Diplomarbeit

Automatische Generierung von Prozessen im jABC

Stefan Naujokat

16. September 2009

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5 für Programmiersysteme

Gutachter

Prof. Dr. Bernhard Steffen
Dipl.-Inform. Christian Kubczak

Hiermit erkläre ich, Stefan Naujokat, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zitate habe ich stets kenntlich gemacht.

Dortmund, den 16. September 2009

_____

Stefan Naujokat

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

BLAST Basic Local Alignment Search Tool

DAG Directed Acyclic Graph

DNA Deoxyribonucleic Acid

EMBOSS European Molecular Biology Open Software Suite

ETI Electronic Tool Integration

GUI Graphical User Interface

jABC Java Application Building Center

jETI Java Electronic Tool Integration

LPC Lightweight Process Coordination

LTL Linear Time Logic

NFA Nondeterministic Finite Automaton

OTA One-Thing-Approach

OWL Web Ontology Language

PLTL Propositional Linear Time Logic

RNA Ribonucleic Acid

SIB Service Independent Building Block

SLTL Semantic Linear Time Logic

SOA Service-Oriented Architecture

TSM Transition System Model

UID Unique Identifier

XMDD Extreme Model Driven Design

XML Extensible Markup Language

1 Einleitung

Seit etwa 6.000 Jahren gibt es das Rad. Und noch heute wird es in der Softwareentwicklung täglich neu erfunden, indem dieselben Programmfeatures immer wieder nahezu identisch implementiert werden. Es gibt unterschiedliche Ansätze, diesem Phänomen entgegenzuwirken. Das beginnt beim einfachen *Copy & Paste* des Quelltextes und geht über das Verteilen von *Softwarebibliotheken* bis hin zu externen *Services*. Letztere sind Kernstück von *Serviceorientierten Architekturen* (*SOA*) [RHS05] und der *Lightweight Process Coordination* (*LPC*) [MS04]. All diese Ansätze haben jedoch gemeinsam, dass eine große Menge von Bibliotheken und Services schnell unübersichtlich wird. Ein einsteigender Entwickler oder Modellierer muss sich trotz eventuell vorhandener Sortier- und Filterfunktionen zunächst langwierig orientieren.

Dies legt die Idee nahe, die Orchestrierung von Services automatisieren zu wollen, um den Modellierer bei dieser Orientierung zu unterstützen. Bereits vor über zehn Jahren bot die *Electronic Tool Integration Platform* (*ETI*) [SMB97] eine solche Funktionalität als Onlinedienst. So konnten Anwender Rahmenanforderungen angeben, aus denen Toolsequenzen synthetisiert wurden. Da sich seitdem die Technologien – insbesondere in Hinblick auf Internetdienste – stark weiterentwickelt haben, ist diese Plattform mittlerweile nicht mehr in Betrieb.

Die vorliegende Diplomarbeit stellt eine Neuimplementierung und Erweiterung dieser Konzepte der ETI-Plattform vor. Umgesetzt wurde sie als Plugin für das jABC, ein Framework zur grafischen Modellierung von Prozessen. Darin wird die Möglichkeit geschaffen, Modellteile als *Lose Spezifikation* zu definieren. Das Plugin führt daraufhin eine Synthese aus und schlägt dem Anwender mögliche Konkretisierungen vor. Darüber hinaus können mithilfe von Formeln in einer temporalen Logik weitere Anforderungen an die Synthese gestellt werden. Der Anwender erhält damit eine Experimentierplattform, die ihn mit iterativer Verfeinerung seiner Spezifikation zum gewünschten Ergebnis leiten kann, ohne dass er sich mit allen Details der vorhandenen Services auseinandersetzen muss.

In Anlehnung an seine „hellseherischen“ Fähigkeiten erhält das Plugin den Namen PROPHETS. Dieses Akronym steht für „Process Realization and Optimization Platform using a Human-readable Expression of Temporal-logic Synthesis“. Dabei deutet der Name auch den zweiten wichtigen Aspekt dieses neuen Ansatzes an: Der Benutzer soll sich nicht mit den formalen Details von Temporallogik und Synthese auseinandersetzen müssen.

Die in der Modellierung nach LPC geforderte Trennung zwischen Anwendungsexperte und Programmierer bleibt mit PROPHETS erhalten. Es findet lediglich eine Erweiterung um die Rolle des Domänenexperten statt. Dieser ermöglicht durch Modellierung

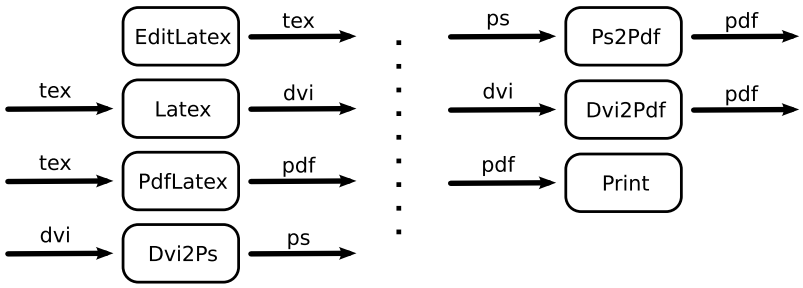


Abbildung 1.1: Übersicht der Domäne des LaTeX-Beispiels

domänenspezifischer Informationen, dass die Synthese angewendet werden kann.

Zur leichteren Veranschaulichung wird der Hauptteil der Arbeit durch ein Beispiel begleitet, welches durch seine Einfachheit den primären Fokus auf die mit PROPHETS umgesetzten Konzepte ermöglicht. Dabei handelt es sich um ein Szenario über Services zum Erstellen von Dokumenten aus LaTeX-Dateien. Abbildung 1.1 zeigt die verfügbaren Dienste. Die ein- und ausgehenden Pfeile bezeichnen jeweils die Ein- und Ausgabetypen. Mit `EditLatex` kann eine tex-Datei erzeugt werden, die dann weiterverarbeitet wird. Die übrigen Services verhalten sich intuitiv gemäß ihrem Namen und den dazugehörigen Kommandozeilenbefehlen.

Die Spezifikation des Modellierers könnte an dieser Stelle so aussehen, dass ein mit `EditLatex` erstelltes Dokument ausgedruckt werden soll. Er würde also `EditLatex` als Start und `Print` als Ziel festlegen. Wie man leicht sieht, führt die Zwischenschaltung von `PdfLatex` zu dem gewünschten Ergebnis, aber auch die längere Sequenz (`Latex`, `Dvi2Ps`, `Ps2Pdf`) liefert einen der Anforderung gemäßen korrekten Prozess.

Folgende Konventionen für die Darstellung von Begriffen werden verwendet:

- Namen von Javaklassen und SIBs werden in dicktengleicher Schriftart gesetzt (zum Beispiel `EditLatex`).
- Für die Darstellung von symbolischen Typnamen werden Kapitälchen verwendet (zum Beispiel PDF).
- Begriffe, die im Index auftauchen sowie die Namen der SIB-Klassen des verallgemeinerten Syntheseprozesses werden kursiv gesetzt (zum Beispiel *Synthesis Algorithm*).

Die Arbeit ist in sieben Kapitel aufgeteilt. Zunächst wird in Kapitel 2 „Grundlagen“ die Basis für das Verständnis der folgenden Kapitel gelegt. Auf diesen Grundlagen aufbauend werden in Kapitel 3 die Ideen von PROPHETS vorgestellt und seine Konzepte entwickelt. Danach folgen zwei Kapitel zur Umsetzung dieser Konzepte. Kapitel 4 „Umsetzung auf Benutzerebene“ zeigt die Benutzersicht, Kapitel 5 „Syntheseprozessmodellierung“ die Sicht des Entwicklers von Syntheseprozessen. Zwei Fallstudien, die über das zuvor genannte Beispiel hinausgehen, wurden im Rahmen der Arbeit durchgeführt. Die Ergebnisse finden sich in Kapitel 6 „Fallstudien“. Abschließend liefert Kapitel 7 eine Zusammenfassung der Ergebnisse dieser Arbeit sowie einen Ausblick auf mögliche Erweiterungen.

2 Grundlagen

Die Konzepte von PROPHETS sind eng mit dem jABC und dessen wissenschaftlichen Hintergrund verbunden. Daher werden diese zunächst vorgestellt. Weiterhin bildet die Synthese die zweite wichtige Grundlage, der sich der darauffolgende Abschnitt widmet. Ein Abschnitt zur Datenflussanalyse schließt dieses Kapitel ab. Sie findet sowohl in der Bestimmung von Synthesestarttypen als auch in der nachträglichen Verfikation bestehender Modelle Anwendung.

2.1 Modellierung von Prozessen

Grafische Modellierung von Prozessen ist ein immer populärer werdendes Konzept in der Softwareentwicklung. Am Lehrstuhl für Programmiersysteme der TU Dortmund befindet sich bereits seit etwa 15 Jahren eine entsprechende Software in Anwendung und Weiterentwicklung. Die aktuelle Version dieses Frameworks ist das Java Application Building Center (jABC). Nach einer Einführung in die dahinterstehenden Entwicklungsparadigmen werden Architektur, Komponenten und Plugins des jABC vorgestellt.

Die Electronic Tool Integration Platform (ETI) ist ein zusammen mit früheren Versionen des jABC entwickeltes Onlinesystem. Neben anderen Funktionen bot sie bereits die Möglichkeit, Sequenzen von Tools durch Synthese zu erzeugen. Da die Synthesekonzepte von PROPHETS darauf aufbauen, schließt dieser Abschnitt mit einer Vorstellung von ETI ab.

