse4Bio nicht praktikabel und die Transformation wurde manuell in Java implementiert.

B.2.3. Transformationsimplementierung in Java

Als letzte Alternative wurde eine in Java implementierte Transformation entwickelt. Die einzelnen Komponenten eines gegebenen Metamodells konnten programmatisch ausgelesen und in ein gewünschtes Format ausgegeben werden. Die Transformation wurde mit Hilfe einer StringBuilder-Klasse [42] realisiert.

public-Methoden Eine Convert-Methode für die Umwandlung einer gegebenen Datei und die Main-Methode.

private-Methoden Diese sind für die Transformation einzelner Bestandteile des gegebenen Metamodells zuständig.

Die Implementierung der Metamodelltransformation mittels StringBuilder wurde nicht zu Ende geführt, da eine Umstellung auf Graphiti bei der GUI-Entwicklung stattgefunden hat. Dadurch entfiel das zweite Metamodell und die Notwendigkeit einer Transformation.

C. Validierung mit EVL

Im grafischen Editor erstellte Workflows werden einem Validierungsprozess unterzogen, um Inkonsistenzen zum Metamodell anzuzeigen und den Benutzer so auf Fehler hinzuweisen. Da der grafische Editor in Eclipse4Bio in den ersten Versionen mit EuGENia erstellt wurde, erwies sich zur Umsetzung der Validierung die Sprache *Epsilon Validation Language* als praktikabel. Mit ihr lassen sich Regeln aufstellen, gegen die der erstellte Workflow geprüft wird. Im Fehlerfall können zusätzlich *QuickFixes* implementiert werden, die Fehler automatisch korrigieren [61]. Es hat sich jedoch herausgestellt, dass bei der Umsetzung mit EuGENia einige Funktionen nicht realisiert werden konnten, wie in Kapitel 4 beschrieben ist. Daher wird die Validierung nun nicht mehr von EVL übernommen. Trotzdem wird der ausgearbeitete, aber nun nicht mehr verwendete Ansatz, in diesem Kapitel dargestellt.

Mit Hilfe des Diagrammeditors (siehe Kapitel 4) können Bioinformatik-Workflows über ein grafisches Interface erstellt und bearbeitet werden. Es muss sichergestellt werden, dass die erzeugten Diagramme dem Metamodell der Workflowdefinition, das in Kapitel 2 beschrieben wird, entsprechen. Jedoch können einige Eigenschaften nicht allein über das Metamodell sichergestellt werden. Diese müssen daher mittels Validierung überprüft werden. Dies geschah in einer frühen Version von Eclipse4Bio mittels der sogenannten Epsilon Validation Language, auf die in Abschnitt C.2 eingegangen wird.

Auf Metamodellebene lassen sich Kardinalitäten von zueinander gehörenden Elementen angeben. Bei der Generierung eines Workfloweditors mittels EuGENia werden auf Basis dieser Kardinalitäten einfache Validierungsregeln erzeugt. Wenn bei der Erstellung eines Workflows im grafischen Editor ein Kardinalitätsfehler auftritt, beispielsweise das Fehlen eines benötigten Attributs, erhält der Nutzer eine entsprechende Fehlermeldung. Durch eine solche Validierung lassen sich jedoch keine komplexen Bedingungen überprüfen, etwa das Erreichbarkeitsproblem, dass in einem Workflow vom Startelement alle anderen Elemente erreichbar sind. Außerdem ist es hilfreich, QuickFixes anzubieten, die Fehler automatisch in vordefinierter Art auflösen können. Dies ist ebenfalls nicht direkt im Metamodell implementierbar.

Um diese genannten Funktionalitäten zu realisieren, wurde die Validierung durch die Epsilon Validation Language realisiert, die auf der *Epsilon Object Language* aufbaut [61]. Im Folgenden werden die verwendeten Sprachen dargestellt und die Umsetzung der Validierung in Eclipse4Bio erläutert. Neben der Validierung der grafischen Workflowmodelle im Diagrammeditor existiert eine zweite Stufe der Validierung auf Ebene der textuellen

```
operation listeZaehlen(auswerten : Boolean, modellTyp : String) : Integer {
  if (auswerten = false) return 0;
  else {
    var meineListe = new Sequence;
    meineListe.addAll(_Model.getAllOfType(modellTyp));
    return meineListe.size();
  }
}
```

Listing C.1: Beispieloperation in der Epsilon Object Language

Darstellung der Workflows im Xtext-Editor, der in Kapitel 3 beschrieben wurde. So wird sichergestellt, dass im Quellcode erstellte oder geänderte Prozesse valide sind. Diese textuelle Validierung ist vom Wegfall der EVL-Validierung nicht betroffen.

C.1. Epsilon Object Language

Die Epsilon Object Language (EOL) ist ein Teil des Epsilon-Projektes [57]. „Epsilon“ steht dabei für „Extensible Platform of Integrated Languages for mOdel maNagement“. Ziel von Epsilon ist es, eine Plattform von konsistenten, interoperablen und anwendungsspezifischen Sprachen bereitzustellen, mit denen Modelle verwaltet werden können. Typische Einsatzzwecke sind Transformation, Generierung, Vergleich, Kombination, Refactoring und Validierung.

EOL ist die Basis für alle anderen in Epsilon enthaltenen Sprachen und bietet Grundfunktionalitäten für den Umgang mit anwendungsspezifischen Modellen an. Allerdings kann EOL auch eigenständig zur Modellierung von Modellen verwendet werden. Listing C.1 stellt ein Beispielprogramm in EOL dar. An diesem werden nun grundlegende Eigenschaften der Epsilon Object Language erläutert.

EOL bietet vier primitive Typen an [61]:

- String
- Integer
- Real
- Boolean

Auf diesen Typen ist ein überschaubarer Satz an Operationen möglich, beispielsweise *substring()* für *String* oder *log()* für *Real*. Diese Typen lassen sich in vier *Map*-Arten zu Sammlungen zusammenfassen, auf denen ebenfalls einige Operationen, wie Hinzufügen und Löschen, möglich sind:

- Bag (nicht einmalig, ungeordnet)

```
//Java-Datei
package meinPaket
public class meineKlasse {
  private String klassenAttribut = "Hello World!";
  public String getAttribut() {
    return klassenAttribut;
  }
}

//EVL-Datei
operation instanzErstellen() : String {
  var meineInstanz = new Native("meinPaket.meineKlasse");
  return meineInstanz.getAttribut();
}
```

Listing C.2: Zugriff auf native Typen

- Sequence (nicht einmalig, geordnet)
- Set (einmalig, ungeordnet)
- OrderedSet (einmalig, geordnet)

Da EOL zur Interaktion mit Modellen entworfen wurde, können Modellelemente als Typen verwendet werden. So kann etwa ein Zugriff auf die Attribute einzelner Elemente des Modells realisiert werden.

EOL soll eine abstrakte Modellsprache sein und keine allgemeine Programmiersprache ersetzen, sodass selbstdefinierbare, komplexe Typen nicht vorhanden sind. Für viele Anwendungsbereiche kann es jedoch nötig sein, komplexere Typen als die in EOL enthaltenen einzusetzen. Dies kann der Fall sein, wenn umfangreiche Berechnungen, Oberflächenentwicklung oder die Kommunikation mit externen Systemen – wie Datenbanken oder entfernten Anwendungen – durchgeführt werden muss. Aus diesem Grund existieren zusätzlich die sogenannten *nativen Typen*. Wie der Name andeutet, stammen native Typen aus der (nativen) Programmierumgebung, auf der Epsilon ausgeführt wird. Das bedeutet beispielsweise für die Java-Variante von EOL, dass auch Java-Klassen instanziiert und genutzt werden können. Listing C.2 zeigt eine Implementierung von nativen Typen in EOL.