2.1.1 Paradigmen

Die *Lightweight Process Coordination* (*LPC*) [MS04] beschreibt einen modellbasierten Softwareentwicklungsprozess[1], der auf einer grafischen Notation aufbaut. Die damit dargestellten Flussgraphen bilden den Grundstein für ein leicht verständliches Konzept zur Modellierung von Geschäftsprozessen. Die Notwendigkeit eines solchen Verfahrens begründet sich darin, dass die Experten für die Geschäftsprozesse einer Firma nicht zwingend programmiertechnische Kenntnisse besitzen. Im Gegenzug sind Programmierer selten mit diesen Geschäftsprozessen vertraut. Zudem hat die häufig verwendete Spezifikation von Anforderungen in einem unstrukturierten Textdokument keine klar definierte Semantik.

[1] vergleiche auch [KM05]

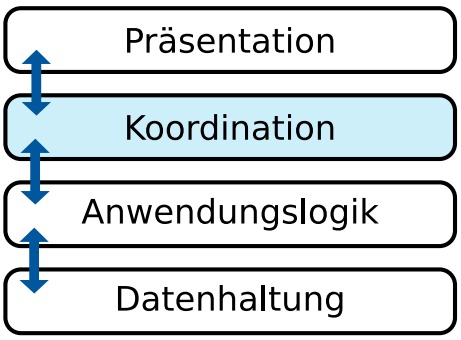


Abbildung 2.1: Vier-Schichten-Architektur nach LPC

Aufgrund von verschiedenem Kontextwissen wird sie von den an der Entwicklung beteiligten Parteien häufig unterschiedlich interpretiert. Es muss also eine gemeinsame Sprache geschaffen werden, die semantisch eindeutig ist und von allen Beteiligten verstanden werden kann. Flussgraphen eignen sich hierfür aufgrund der Intuitivität ihrer Semantik [Nag09, S. 5f]. Der Experte für Geschäftsprozesse wird stärker in den Entwicklungsprozess eingebunden als es bei klassischen Entwicklungsverfahren der Fall ist. Dennoch muss er keine Programmierkenntnisse besitzen. Die klassiche Drei-Schichten-Architektur des Softwaredesigns wird somit um eine vierte Schicht erweitert (vergleiche Abbildung 2.1). Diese Koordinationsschicht ist zwischen Präsentationsschicht und Anwendungslogik angesiedelt.

Zentrale Bedeutung bei der LPC haben die Komponenten, mit denen der Graph modelliert wird. Diese *Service Independent Building Blocks* (*SIBs*) werden vom *Anwendungsexperten* verwendet und vom *SIB-Experten* (Programmierer) entwickelt. Die Entwicklung kann hierbei mit klassischen Softwareentwicklungsmethoden erfolgen.

Ein wichtiger Aspekt der SIBs ist, dass sie wiederverwendbare Komponenten darstellen. Das heißt, dass sie gezielt für eine allgemeine Aufgabe konzipiert sind und im jeweiligen Kontext des Modells durch das Zusammenspiel vieler SIBs komplexere Funktionalität entfalten. Zusammen mit einem Konzept zur hierarchischen Modellierung können so immer komplexere Funktionen modelliert und wiederverwendet werden.

Die Entwicklung kann von zwei Seiten aus erfolgen. Die erste Variante ist, dass der Anwendungsexperte zunächst mit Platzhaltern den Geschäftsprozess modelliert. Die SIBs werden danach implementiert und ersetzen die Platzhalter im Modell. Diesem *Top-Down-Ansatz* steht der *Bottom-Up-Ansatz* entgegen, bei dem der Anwendungsexperte eine vorhandene SIB-Bibliothek zur Modellierung seiner Prozesse verwendet. Ebenso ist eine Kombination dieser beiden Ansätze möglich.

In Anlehnung an [Ste+06] werden die einzelnen Anforderungen an eine Entwicklung nach der Lightweight Process Coordination wie folgt definiert:

Agilität Da sich Modelle, Komponenten und Anforderungen mit der Zeit ändern können, soll diese Veränderung als fester Bestandteil in den Entwicklungsprozess eingebunden sein.

Anpassbarkeit	SIBs sollen der Domäne entsprechend sortiert und umbenannt werden können, um dem Anwendungsexperten eine gewohnte Umgebung zu präsentieren, ohne dass die Implementierung dies berücksichtigen muss.
Konsistenz	Durch die Verwendung desselben Modellierungsparadigmas vom Design bis zur Ausführung und Kompilierung wird eine semantische Konsistenz gewährleistet.
Verifikation	Modelle können mit bekannten Methoden wie Model Checking verifiziert werden.
Service-Orientierung	Bestehende Services und Bibliotheken sollen durch geeignete Konnektoren als Modellierungskomponenten zur Verfügung gestellt werden können.
Ausführbarkeit	Aus dem Modell soll ausführbarer Code generiert werden können.
Universalität	Die Modellierung soll unabhängig von verwendeten Technologien und Plattformen erfolgen.

Eine Erweiterung zur Lightweight Process Coordination ist das *Extreme Model Driven Development* (*XMDD*) [MS08]. Auf denselben Ideen basierend stellt es allerdings Agilität und Ausführbarkeit besonders in den Vordergrund.

Es hat sich gezeigt, dass laufende Systeme häufig angepasst werden müssen. Gründe hierfür sind veränderte Marktsituationen, Gesetze und Firmenpolitik, aber auch die Erkenntnis, dass sich das zuvor Spezifizierte nicht genau so verhält, wie es erwartet wurde. Zunehmende Globalisierung verstärkt diese Faktoren noch. Agilität muss also als zentrales Konzept in die Softwareentwicklung und den Produktlebenszyklus integriert sein.

Häufige Veränderungen bedeuten aber, dass Deployment und Integration mit möglichst wenig manuellem Aufwand durchzuführen sein müssen. Aus diesem Grund folgt die Modellierung nach XMDD dem *One-Thing-Approach* (*OTA*) [MS09]. Dort wird die Existenz eines einzelnen zentralen Modells gefordert. Ausschließlich an diesem Modell finden Veränderungen statt. Statt Systemteile unabhängig voneinander zu erzeugen, die daraufhin zusammengefügt werden müssen, wird aus diesem Modell das gesamte System generiert. Dies schließt allerdings hierarchische Modellierung nicht aus. Es wird lediglich ein globaler Einstiegspunkt definiert.

Essenziell für die Anwendbarkeit der genannten Konzepte ist die gute Unterstützung durch eine geeignete Software [MS97], welche im folgenden Abschnitt vorgestellt wird.

2.1.2 Java Application Building Center

Das *Java Application Building Center* (*jABC*) ist die aktuelle Version der Umsetzung der zuvor beschriebenen Konzepte. Es handelt sich dabei um ein Framework, welches durch weitreichende Plugin-Schnittstellen eine hohe Flexibilität ermöglicht. Dabei besteht der

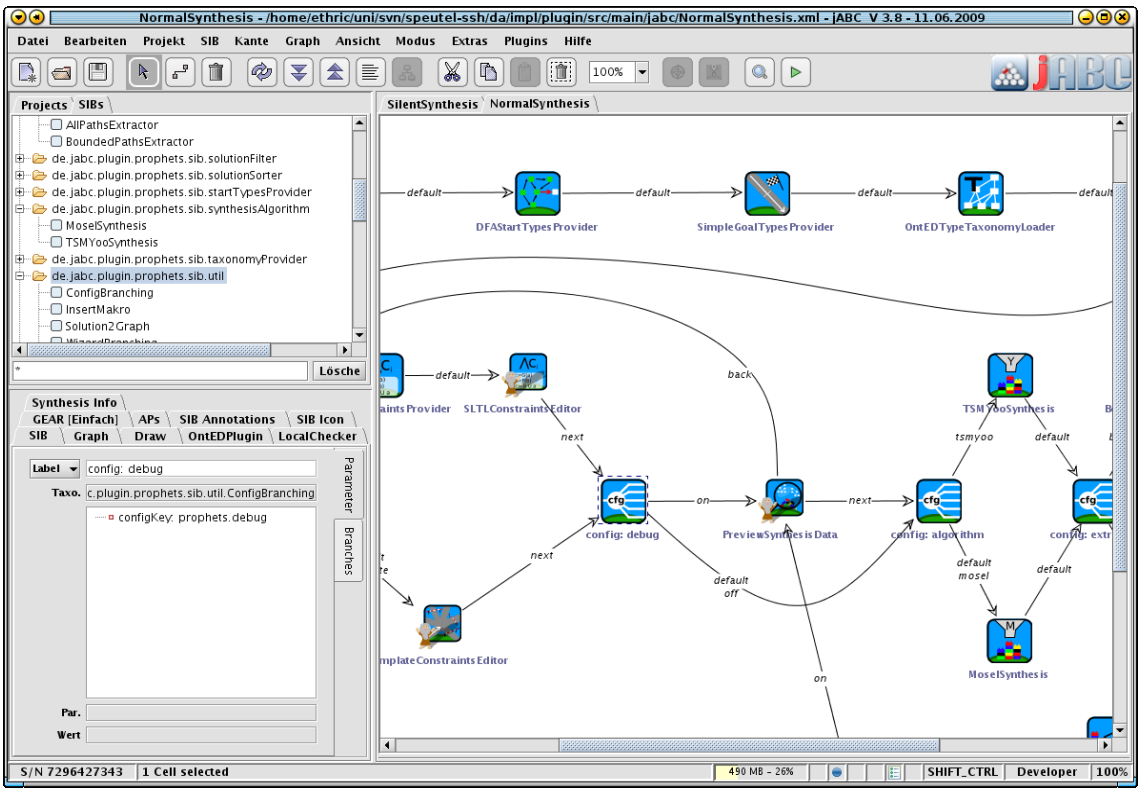


Abbildung 2.2: Benutzeroberfläche des jABC

Kern des jABC im Wesentlichen aus einer Bibliothek zur Verwaltung von Modellgraphen sowie einem Editor für diese. Die Knoten des Graphen werden von den SIBs gebildet. Die Kanten können mit einem oder mehreren Werten gelabelt sein. Wie diese Graphen semantisch interpretiert werden, hängt von den geladenen Plugins ab.