C.2. Epsilon Validation Language

Die Epsilon Validation Language, kurz EVL, baut auf EOL auf und dient der Validierung von Modellen [61]. In Eclipse4Bio wurde EVL eingesetzt, um die im grafischen Editor erstellten Workflows zu überprüfen. Mit EVL ist es möglich, gewisse Annahmen für Modelle

zu formulieren und auf ihre Gültigkeit zu evaluieren. Im Folgenden werden die Grundlagen von EVL dargestellt.

Ausdrücke in EVL bestehen aus folgenden Elementen:

Invariant Eine Invariante besteht aus einem identifizierenden Namen und einem Body-Teil, in dem der Check – die Überprüfung der Annahme – stattfindet. Eine solche Überprüfung ist eine in EOL formulierte Anweisung, die sich zu *wahr* oder *falsch* auswerten lassen muss. Sofern die Überprüfung fehlschlägt, wird dem Benutzer ein Error oder eine Warning im Error Log ausgegeben. Diese können mit Hilfe einer Message informativ aufgewertet werden. Die Invariante ist eine abstrakte Superklasse von Constraint und Critique. Welche Art von Invariante gewählt wird, entscheidet darüber, ob Error oder Warning ausgegeben wird. Für eine automatische Fehlerkorrektur können Fixes in einer Invariant definiert werden.

Context Der Context legt fest, auf welche Typen von Modellelementen sich die in ihm enthaltenen Invarianten beziehen. Die Invarianten werden für alle Instanzen der Typen überprüft. Um diese Menge an zu überprüfenden Instanzen einzuschränken, lassen sich vor den Invarianten sogenannte Guards einsetzen.

Guard Guards schränken die Anzahl der zu überprüfenden Elementinstanzen ein und können sowohl innerhalb des Context für alle Invarianten, als auch innerhalb einzelner Invarianten definiert werden.

Fix Mit einer Fix wird dem Nutzer die Möglichkeit geboten, einen Error oder eine Warning automatisch zu beseitigen. Dabei müssen die durchzuführenden Korrekturen in EOL implementiert werden und erscheinen dem Nutzer als QuickFixes.

Constraint Constraints sind vom Typ einer Invariant und werden nach *wahr* oder *falsch* ausgewertet. Sie geben bei Falschauswertung einen Error aus.

Critique Critiques ähneln den Constraints, geben aber nur Warnings aus, die unkritisch für die Modellkonformität sind.

Pre/Post In den Pre- und Post-Bereichen können EOL-Anweisungen ausgeführt werden, bevor oder nachdem Invarianten überprüft werden.

Codeausschnitt C.3 zeigt eine EVL-Validierung mit zwei Constraints und einem Quickfix.

C.3. Umsetzung der Validierung

Im Folgenden wird die Umsetzung der Validierung in Eclipse4Bio dargestellt, die die Konformität der mit dem grafischen Editor erstellten Modelle sicherstellt. EuGENia (siehe

```
context Rechteck {
  constraint hatName {
    check : self.meinName.isDefined()
    message: "Unbenanntes Objekt vom Typ " + self.eClass().name
  }
  constraint istQuadrat {
    guard : self.satisfies("hatName")
    check : self.x = self.y
    message : self.name + " ist kein Quadrat!"
    fix {
      title : "Rechteck zum Quadrat konvertieren."
      do : self.x = self.y
    }
  }
}
```

Listing C.3: Beispielvalidierung in der Epsilon Validation Language

Kapitel 4) generiert Teile der Validierung aus dem Metamodell. Dazu gehören die Kardinalitäten von Objekten, wie die Notwendigkeit eines Namens oder das Ausgehen genau eines Sequenzflusses von einem Service. Wenn ein Benutzer bei der Erstellung eines Workflows im grafischen Editor gegen diese Bedingungen verstößt, werden im Error Log entsprechende Meldungen angezeigt, die von EuGENia automatisch generierte Informationstexte enthalten.