Abbildung 2.2 zeigt die Benutzeroberfläche des jABC. Neben üblichen Komponenten wie Menü-, Schnellzugriffs- und Statusleiste (oben und unten) befindet sich rechts die Zeichenfläche, die den Hauptteil der Oberfläche einnimmt. Dort ist das aktuell bearbeitete Modell dargestellt. Mehrere geöffnete Modelle werden mit Karteireitern über der Zeichenfläche angezeigt, mit denen sie auch zur Bearbeitung ausgewählt werden können. Im linken oberen Teil kann die Ansicht zwischen Projektbaum und verfügbaren SIBs umgestellt werden. Eine besondere Rolle spielen die Inspektoren unten links. Sie stellen Informationen in Abhängigkeit der gerade selektierten Elemente auf der Zeichenfläche dar.

Durch die Verwendung von Java [Sun09] als Programmiersprache wird eine weitreichende Plattformunabhängigkeit und damit die Anforderung der *Universalität* gewährleistet. Das jABC läuft auf allen Systemen, für die eine Java-VM zur Verfügung steht, was für alle verbreiteten Architekturen der Fall ist.

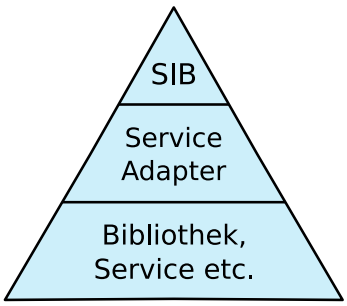


Abbildung 2.3: SIB-Entwurfsmuster

Plugin-Architektur

Die Semantik von Modellen im jABC ist nicht fest vorgegeben. Sie wird von speziellen *Semantik-Plugins* [Nag09, S. 35f] eingebracht. Hierzu stellen diese ein Plugin-Interface zur Verfügung, welches von den SIBs implementiert werden kann. Die auf diese Weise mit spezifischen Informationen angereicherten SIBs können dann vom Plugin semantisch interpretiert werden. *Feature-Plugins* hingegen stellen allgemeine Funktionalitäten zur Verfügung, die mit allen SIB-Bibliotheken verwendet werden können. Durch die sehr allgemein gehaltene Pluginschnittstelle kann das Framework hinsichtlich der zuvor definierten Anforderungen erweitert werden. So wird zum Beispiel durch den *Taxonomy Editor* [Doe08] die *Anpassbarkeit* erfüllt, durch GEAR [Bak06] die Anforderung der *Verifikation*.

Plugins bekommen die Möglichkeit, eigene Menüeinträge am Framework anzumelden. Im Hauptmenü werden diese unter „Plugins“ abgelegt. Auch die Kontextmenüs auf Graphen, SIBs, Kanten und Projekten können um pluginspezifische Einträge erweitert werden.

Darüber hinaus dürfen Plugins eigene Inspektoren anmelden und beliebig gestalten. Dadurch können sie als konsistentes Bedienelementkonzept für Plugin-GUIs genutzt werden. Weiterhin benachrichtigt das Framework alle angemeldeten Inspektoren über Änderungen der selektierten SIBs und Kanten, sodass das Plugin spezifische Informationen zu ihnen darstellen kann.

Das Framework bietet zudem die Möglichkeit, Metadaten durch sogenannte *Userobjekte* an SIBs, Modelle und Kanten anzufügen. Damit kann ein Plugin Zusatzinformationen erzeugen, die automatisch beim Speichern das Modells mitgespeichert werden und entsprechend nach dem Öffnen wieder verfügbar sind.

SIB-Entwurfsmuster

SIBs sind gewöhnliche Javaklassen. Sie können daher die Interfaces der Semantik-Plugins implementieren und ihre Funktion vollständig in der SIB-Klasse selbst ausführen. Es existiert allerdings eine Konvention zur Implementierung, bei der SIB und Funktionalität getrennt werden.

Dieses *SIB-Entwurfsmuster* (vergleiche Abbildung 2.3 auf der vorherigen Seite) sieht eine Dreiteilung vor, bei der das SIB selbst nur eine leere Hülle ist, die eine Funktion im *Service Adapter* aufruft. Dieser wiederum greift auf darunterliegende Services und Bibliotheken zurück. Dadurch wird erreicht, dass SIBs, die Funktionen von großen Bibliotheken zur Verfügung stellen, auch geladen werden können, wenn diese Bibliotheken gerade nicht vorliegen. Die SIBs können trotzdem im Modell verwendet werden. Lediglich zur Ausführung werden dann die Bibliotheksdateien benötigt. Dieses Entwurfsmuster unterstützt zudem die Anforderung der *Service-Orientierung*.

Weiterhin besitzt jedes SIB einen *Unique Identifier* (*SIB-UID*). Im jABC-Modell wird dieser zur Identifikation und Speicherung von SIBs verwendet. Dadurch können Komponenten eindeutig zugeordnet werden, auch wenn sich die implementierende Klasse ändert. Die Trennung von der Javaklasse ermöglicht darüber hinaus, dass das Modell unabhängig von Java interpretiert werden kann.

Zudem werden in einem SIB *Branches* und *Parameter* definiert. Diese können im jABC-Editor verwendet werden. Branches stellen die Ausgänge der SIBs dar und können ausgehenden Kanten zugewiesen werden. Die Parameter kann der Anwender mit Werten belegen. Für beide Aufgaben existiert hierfür der SIB-Inspektor. Darüber hinaus können dort auch Namen für Branches hinzugefügt werden, die zuvor nicht explizit im SIB deklariert waren. Diese werden *Mutable Branches* genannt.

Flussgraphsemantik

Es existiert eine Vielzahl von Semantik-Plugins für das jABC, welche zum Beispiel die Modellierung von relationalen Datenbanken [Win06] oder kontextfreien Grammatiken [Nag09, S. 57ff.] als jABC-Graphen ermöglichen. Für PROPHETS sind allerdings diejenigen Plugins von besonderem Interesse, die eine Flussgraphsemantik auf den Modellen definieren. Dies sind der Tracer [Doe06] und Genesys [JMS08].

Der Tracer ermöglicht eine direkte Ausführung des Graphen während der Modellierung. Er stellt zudem eine Oberfläche zum schrittweisen Ausführen zur Verfügung. Genesys liefert verschiedene Code-Generatoren, die eine Ausführung des Modells außerhalb des jABC ermöglichen. Dabei existieren Generatoren für unterschiedliche Plattformen (siehe [Jö09]). Generatoren, die Javacode erzeugen, welcher unabhängig von den Framework-Klassen des jABC ist, werden „Pure-Generatoren“ genannt. Tracer und Genesys erfüllen die zuvor definierte Anforderung der *Ausführbarkeit*, Genesys darüber hinaus noch die *Universalität*.

Die Art, wie der Tracer und Genesys die Semantik eines SIB-Graphen interpretieren, ist sehr ähnlich. Beide erwarten nach dem Ausführen eines SIBs die Rückgabe eines Strings. Gemäß dieses zurückgegebenen Strings wird dann über den Branch, der damit gelabelt ist, das nachfolgend auszuführende SIB gesucht. Als Mittel zum Datenaustausch zwischen den SIBs existiert ein *Ausführungskontext*. Dort können unter Angabe eines Schlüssel-Strings Daten abgelegt oder ausgelesen werden. SIBs enthalten häufig Parameter, mit denen der Anwendungsexperte diese Schlüssel konfigurieren kann. Damit kann er

beeinflussen, welche SIBs Daten miteinander austauschen.

Ausführbare Prozesse bei PROPHETS und in dieser Arbeit sind stets als solche zu verstehen, die von Tracer und Genesys gemäß der hier vorgestellen Flussgraphsemantik interpretiert werden.

2.1.3 Electronic Tool Integration

Mit dem Namen *Electronic Tool Integration* (*ETI*) wird eine seit 1997 betriebene Experimentierplattform für Services[2] bezeichnet. Im Vordergrund stand, damit eine Community von Entwicklern und Anwendern aufzubauen, sodass möglichst viele Tools einer großen Menge von Nutzern zum Ausprobieren zur Verfügung stehen konnten. Als Benutzeroberfläche diente MetaFrame, eine Vorgängerversion des jABC.

Zur Umsetzung dieser Ziele gab es zwei zentrale Ideen. Zum Einen waren die Tools online verfügbar und konnten entfernt ausgeführt werden. Ein potenzieller Anwender musste sich daher nicht um Installation und Wartung kümmern. Zudem konnte er die Tools auch dann verwenden, wenn sie für sein System nicht verfügbar waren[3]. Zum Anderen konnte durch Einbindung von Synthese die Komposition von Tools automatisch erfolgen. Damit war auch ein neuer Anwender in der Lage, zu produktiven Ergebnissen zu gelangen, auch wenn er sich in der großen Menge von Tools noch nicht vollständig zurechtfand.

Aufgrund der Verwendung veralteter Technologien wurden Teile des Systems bereits für das jABC neu implementiert. Diese als *jETI* benannte Implementierung umfasst im Wesentlichen den Aspekt der entfernten Ausführung [Nau09]. Primär relevant für diese Arbeit ist allerdings nur der Aspekt der Synthese, der im Folgenden näher vorgestellt wird.

Bei ETI besteht ein Tool aus dem Tripel `(inputType, name, outputType)`. Eine ausführbare ETI-Sequenz besteht aus einer Liste von diesen Tripeln, wobei jeweils der Eingabetyp eines Tools dem Ausgabetyp des Vorgängers entsprechen muss. Die Eingabe des ersten und die Ausgabe des letzten Tools in der Sequenz ergeben Ein- und Ausgabe der gesamten Sequenz.

Bei der Synthese konnten ein initialer Typ und ein gewünschter Zieltyp angegeben werden. Die Plattform hat daraufhin dem Anwender mögliche Sequenzen präsentiert, die diese Typen als Ein- beziehungsweise Ausgabe haben. Zusätzlich konnte durch SLTL-Formeln (siehe Abschnitt 2.2.2 auf Seite 12) die Synthese weiter angepasst werden. Hierzu wurden auch Formelmuster definiert, die einem unerfahrenen Anwender komplexere Formeln zugänglich machen sollten.

Mit der zuvor benannten Neuimplementierung jETI wurde das System so erweitert, dass Tools gewöhnliche SIBs sind, bei denen der Service-Adapter einen entfernten Aufruf an einen jETI-Toolserver ausführt. Sie lassen sich zusammen mit anderen SIB-Bibliotheken

[2] Im Folgenden wird die originale Bezeichnung „Tool“ verwendet, auch wenn „Service“ der aktuell gängigeren Bezeichnung entspricht.

[3] Zum Beispiel Unix-Tools, für die keine Windows-Version exisitierte.

verwenden und implementieren die Interfaces für Tracer und Genesys. Zudem wurde ihr Ein- und Ausgabeverhalten verallgemeinert, sodass mehrere Typen möglich sind. Eine neue Integration der Synthese muss diese veränderten Umstände berücksichtigen.

2.2 Synthese

In der Informatik wird unter Synthese die Generierung eines ausführbaren Systems aus deklarativen Beschreibungen verstanden. So lassen sich aus unzusammenhängenden Anforderungen und Komponenten imperative Strukturen schaffen. Damit steht die Synthese in engem Zusammenhang zu automatischen Planungsverfahren [RN04] oder dem Situationskalkül [MH69].

Die Synthese in PROPHETS basiert auf einer am Lehrstuhl für Programmiersysteme entwickelten Softwarebibliothek für Synthese mittels *Semantic Linear Time Logic*. Die dort verwendeten Begrifflichkeiten werden im Folgenden zunächst vorgestellt. Syntax und Semantik dieser Logik behandelt der darauffolgende Unterabschnitt. Die letztendlich für die Verwendung in PROPHETS relevanten Details zur API bilden den Abschluss dieses Abschnitts.

2.2.1 Definition

Die Wissensbasis, auf der die Synthese aufbaut, besteht aus ausführbaren Aktionen, die *Module* genannt werden. Diese Module operieren auf *Typen*. Die Menge aller Module und Typen wird *Universum* genannt. Damit ein Modul ausführbar ist, müssen bestimmte Voraussetzungen erfüllt sein, die als Teilmengen der Menge aller Typen notiert sind. Gleichermaßen werden Effekte beschrieben, die durch die Ausführung eines Moduls eintreten. Dadurch ergeben sich vier Mengen, durch die ein Modul charakterisiert wird:

use beschreibt die Typen, die vorhanden sein müssen, damit das Modul ausgeführt werden kann.

forbid beschreibt Typen, die nicht vorhanden sein dürfen, damit das Modul ausgeführt werden kann.

gen ist die Menge der Typen, die durch Ausführung des Moduls generiert wird, also der Menge der vorher verfügbaren Typen hinzugefügt wird.

kill definiert Typen, die durch die Ausführung des Moduls „zerstört“ werden. Diese Typen werden aus der vor der Ausführung verfügbaren Menge von Typen entfernt.

Damit lassen sich Module als bedingte Transformationen auf der Potenzmenge der Typen auffassen.

Sei T die Menge der Typen und M die Menge der Module. Dann transformiert das Modul $m \in M$ die zu Beginn verfügbare Menge $t \subseteq T$ wie folgt:

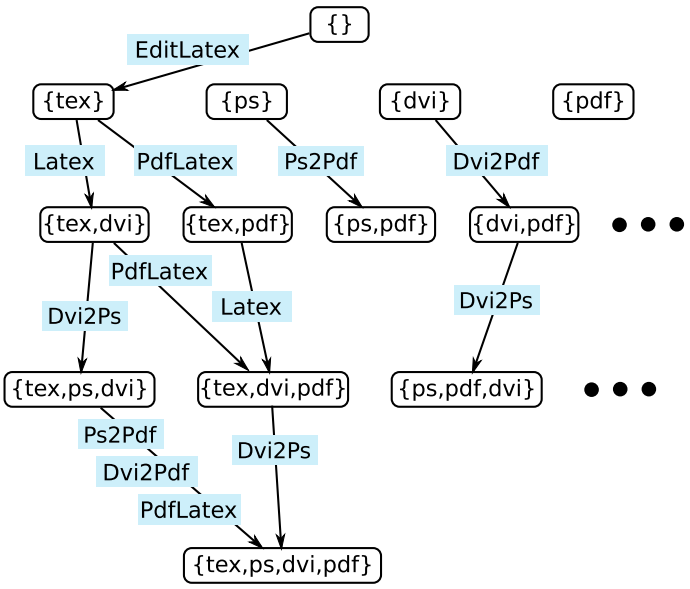


Abbildung 2.4: Synthese-Universum im LaTeX-Beispiel

$$\begin{aligned} m : \mathcal{P}(T) &\rightarrow \mathcal{P}(T) \\ t &\mapsto (t \setminus kill(m)) \cup gen(m) \end{aligned}$$

Damit das Modul angewendet werden kann, müssen folgende Voraussetzungen erfüllt sein:

$$\begin{aligned} use(m) &\subseteq t \\ forbid(m) \cap t &= \emptyset \end{aligned}$$

Ausgehend von einer Startmenge $s \subseteq T$ wird eine Sequenz $(m_1, m_2, \ldots m_k)$ von Modulen gesucht, sodass für einen Zielmenge $g \subseteq T$ gilt:

$$m_k \circ \ldots \circ m_2 \circ m_1(s) \subseteq g$$

Dabei müssen die oben benannten Voraussetzungen für jedes Modul erfüllt sein.

Durch diese Betrachtungsweise kann das Universum als Graph dargestellt werden, in dem die Knoten Teilmengen von Typen repräsentieren. Die Kanten werden mit den Modulen gelabelt, die ausführbar sind und die entsprechende Typtransformation durchführen. Abbildung 2.4 zeigt einen Ausschnitt des Universums im LaTeX-Beispiel. Da die Anzahl der Teilmengen (und damit die Anzahl der Knoten) exponentiell in Anzahl der möglichen Typen wächst, kann der Graph unter Umständen nicht explizit dargestellt werden. In solch einem Fall wird eine implizite Darstellung benutzt, die durch Regelsätze zu einem

Knoten die möglichen Nachfolger generiert. Diese Regelsätze ergeben sich aus obigen Voraussetzungen.

Bis hierher ist die Synthese, von unterschiedlichem Vokabular abgesehen, sehr ähnlich zu bekannten Planning-Konzepten. Im Folgenden wird gezeigt, wie die in PROPHETS verwendete Synthese zusätzlich parametrisiert werden kann.

2.2.2 Semantic Linear Time Logic

Die in PROPHETS verwendete Synthese basiert neben dem Universum auf einer temporallogischen Formel. Damit lässt sich die Anforderung über die Menge an Modulen hinaus spezifizieren. Temporallogiken lassen sich in die zwei Gruppen *Linear Time* und *Branching Time* unterteilen. Der Unterschied liegt auf dem Modell, auf dem sie arbeiten. Linear-Time-Logiken haben als Modell einen Pfad, Branching-Time-Logiken hingegen einen Baum [CGP99]. Dies kann zum Beispiel der „Computation Tree“ eines Programms sein.

Da mit der Logik im Rahmen der Prozesssynthese Einschränkungen über die Zulässigkeit von Modulsequenzen formuliert werden sollen, wird eine Linear-Time-Logik benötigt. Das Modell entspricht dann in dem Universumsgraphen dem Pfad vom Start- zum Zielzustand. Die von PROPHETS verwendete Synthese basiert auf der *Semantic Linear Time Logic* (*SLTL*) [SMF93], die im Folgenden in Syntax und Semantik vorgestellt wird. Auf die Beschreibung von Branching-Time-Logiken wird verzichtet, da sie im Kontext der Synthese im jABC keine Anwendung finden.

Für die Gültigkeit eines Pfades ist also die Entscheidung relevant, ob eine gegebene SLTL-Formel erfüllt ist. Ziel der Synthese ist es, Pfade zu finden, die vom Start- zum Zielzustand führen und die gegebene *Zielformel* erfüllen.

Die Syntax ist durch folgende BNF gegeben [SMF93]:

$$\Phi \quad ::= \quad true \quad | \quad t \quad | \quad \neg\Phi \quad | \quad \Phi \wedge \Phi \quad | \quad < m > \Phi \quad | \quad \mathbf{G}\ \Phi \quad | \quad \Phi\ \mathbf{U}\ \Phi$$

Das logische „OR“ ($\Phi_1 \vee \Phi_2$) kann durch $\neg(\neg\Phi_1 \wedge \neg\Phi_2)$ dargestellt werden. Als abkürzende Schreibweise wird $\mathbf{F}\ \Phi$ für das häufig benötigte Muster $true\ \mathbf{U}\ \Phi$ definiert.

Anhand von Abbildung 2.5 auf der nächsten Seite wird die Semantik von SLTL zunächst informell beschrieben. Die logischen Operatoren „AND“ und „NOT“ werden als bekannt vorausgesetzt und nicht separat erläutert. Durch die Fragezeichen in der Abbildung wird ein „don’t care“ dargestellt. Das heißt, dass die entsprechend markierten Kanten und Knoten nicht für die Auswertung der Formel relevant sind.

t Die Formel ist genau dann erfüllt, wenn die Menge von Typen, die durch den ersten Knoten im Pfad repräsentiert wird, *t* enthält.