Durch die automatisch erzeugte Fehlererkennung lassen sich jedoch nicht alle notwendigen Validierungen abdecken. Des Weiteren ist es nicht möglich, angepasste Fehlertexte anzuzeigen und QuickFixes anzubieten, mit denen der Nutzer Fehler automatisch korrigieren kann. Folgende Validierungen sind zusätzlich nötig und wurden mit der Epsilon Validation Language (EVL) umgesetzt:

ConditionOut Conditions müssen genau einen ausgehenden *true*-Sequenzfluss – also die Fortsetzung des Workflowablaufs, wenn die Condition *true* wird – haben. Analog ist ein ausgehender *false*-Sequenzfluss nötig. *true* und *false* werden durch den *Value*-Parameter eines Sequenzflusses festgelegt.

XORProblem Diese Validierung stellt sicher, dass die Anzahl der ausgehenden Sequenzflüsse für die einzelnen Elemente des Modells richtig ist. Dies wird teilweise durch die von EuGENia generierte Validierung abgedeckt, jedoch lässt sich mit dieser zum Beispiel nicht explizit die Kardinalität *2* bei Conditions ausdrücken. Der Eclipse4Bio-interne Name *XORProblem* kommt von der Notwendigkeit, dass für eine Condition nur entweder die eine oder die andere ausgehende Kante aktiviert werden darf, jedoch nicht beide.

Reachability Für die von Eclipse4Bio ausführbaren Workflows muss sichergestellt werden, dass von einem Startelement aus immer ein Endelement erreicht wird, der Workflow also nach einer gewissen Abfolge von Schritten endet. Reachability stellt für Startelemente fest, ob von ihnen aus auf jedem möglichen Pfad ein Endelement erreicht werden kann. Außerdem werden unerreichbare Workflowelemente erkannt.

Die Reachability-Invariante entspricht der *checkPathToElements*-Methode der textuellen Validierung, wie in Kapitel 3 dargestellt. Da für die grafische Validierung EVL genutzt wird, kann nicht unmittelbar auf die textuelle Validerung zugegriffen werden. Der eingesetzte Floyd-Warshall-Algorithmus [17] musste daher in EVL umgesetzt werden. Bei der Umsetzung des Reachability-Algorithmus in EVL ergab sich folgendes Problem: Wie in Abschnitt C.1 beschrieben, existieren in EVL neben den vier primitiven Typen nur vier Map-Typen. Für den Algorithmus wird jedoch eine zweidimensionale Datenstruktur benötigt. Es bot sich an, auf die nativen Typen zuzugreifen und eine solche Datenstruktur in Java anzulegen.

Zusätzlich zu den drei oben genannten Validierungen wurden für einige Parameter QuickFixes umgesetzt. Diese dienen der Erkennung von nicht gesetzten Parametern, wie beispielsweise dem Namen von Modellelementen. Sofern ein Parameter nicht gesetzt wird, steht dem Nutzer ein QuickFix zur Verfügung, der dem Parameter einen Standardwert zuordnet.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] *DiscoveryNet*. Website. http://www.discovery-on-the.net/.

[2] *Picard*. Website. http://sourceforge.net/projects/picard/.

[3] *SCUFL*. Website. http://www.gridworkflow.org/snips/gridworkflow/space/SCUFL.

[4] Apache: *Apache ODE*. Website. http://ode.apache.org/.

[5] Eclipse Foundation: *Eclipse - Model To Model (M2M)*. Website. http://www.eclipse.org/m2m/.

[6] Eclipse Foundation: *Eclipse - Model To Text (M2T)*. Website. http://www.eclipse.org/modeling/m2t/.

[7] Eclipse Foundation: *Javadoc Dokumentation org.eclipse.emf.ecore*. Website. http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html.

[8] Eclipse Foundation: *JET (Java Emitter Templates)*. Website. http://www.eclipse.org/modeling/m2t/?project=jet.

[9] Eclipse Foundation: *EMF Documentation*. Website, 2006. http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html#details.

[10] Eclipse Foundation: *Xtext Documentation*. Website, Juni 2011. http://www.eclipse.org/Xtext/documentation/2_0_0/Xtext-Documentation.pdf.