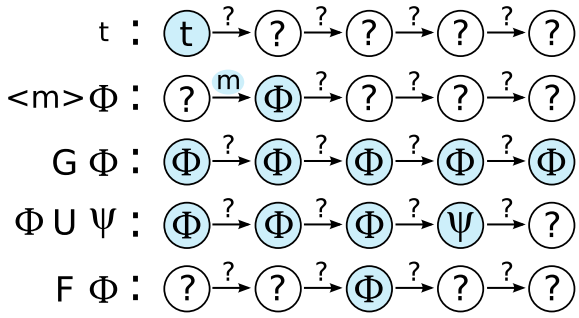


Abbildung 2.5: Informelle Semantik von SLTL

Next: **<m>**Φ — Diese Formel benötigt zur Auswertung die Betrachtung um einen Schritt in die Zukunft. Sie ist erfüllt, wenn im nächsten Knoten Φ erfüllt ist und dieser über das Modul *m* erreicht wird.

Globally: **G** Φ — Zur Erfüllung dieser Formel ist notwendig, dass die Eigenschaft Φ auf allen Knoten bis zum Ende des Pfades erfüllt ist.

Until: Φ **U** Ψ — Mit dieser Formel kann dargestellt werden, dass eine Eigenschaft (Φ) so lange erfüllt sein muss, bis eine andere (Ψ) sie ablöst. Zur Erfüllung der gesamten Formel muss Ψ auf jeden Fall irgendwann erfüllt sein[4].

Eventually: **F** Φ — Diese Formel ist erfüllt, wenn es einen Knoten auf dem Pfad gibt, in dem Φ erfüllt ist.

Der Unterschied zur sonst stärker verbreiteten *Propositional Linear Time Logic* (*PLTL*) besteht darin, dass der „Next"-Operator durch Angabe eines Moduls parametrisiert werden kann und die Auswertung von *m* und *t* anhand einer *Taxonomie* geschieht. Dies ist ein gerichteter kreisfreier Graph mit einem dedizierten Wurzelelement. Knoten ohne Nachfolger werden „konkret" genannt, Knoten mit Nachfolgern „abstrakt". In der Formel dürfen für *t* konkrete und abstrakte Typen eingesetzt werden. Gleichermaßen gilt dies für die Module *m*. Bei der Auswertung wird nun nicht ausschließlich auf die Gleichheit in Formel und Pfad geprüft, sondern gegen eine Menge von Möglichkeiten, die sich aus der Taxonomie ergeben.

Im Folgenden seien mit M_c die konkreten und mit M_a die abstrakten Module bezeichnet. Die Menge aller Module M ergibt sich aus der Vereinigung von M_a und M_c. Die Typmengen T, T_c und T_a werden analog definiert[5].

Für die formale Semantik seien Pfade als alternierende Sequenz von Typmengen und Modulen dargestellt. Ein Pfad $p = (t_0, m_1, t_1, m_2, t_2, \ldots m_k, t_k)$ mit $k \in \mathbb{N}_0$, $t_i \in \mathcal{P}(T_c)$ und $m_i \in M_c$ erfüllt eine Formel Φ (geschrieben: $p \models \Phi$) gemäß der rekursiven Definition:

[4] Das Verhalten eines „Weak Until", bei dem Ψ nicht zwingend wahr werden muss, kann durch Verwendung von „Until" und „Globally" simuliert werden: Φ **WU** Ψ = (Φ **U** Ψ) ∨ **G** Φ

[5] Die im vorangegangenen Abschnit verwendeten Bezeichnungen *M* und *T* entsprechen den hier definierten Mengen M_c und T_c, da tatsächliche Module immer konkret sind und auch nur auf konkreten Typen arbeiten können.

$$
\begin{array}{rll}
p \models \mathit{true} & & \text{gilt für jedes } p \\
p \models t & \text{gdw.} & t \in TTAX(t_0) \\
p \models \neg\Phi & \text{gdw.} & p \not\models \Phi \\
p \models \Phi_1 \wedge \Phi_2 & \text{gdw.} & p \models \Phi_1 \quad \text{und} \quad p \models \Phi_2 \\
p \models < m > \Phi & \text{gdw.} & k > 0 \quad \text{und} \quad m \in MTAX(m_1) \quad \text{und} \quad p_1 \models \Phi \\
p \models \mathbf{G}\ \Phi & \text{gdw.} & \forall i \in \{0, \ldots k\} : p_i \models \Phi \\
p \models \Phi_1\ \mathbf{U}\ \Phi_2 & \text{gdw.} & \exists i \in \{0, \ldots, k\} : \forall j \in \{0, \ldots, i-1\} : p_j \models \Phi_1 \quad \text{und} \quad p_i \models \Phi_2
\end{array}
$$

Dabei sei die abkürzende Schreibweise p_i definiert als:

$$
\begin{array}{rl}
p_i = (t_i, m_{i+1}, \ldots m_k, t_k) & \text{wenn} \quad i \in \{0, \ldots, k-1\} \\
p_i = (t_k) & \text{wenn} \quad i = k
\end{array}
$$

Die verwendeten Funktionen zur Auswertung der taxonomischen Informationen sind wie folgt definiert:

$$
\begin{array}{rcl}
MTAX : M_c & \longrightarrow & \mathcal{P}(M) \\
x & \mapsto & \{m \in M \mid m \text{ liegt in der Taxonomie auf einem} \\
 & & \text{Pfad von der Wurzel nach x}\} \\
TTAX : \mathcal{P}(T_c) & \longrightarrow & \mathcal{P}(T) \\
X & \mapsto & \{t \in T \mid \exists x \in X : \text{t liegt in der Taxonomie} \\
 & & \text{auf einem Pfad von der Wurzel nach x}\}
\end{array}
$$

2.2.3 Verwendete Bibliothek

Als Synthese-Backend verwendet PROPHETS eine aktuell am Lehrstuhl für Programmiersysteme entwickelte Bibliothek. Diese enthält momentan zwei Syntheseverfahren. Ersteres baut auf dem Konzept der ETI-Synthese auf [SMF93], erweitert dieses aber um die zuvor beschriebenen Mengen von Typen als Zustände. Das zweite Verfahren [Mar+09] verwendet zur Auswertung eine Repräsentation der SLTL-Formel in monadischer Logik 2. Ordnung, aus der mit jMosel [Top05] ein minimierter endlicher Automat erzeugt wird. Für die formalen Details zu den Algorithmen wird an dieser Stelle auf die genannten Quellen verwiesen, da sie für diese Arbeit nicht relevant sind. Im Folgenden werden die Bezeichungen „ETI-Synthese“ und „Mosel-Synthese“ für diese beiden Implementierungen verwendet.

Beide Algorithmen erzeugen unter Angabe von Modulen, Taxonomien und SLTL-Zielformel einen impliziten NFA. Alle Pfade in diesem Automaten, die vom Startzustand zu einem

akzeptierenden Zustand führen, erfüllen die Synthese. Das heißt, sie stellen eine gültige Sequenz gemäß der in Abschnitt 2.2.1 „Definition“ auf Seite 10 eingeführten Bedingungen dar und erfüllen die Zielformel. Eine explizite Angabe von Zieltypen ist nicht möglich. Allerdings können diese durch eine entsprechende SLTL-Formel kodiert werden[6].

Der Automat, den die Synthese erzeugt, wird *Transition System Model* (*TSM*) genannt. Dieses Synthese-TSM ist ein um SLTL-Informationen erweitertes Universum. Es kann mit einem beliebigen Suchalgorithmus nach Zielzuständen durchsucht werden. Dabei wird die Formel automatisch ausgewertet. Pfade vom Start zu akzeptierenden Zuständen in dem Synthese-TSM sind gültige Ergebnisse der Synthese.

2.3 Datenflussanalyse

Die Verfügbarkeit von Daten stellt ein zentrales Konzept in PROPHETS dar. Module benötigen Typen, damit sie ausgeführt werden können und generieren gleichzeitig andere Typen für die Ausführung weiterer Module. Dadurch ergibt sich eine Verwandtschaft zur Datenflussanalyse, deren Methoden in PROPHETS zur Unterstützung herangezogen werden können.

Ihren Ursprung hat die Datenflussanalyse in der Entwicklung von optimierenden Compilern. Dort wird zum Beispiel im Programmcode überprüft, ob eine Variable, der ein Wert zugewiesen wird, überhaupt gelesen wird, bevor eine weitere Zuweisung einen neuen Wert setzt. Ist dies nicht der Fall, kann die Zuweisung ausgelassen und die Zeit zur Berechnung des Wertes gespart werden.

Zur Analyse wird der Programmcode in den dazugehörigen Flussgraphen überführt. Es wird zwischen „must“- und „may“-Analyse unterschieden. Dabei liefern „must“-Analysen jene Informationen, die auf allen Pfaden zu dem untersuchten Programmpunkt gelten. „may“-Analysen hingegen sammeln Informationen, für die es einen solchen Pfad gibt. Das Problem, die tatsächlich möglichen Pfade zu bestimmen, ist unentscheidbar[7]. Daher muss das Problem auf sichere Schranken relaxiert werden. Hierbei kann vorkommen, dass ein für die „must“-Analyse herangezogener Pfad niemals ausgeführt werden kann. Dennoch wird er verwendet, um die verfügbaren Informationen einzuschränken.

Eine formale Grundlage für eine ganze Reihe optimierender Analysen bilden die monotonen Frameworks. Im Rahmen von PROPHETS wird aber nur eine dieser Analysen in einer vereinfachten Form benötigt. Diese Available-Expressions-Analyse wird im Folgenden auf die Gegebenheiten von PROPHETS übertragen, gefolgt von einem einfachen Algorithmus zur Lösung dieses Problems.