[11] Eclipse Foundation: *Xtext wiki*. Website, März 2012. http://wiki.eclipse.org/Xtext.

[12] Eclipse Foundation: *Xtext wiki*. Website, März 2012. http://www.eclipse.org/modeling/m2t/?project=xpand.

[13] Eclipse Foundation: *Xtext wiki*. Website, März 2012. http://wiki.eclipse.org/Modeling_Workflow_Engine_

[14] Eclipse Foundation: *Xtext wiki*. Website, März 2012. http://www.eclipse.org/epsilon/doc/evl/.

[15] eMundo GmbH: *Java Workflow Tooling (JWT).* Website. http://www.e-mundo.de/de/forschungpublikationen/agilpro.html.

[16] eMundo, Universität Augsburg: *Java Workflow Tooling Extensions.* Website. http://wiki.eclipse.org/JWT_Extensions.

[17] Fachhochschule Flensburg: *Floyd-Warshall-Algorithmus.* Website. http://www.iti.fh-flensburg.de/lang/algorithmen/graph/warshall.htm.

[18] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns.* Addison-Wesley, Boston, MA, January 1995.

[19] Google: *Google Web Toolkit.* Website. http://code.google.com/webtoolkit/.

[20] IBM: *Rational Rose.* Website. http://www-01.ibm.com/software/awdtools/developer/rose/.

[21] jBOSS Community: *Java Business Process Management.* Website. http://www.jboss.org/jbpm.

[22] Jkwchui: *ChIP-sequencing.* Website Wikipedia. http://upload.wikimedia.org/wikipedia/commons/1/15/ChIP-sequencing.svg.

[23] LendingTree, LLC.: *MyGeneTree.* http://www.mygenetree.com/articles/types-of-dna-tests/snps.php, Letzter Zugriff: 10. Februar 2012.

[24] Li, Heng und Richard Durbin: *Fast and accurate short read alignment with Burrows–Wheeler transform.* Bioinformatics, 25(14):1754–1760, 2009.

[25] Li, Heng, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis und Richard Durbin: *The Sequence Alignment/Map (SAM) Format and SAMtools.* Bioinformatics, 25(16):2078–2079, 2009.

[26] McKenna, Aaron, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly und et al.: *The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data.* Genome Research, 20(9):1297–1303, 2010.

[27] OASIS: *Web Services Business Process Execution Language Version 2.0.* Website. http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm.

[28] Object Management Group: *Business Process Model and Notation.* Website. http://www.omg.org/spec/BPMN/index.htm.

[29] Object Management Group: *CORBA Component Model, v4.0.* PDF, April 2006. http://www.omg.org/spec/CCM/4.0/PDF.

[30] Object Management Group: *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure.* Website, November 2010. http://www.omg.org/spec/UML/2.4/Infrastructure/Beta2/PDF/.

[31] Object Management Group: *Business Process Model and Notation (BPMN) Version 2.0.* PDF, Januar 2011. http://www.omg.org/spec/BPMN/2.0/PDF.

[32] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1.* PDF, Januar 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

[33] Object Management Group: *MOF XMI specification Version 2.4.1.* PDF, August 2011. http://www.omg.org/spec/XMI/2.4.1/PDF.

[34] Object Management Group: *OMG's MetaObject Facility.* Website, Juni 2011. http://www.omg.org/mof/.

[35] Object Management Group: *OMG's MetaObject Facility.* Website, Juni 2011. http://www.omg.org/spec/MOF/2.0/PDF/.

[36] Object Management Group: *OMG's MetaObject Facility Core specification.* PDF, August 2011. http://www.omg.org/spec/MOF/2.4.1/PDF.

[37] Object Management Group: *UML Infrastructure specification.* PDF, August 2011. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF.

[38] Object Management Group: *UML Superstructure specification.* PDF, August 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[39] Object Management Group: *Model Driven Architecture.* Website, September 2012. http://www.omg.org/mda/specs.htm.

[40] Object Management Group: *Object Management Architecture.* Website, März 2012. http://www.omg.org/oma/.

[41] Oracle: *Doclet Javadoc.* Website. http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/doclet/s

[42] Oracle: *The StringBuilder Class.* Website. http://docs.oracle.com/javase/tutorial/java/data/buffers

[43] School of Computer Science, University of Manchester, UK: *Taverna.* Website. http://www.taverna.org.uk/.

[44] Spring Source: *Spring Batch.* Website. http://static.springsource.org/spring-batch/.

[45] StringTemplate: *StringTemplate*. Website. http://www.stringtemplate.org/.

[46] The Apache Software Foundation: *Apache Maven Project*. Website. http://maven.apache.org/.

[47] The Apache Software Foundation: *ATL - a model transformation technology*. Website. http://www.eclipse.org/atl/.

[48] The Apache Software Foundation: *Eclipse Update Site*. Website. http://update.eclipse.org/updates/.

[49] The Apache Software Foundation: *Epsilon Object Language*. Website. http://www.eclipse.org/epsilon/doc/eol/.

[50] The Apache Software Foundation: *EuGENia*. Website. http://www.eclipse.org/gmt/epsilon/doc/eugenia/.

[51] The Apache Software Foundation: *Extended Editing Framework*. Website. http://wiki.eclipse.org/EEF.

[52] The Apache Software Foundation: *Graphical Modeling Project (GMP)*. Website. http://www.eclipse.org/modeling/gmp/.

[53] The Apache Software Foundation: *Graphiti*. Website. http://www.eclipse.org/graphiti/.

[54] The Apache Software Foundation: *Java Workflow Tooling (JWT)*. Website. http://eclipse.org/jwt/.

[55] The Apache Software Foundation: *Model To Text (M2T) – Xpand*. Website. http://www.eclipse.org/modeling/m2t/?project=xpand.

[56] The Eclipse Foundation: *Emfatic*. Website. http://www.eclipse.org/modeling/emft/?project=emfatic.

[57] The Eclipse Foundation: *Epsilon*. Website. http://www.eclipse.org/gmt/epsilon/.

[58] The Eclipse Foundation: *Overview of Graphiti*. Website. http://www.eclipse.org/graphiti/documentation/overview.php.

[59] The Eclipse Foundation: *Overview of Graphiti*. Website. http://www.eclipse.org/graphiti/documentation/overview.php.

[60] The Epsilon Foundation: *Eclipse Plug-in Development Environment*. Website. http://www.eclipse.org/pde/.

[61] The Epsilon Foundation: *The Epsilon Book*. Website. http://www.eclipse.org/gmt/epsilon/doc/book/.

[62] The Epsilon Foundation: *Tycho - Building Eclipse plug-ins with maven*. Website. http://eclipse.org/tycho/.

[63] Universität Augsburg: *Java Workflow Tooling (JWT)*. Website. http://www.informatik.uni-augsburg.de/de/lehrstuehle/swt/vs/projekte/mde/jwt/.

[64] Universität Augsburg, eMundo GmbH: *AgilPro Beta*. Website. http://sourceforge.net/projects/agilpro/files/.

[65] Vasa Curcin, Moustafa Ghanem, Patrick Wendel: *Heterogeneous Workflows in Scientific Workflow Systems*. ICCS, III:204–211, 2007.

[66] Vogel, Lars und Dominik Zapf: *Eclipse PDE Build - Tutorial*. Website. http://www.vogella.de/articles/EclipsePDEBuild/article.html.

[67] Workflow Management Coaltition: *XML Process Definition Language*. Website. http://www.wfmc.org/xpdl.html.

[68] Y, Zhang, Liu T, Meyer CA, Eeckhoute J, Johnson DS, Bernstein BE, Nussbaum C, Myers RM, Brown M, Li W und Liu XS: *Model-based Analysis of ChIP-Seq (MACS)*. Genome Biology, 9(9):R137, 2008.