[6] zum Beispiel $\mathbf{F}\ (t_1 \wedge t_2 \wedge \ldots \wedge t_k)$

[7] siehe MOP-Solution [NNH05, S. 78]

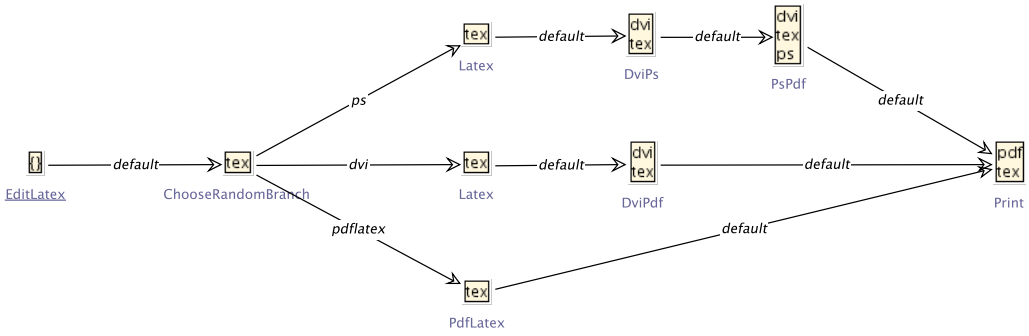


Abbildung 2.6: Datenflussanalyse auf den Typen im LaTeX-Beispiel

2.3.1 Verfügbare Typen

Die Available-Expressions-Analyse untersucht für jeden Programmpunkt, welche Ausdrücke (Expressions) auf jeden Fall vorher schon berechnet wurden („must“-Analyse). Wird an einem Programmpunkt ein Ausdruck benötigt, der vorher schon berechnet wurde, dann muss dies nicht erneut erfolgen.

Auf das jABC und das Typkonzept von PROPHETS abgebildet wird daraus die Fragestellung, welche Typen an einem SIB verfügbar sind, unabhängig davon über welchen Pfad vom Start-SIB aus dieses SIB erreicht wurde. Diese Information wird von PROPHETS an zwei Stellen verwendet. Zum Einen lässt sich damit bestimmen, welche Starttypen an den Synthesealgorithmus übergeben werden, zum Anderen kann damit aber auch verifiziert werden, ob ein SIB alle für seine Ausführung benötigten Typen verfügbar hat.

Abbildung 2.6 zeigt anhand des LaTeX-Beispiels einen jABC-Graph mit den drei alternativen Pfaden, mit denen aus **TEX** ein **PDF** erzeugt und ausgedruckt werden kann. Die SIB-Icons wurden hierbei durch die Information ersetzt, welche Typen laut Datenflussanalyse am Eingang der SIBs verfügbar sind. Man sieht, dass am `Print`-SIB ausschließlich **TEX** und **PDF** annotiert sind. Da es mindestens einen Pfad gibt, auf dem **PS** oder **DVI** nicht generiert werden (nämlich der unterste), sind diese beiden Typen dort nicht vorhanden.

2.3.2 Worklist-Algorithmus

Der Worklist-Algorithmus ist ein iteratives Verfahren zur Bestimmung von Datenflussinformationen. Die grundlegende Idee[8] wird zunächst anhand eines Ausschnittes aus einem Graph, für den die Analyse durchgeführt werden soll, vermittelt (vergleiche Abbildung 2.7 auf der nächsten Seite). Gesucht werden die verfügbaren Typen des blau markierten Knotens. Da genau die Typen bestimmt werden sollen, die immer verfügbar sind, muss der Schnitt über alle Typen gebildet werden, die über die Vorgänger ankommen können. Dabei wird für jede Kante die Menge der Knoten, die am Vorgänger verfügbar sind, mit denen vereinigt, die über die Kante neu erzeugt werden.

[8] Für die vollständigen formalen Hintergründe, siehe [NNH05].

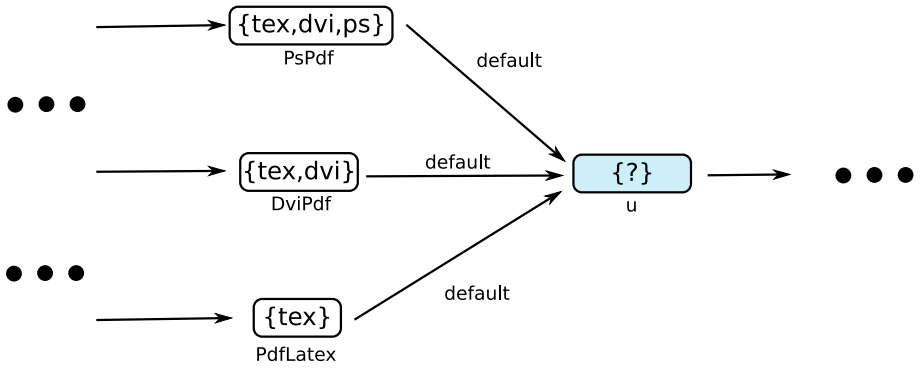


Abbildung 2.7: Lokale Betrachtung der Worklist-Idee

Unter der Annahme, dass bei den Vorgängerknoten die verfügbaren Typen schon bekannt sind, ergeben sich die Typen $T(u)$ an dem markierten Knoten u durch:

$$T(u) = \Big(\{tex, dvi, ps\} \cup \{pdf\}\Big) \cap \Big(\{tex, dvi\} \cup \{pdf\}\Big) \cap \Big(\{tex\} \cup \{pdf\}\Big) = \{tex, pdf\}$$

Dies lässt sich auch für jeden Vorgänger einzeln betrachten. Hierzu seien V die Vorgänger von u und $M^+(v,u)$ die Menge der Typen die über die Kante von v nach u generiert werden („gen“); in diesem Beispiel also immer $\{pdf\}$. Analog dazu seien $M^-(v,u)$ die zerstörten Typen („kill“).

$$T(u) \subseteq \Big((T(v) \cup M^+(v,u)) \setminus M^-(v,u)\Big)$$

Gesucht wird also eine Belegung mit möglichst vielen[9] Typen, sodass diese Eigenschaft für alle Knoten im Graph erfüllt ist. Dieses wird iterativ bestimmt. Dabei werden die Typen in der Iteration i wie folgt berechnet:

$$T_i(u) = \bigcap_{v \in V} \Big((T_{i-1}(v) \cup M^+(v,u)) \setminus M^-(v,u)\Big) \cap T_{i-1}(u)$$

Für T_0 werden der Startknoten mit der leeren Menge und alle erreichbaren Knoten mit der vollständigen Typmenge initialisiert.

Formal entspricht T der unendlichen Iteration T_∞. Es kann aber gezeigt werden, dass dies gleich T_i ist, wenn $T_i = T_{i-1}$ gilt. Es reicht also aus, durch Iteration das erste i zu suchen, sodass sich die Typen nicht mehr ändern.

Die Idee des Worklist-Algorithmus ist nun, dass in jeder Iteration nicht das gesamte T_i neu berechnet werden muss, sondern nur für die Knoten, bei denen sich seit der letzten

[9] Die leere Menge erfüllt die genannte Eigenschaft immer, enthält jedoch auch keine verwertbaren Informationen.

Iteration etwas geändert haben kann. Immer, wenn $T_i(v)$ ungleich $T_{i-1}(v)$ ist, werden alle Nachfolger zur Neuberechnung an die Worklist angehängt. In einer Iteration wird ein Knoten aus der Liste entfernt und neu berechnet. Der Algorithmus terminiert, wenn sich kein Knoten mehr in der Worklist befindet.

3 Konzept

Wie bereits in der Einleitung angedeutet, ist der Kern dieser Arbeit die Erweiterung des jABC um Funktionen, die ein *geleitetes Experimentieren* mit einer Menge von Services unterstützen sollen. Hierbei werden unvollständig spezifizierte Flussgraphen mittels temporallogischer Synthese vervollständigt. Die Umsetzung erfolgt als *PROPHETS-Plugin* für das jABC.

Die Arbeit mit dem PROPHETS-Plugin ist für drei Benutzerrollen ausgelegt. Hauptsächlich arbeitet der *Endanwender* mit dem Plugin, der die Synthese ausführt und damit die Modellierung seiner Prozesse unterstützt[1]. Damit der Endanwender arbeiten kann, muss das Projekt zuvor von dem *Domänenmodellierer* eingerichtet worden sein. Dieser bereitet domänenspezifisches Wissen auf und macht es dem Plugin bekannt. Zuletzt ist noch die Rolle des *Syntheseprozessdesigners* vorgesehen, der unter Verwendung einer speziellen SIB-Palette jene Prozesse modelliert, die die Synthese tatsächlich ausführen.

In den folgenden Abschnitten werden jeweils Ideen und Konzepte vorgestellt, die für die Umsetzung des Plugins notwendig sind. Hierbei findet auch immer eine Zuordnung zu der betreffenden Rolle statt. Ein abschließender Abschnitt fasst zuletzt die Aufgaben und Anforderungen an die jeweiligen Rollen zusammen.

Die Abschnitte selbst sind *Top Down* angeordnet. Das bedeutet, dass die Darstellung bei den Konzepten des Endanwenders beginnt, auch wenn dies zuvor geleistete Arbeit seitens des Domänenmodellierers und Syntheseprozessdesigners voraussetzt. Durch diesen Aufbau kann eine schrittweise Verfeinerung der Informationen vermittelt werden. Analog zur schrittweisen Verfeinerung eines Modells soll dies das Verständnis erleichtern.

3.1 Anwendung des Plugins

Die Arbeit des Endanwenders mit dem PROPHETS-Plugin gestaltet sich zunächst identisch zur üblichen Modellierung mit dem jABC. Zusätzlich erhält er nun die Möglichkeit, Kanten in seinem Modell als *Lose Spezifikation* zu markieren. Hat er dies getan, kann er den Synthesevorgang starten.

[1] Somit ist der Endanwender in PROPHETS eine Erweiterung des Anwendungsexperten beim XMDD.

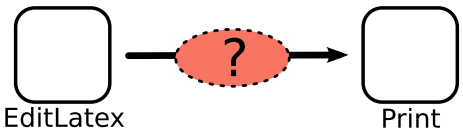


Abbildung 3.1: Ungenaue Anforderung des Modellierers

3.1.1 Lose Spezifikation

Durch die Markierung lose spezifizierter Kanten wird dem *Endanwender* die Möglichkeit gegeben, die grobe Struktur des Modells anzugeben, ohne dabei bis ins ausführbare Detail zu gehen. Die Semantik einer so spezifizierten Kante entspricht dem intuitiven Konzept „Was dazwischen passieren soll, ist nicht vorgegeben.“ (vergleiche Abbildung 3.1). Der Modellierer fordert lediglich, dass er bei `EditLatex` beginnen und mit `Print` beenden möchte. Er kann an dieser Stelle erwarten, dass zur Ausführung notwendige Zwischenschritte vom PROPHETS-Plugin automatisch eingesetzt werden.

3.1.2 Synthesedurchführung

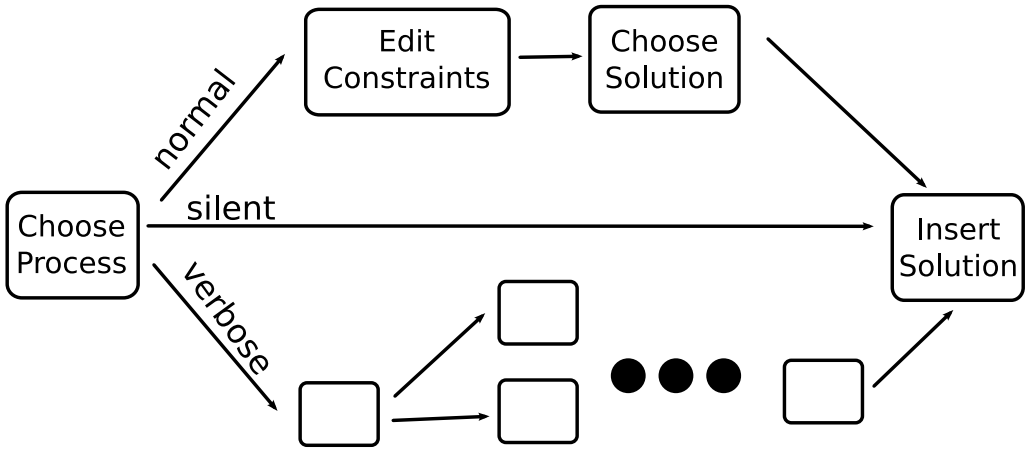


Abbildung 3.2: Workflow einer Synthese aus Sicht des Endanwenders

Eine einzelne lose spezifizierte Kante ist als Eingabe für die Synthese konzipiert. Hat der Endanwender mehr als eine dieser Kanten spezifiziert, führt PROPHETS für jede separat den *Syntheseprozess* durch. Der Workflow dieses Syntheseprozesses aus Sicht des Endanwenders ist in Abbildung 3.2 dargestellt. Zunächst wählt er einen von drei Prozessen aus, die sich im Grad der Interaktivität unterscheiden. Der Prozess „silent“ verlangt keine weiteren Eingaben vom Endanwender und fügt die Lösung direkt ein. Sollte es mehr als eine Lösung geben, wird die zuerst gefundene kürzeste automatisch ausgewählt. Bei dem Prozess „normal“ kann der Endanwender weitere Einschränkungen an die Synthese formulieren und bekommt dann eine Liste der Lösungsmöglichkeiten gezeigt, aus denen er eine auswählen kann. Der „verbose“-Prozess ist für erfahrene Anwender gedacht. Dort

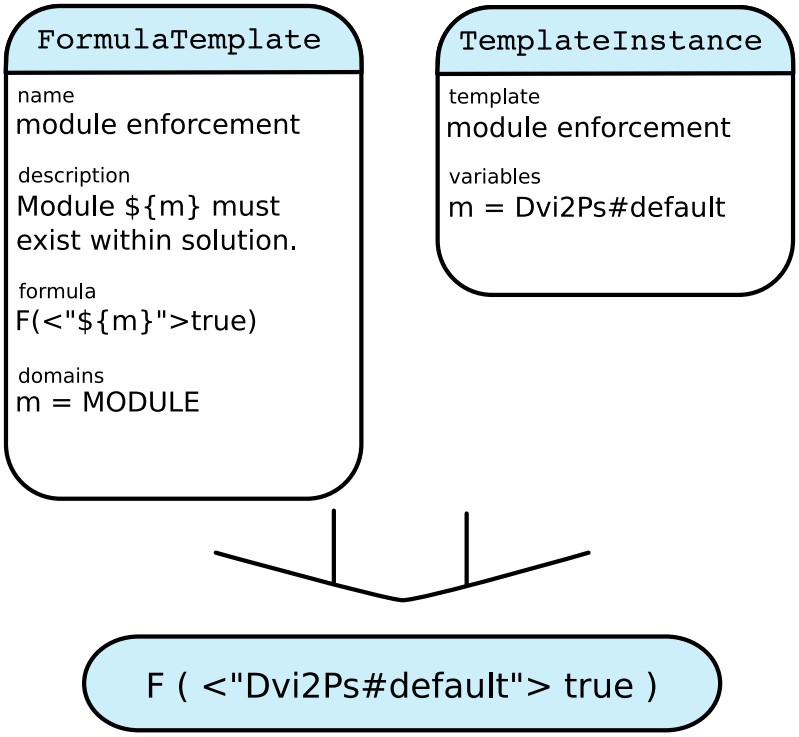


Abbildung 3.3: Template und Instanz liefern Constraint-Formel

werden eine Reihe von Fragen gestellt, mit denen das Verhalten der Synthese gezielt beeinflusst werden kann.

Zusätzlich kann über die Konfiguration des PROPHETS-Plugins ein Standardprozess festgelegt werden, sodass die initiale Abfrage nach dem Prozess übersprungen wird. Wählt der Endanwender dort den Prozess „silent“ aus, wird die Synthese vollständig ohne Interaktion durchgeführt.

3.1.3 Formeln aus Vorlagen

Der *Endanwender* soll keine Kenntnisse in temporaler Logik und der Funktionsweise der damit durchgeführten Synthese benötigen. Dennoch soll er die Synthese über die Definition von Start- und Ziel-SIB hinausgehend beinflussen können. Um dies zu ermöglichen, wurde für PROPHETS ein vereinfachtes Konzept zur Spezifikation der Synthesezielformel entwickelt. Dieses basiert auf Satzvorlagen in natürlicher Sprache.

Die Spezifikation der *Zielformel* wird zunächst in konjunktive Komponenten aufgeteilt. Jede dieser als *Constraint* benannten Komponenten wird vom Endanwender einzeln spezifiziert. Alle spezifizierten Constraints werden nachträglich vom Plugin mit dem logischen „AND“ verknüpft und bilden die Zielformel. Der Endanwender legt also Eigenschaften fest, die gemeinsam erfüllt sein müssen.

Um nun ein solches Constraint ohne SLTL-Kenntnisse zu erstellen, kann der Endanwender aus einer Menge von Satzvorlagen (*Templates*) diejenigen auswählen, die er verwenden möchte. Dabei enthält jedes Template einen natürlichsprachlichen Satz, der Lücken (Variablen) enthält, welche vom Endanwender mit Daten zu füllen sind. Ein Template mit definierten Werten für die Variablen wird als *Template-Instanz* bezeichnet. Die Werte, die der Anwender in diese Lücken einträgt, werden dann in eine zum Template gehörende SLTL-Formel übertragen (vergleiche Abbildung 3.3 auf der vorherigen Seite), ohne dass die resultierende Formel dargestellt wird. Die gültigen Wertebereiche für die Lücken ergeben sich aus der Domänenmodellierung, die im folgenden Abschnitt näher erläutert wird. Es sei allerdings vorweggenommen, dass sich damit folgende Wertemengen für Lücken ergeben: „Module“, „Abstrakte Module“, „Konkrete Module“, „Typen“, „Abstrakte Typen“, „Konkrete Typen“ und „Freitext“.

Zusätzlich zu den vom Endanwender spezifzierten Constraints wird noch ein *Goal Constraint* automatisch vom Plugin hinzugefügt. Dieses wird über die Eingangstypen des Ziel-SIBs der lose spezifizierten Kante gebildet. Es enthält die Information, dass mindestens die für die Ausführung des Ziel-SIBs notwendigen Typen vorhanden sein müssen.

3.2 Modellierung der Domäne

Damit das PROPHETS-Plugin lose spezifizierte Kanten mittels Synthese konkretisieren kann, benötigt es Wissen über Möglichkeiten und Einschränkungen der Modellierung. Dieses je nach Anwendungsfall unterschiedliche *Domänenwissen* aufzubereiten und zu spezifizieren ist Aufgabe des *Domänenmodellierers*. Es umfasst die verwendbaren SIBs sowie taxonomische Klassifizierungen über SIBs und Typen. Darüber hinaus können noch allgemeingültige Constraints, die für jede Synthese gelten müssen, formuliert werden.

3.2.1 Moduldefinitionen

Die beiden in PROPHETS integrierten Synthesealgorithmen arbeiten neben der in SLTL formulierten Zielformel auf einem *Universum*, das aus einer Menge von *Modulen* besteht. Das Universum aus Sicht des Endanwenders besteht allerdings aus jenen SIBs, die er für die Modellierung zur Verfügung hat. Bei der Einbindung der Synthesealgorithmen in PROPHETS musste daher eine Abbildung von SIBs auf eine passende Modulbeschreibung entwickelt werden.

Der grundlegende Unterschied zwischen SIBs und den für die Synthese benötigten Modulen liegt darin, dass Module in ihrem Ein- und Ausgabeverhalten klar definiert und linear sind. SIBs hingegen können mit verschiedenen Ergebnissen enden. Dies wird durch unterschiedliche Branches im jABC dargestellt. Dabei kann sich das Ein- und Ausgabeverhalten eines SIBs zwischen seinen Branches unterscheiden. Aus diesem Grund wird in PROPHETS für jeden möglichen Branch eines SIBs ein eigenes Modul in die Synthese gegeben.

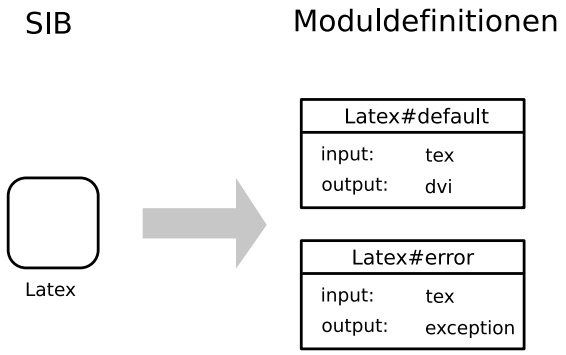


Abbildung 3.4: Moduldefintionen aus einem SIB erstellen

Diese Moduldefinitionen zu erstellen ist Aufgabe des *Domänenmodellierers*. Abbildung 3.4 verdeutlicht diesen Prozess. Sie zeigt, dass eine Moduldefintion aus einem SIB, dem Branch-Namen sowie der Angabe von Ein- und Ausgabetypen besteht. Das SIB `Latex` erzeugt im Normalfall (also auf dem Branch „default“) aus einer tex-Datei eine dvi-Datei. Im Fehlerfall hingegen wird statt einer dvi-Datei der Fehler in Form einer **EXCEPTION** erzeugt.

Bei der Definition von Modulen werden *symbolische Typnamen* für die Ein- und Ausgabe verwendet. Diese sind in ihrer Definition zunächst unabhängig von konkret implementierten Javaklassen. Die Kompatibilität der letztendlich von den SIBs verwendeten Javaklassen muss vom Domänenmodellierer beachtet werden. Inkompatible Klassen können von der Synthese nicht entdeckt werden und führen bei der Ausführung zu Laufzeitfehlern.

Zur Vereinfachung werden in PROPHETS die zu den Synthesemodulen gehörenden Mengen „kill“ und „forbid“ (vergleiche Abschnitt 2.2.1 auf Seite 10) als leer angenommen. Das heißt, dass ein Modul ausschließlich durch Eingabe („use“-Menge) und Ausgabe („gen“-Menge) charakterisiert ist. Damit soll die Verwendung für Endanwender und Domänenmodellierer vereinfacht und das Verständnis erleichtert werden. Sollten Informationen dieser Art trotzdem benötigt werden, können sie durch eine entsprechende SLTL-Formel ausgedrückt werden.

3.2.2 Taxonomien

Die Synthese verwendet Klassifizierungsinformationen auf Modulen und Typen in Form von *Taxonomien*. Eine Taxonomie besteht aus Elementen, die in einer hierarchischen Vererbungsrelation zueinander stehen. Dabei können konkrete und abstrakte Elemente unterschieden werden. Konkrete Typen und Module sind diejenigen, die in den Moduldefinitionen vorkommen. Ihre abstrakten Pendants hingegen werden zur Gruppierung verwendet. Die konkreten Elemente erben die Eigenschaften der Eltern. Mehrfachvererebung ist möglich, sodass sich eine Taxononmie durch einen *gerichteten azyklischen Graph* (*DAG*) mit genau einem Wurzelelement darstellen lässt. Formal handelt es sich um eine

partielle Ordnung auf den Elementen mit einem dedizierten Element als Supremum.

Da kein verbreiteter Standard als Datenformat für Taxonomien existiert, werden häufig eigene Lösungen zur Darstellung und Speicherung verwendet. Dies beinhaltet die Notwendigkeit eines speziell für die eigene Sprache entwickelten Parsers. Eine Alternative stellt die Verwendung von Ontologien [Hes02] dar. Im Gegensatz zum rein auf Hierarchien begrenzten Wissen von Taxonomien werden sie in der Regel verwendet, um allgemeineres Wissen zu formulieren. So sind unter anderem beliebige Relationen auf den Elementen möglich. Weiterhin existiert eine Trennung der Elemente in Individuen und Klassen. Eine der standardmäßig definierten Relationen ist aber auch hier die Vererbungsrelation, sodass sich eine Taxonomie mittels einer Ontologie beschreiben lässt. Der Vorteil liegt nun darin, dass sich mit *OWL* [W3C04] ein Standard für die Speicherung von Ontologien etabliert hat, für den es eine Vielzahl von grafischen Editoren und APIs in nahezu allen gebräuchlichen Programmiersprachen gibt. Aus diesem Grund werden die Typ- und Modultaxonomien in PROPHETS im OWL-Format gespeichert.

Weiterhin wurde im Rahmen einer vorherigen Diplomarbeit am Lehrstuhl für Programmiersysteme der Ontologie-Editor *OntED* [May09] entwickelt, der in das jABC integriert ist. Dieser besteht aus zwei Komponenten, der OntED-API und dem OntED-Plugin. Die OntED-API wird von PROPHETS verwendet, um die im Projektverzeichnis befindlichen OWL-Dateien auszuwerten. Die Wahl des Editors für diese Dateien steht dem Domänenmodellierer grundsätzlich frei. Allerdings liefert PROPHETS eine auf dem OntED-Plugin basierende grafische Integration in das jABC. Diese wird in Abschnitt 4.4 „Erstellen der Taxonomien“ auf Seite 37 genauer vorgestellt.

3.2.3 Globale Formeln

Zusätzlich zu Modulen und Taxonomien kann der Domänenmodellierer Wissen in Form von *globalen Constraints* beschreiben. Diese Constraints, die selbst als Instanzen von Templates definiert werden, werden automatisch bei jeder Synthese einbezogen. Ein solches Constraint für die LaTeX-Domäne wäre zum Beispiel: „Wenn `PdfLatex` verwendet wird, dann sind weder PS noch DVI sinnvoll.“ Die dazugehörige Formel

```
F(<PdfLatex>true) => G!(ps OR dvi)
```

lässt sich allerdings schlecht mit einem einzelnen sinnvollen Template bilden. Teilt man sie hingegen in

```
(F(<PdfLatex>true) => G !ps) & (F(<PdfLatex>true) => G !dvi)
```

auf, so lässt es sich mit zweifacher Anwendung des Templates „Modul A schließt Typ B aus“ formulieren.

3.3 Modellierung der Synthese

Den Kern der Implementierung des PROPHETS-Plugins bildet der *Syntheseprozess*. Um diesen verständlich, übersichtlich und leicht anpassbar zu halten, wird er selbst mit dem jABC modelliert. Diese Modellierung ist Aufgabe des *Syntheseprozessdesigners*, der damit die dritte und letzte Rolle für die Arbeit mit PROPHETS bildet. Er verwendet hierfür eine eigens entwickelte SIB-Bibliothek (siehe Abschnitt 5.2 ab Seite 43). Für diese Bibliothek wurde untersucht, welche *Granularität* der Modellierung des Syntheseprozesses am sinnvollsten ist. Je feiner diese ausfällt, desto eher können vorhandene SIB-Bibliotheken wie die *Common-SIBs* [Leh09] hierfür verwendet werden. Dadurch erhöht sich aber auch der Aufwand beim Erstellen und Verändern des Modells. Aus diesem Grund fiel die Entscheidung auf eine recht grobgranulare Modellierung des Syntheseprozesses, dessen Agilität im Wesentlichen in der Austauschbarkeit einzelner SIBs zu sehen ist. Beim Kompilieren des Plugins wird der Syntheseprozess automatisch mit *Genesys* „pure-generiert“ und mit den entsprechenden GUI-Aktionen verknüpft. Änderungen und Erweiterungen an dem Prozess können daher mit sehr geringem Aufwand mit dem jABC durchgeführt und in das Plugin eingebunden werden.

Bei der Modellierung ist der Syntheseprozessdesigner an gewisse Abhängigkeiten der verwendeten SIBs untereinander gebunden. So muss zum Beispiel ein Synthesealgorithmus ausgeführt werden, bevor eine Lösungsauswahl stattfinden kann. Aus Abhängigkeiten dieser Art ergibt sich ein *verallgemeinerter Syntheseprozess*, den der Syntheseprozessdesigner bei der Modellierung beachten muss. Dieser wird im Folgenden vorgestellt.

Abbildung 3.5 auf der nächsten Seite zeigt die Abhängigkeiten des verallgemeinerten Syntheseprozesses. Es ist zu sehen, dass sie im Wesentlichen durch Ein- und Ausgabetypen bestimmt sind. Betrachtet man den Abhängigkeitsgraphen als *partielle Ordnung* auf den Komponenten, so ist jede *topologische Sortierung* darauf ein gültiger verallgemeinerter Syntheseprozess. Dieser Prozess wird als „verallgemeinert“ bezeichnet, da sich die SIBs der PROPHETS-SIB-Bibliothek den hier vorgestellten Klassen zuordnen lassen.

Eine weitere Designentscheidung zur Modellierung von Syntheseprozessen mit dem jABC ist ein Konzept zum einheitlichen Datenaustausch. Grundsätzlich können die SIBs beliebige Objekte in den *Ausführungskontext* legen und aus diesem lesen. Die meisten SIB-Bibliotheken sind so konzipiert, dass die Schlüssel, unter denen Objekte im Kontext abgelegt werden, mittels SIB-Parametern festgelegt werden müssen. Für eine allgemein zu verwendene SIB-Bibliothek ist dieser Ansatz durchaus sinnvoll. Im Kontext der PROPHETS-SIB-Bibliothek hingegen würde es unnötigen Mehraufwand seitens des Syntheseprozessdesigners verlangen. Daher wurde hier der Ansatz gewählt, auf sämtliche Daten streng getypt über ein spezielles Java-Bean zuzugreifen. Dieses `ProphetsDataBean` wird unter einem fest definierten Schlüssel im Ausführungskontext gespeichert. Somit muss sich der Syntheseprozessdesigner nicht um Kontextschlüssel kümmern. Die PROPHETS-SIBs lesen und schreiben ihre Daten automatisch richtig.

Im Folgenden werden die für die PROPHETS-SIB-Bibliothek vorgesehenen Klassen von SIBs im Einzelnen vorgestellt. Dabei wird ihr Verhalten auf Interface-Ebene dargestellt.

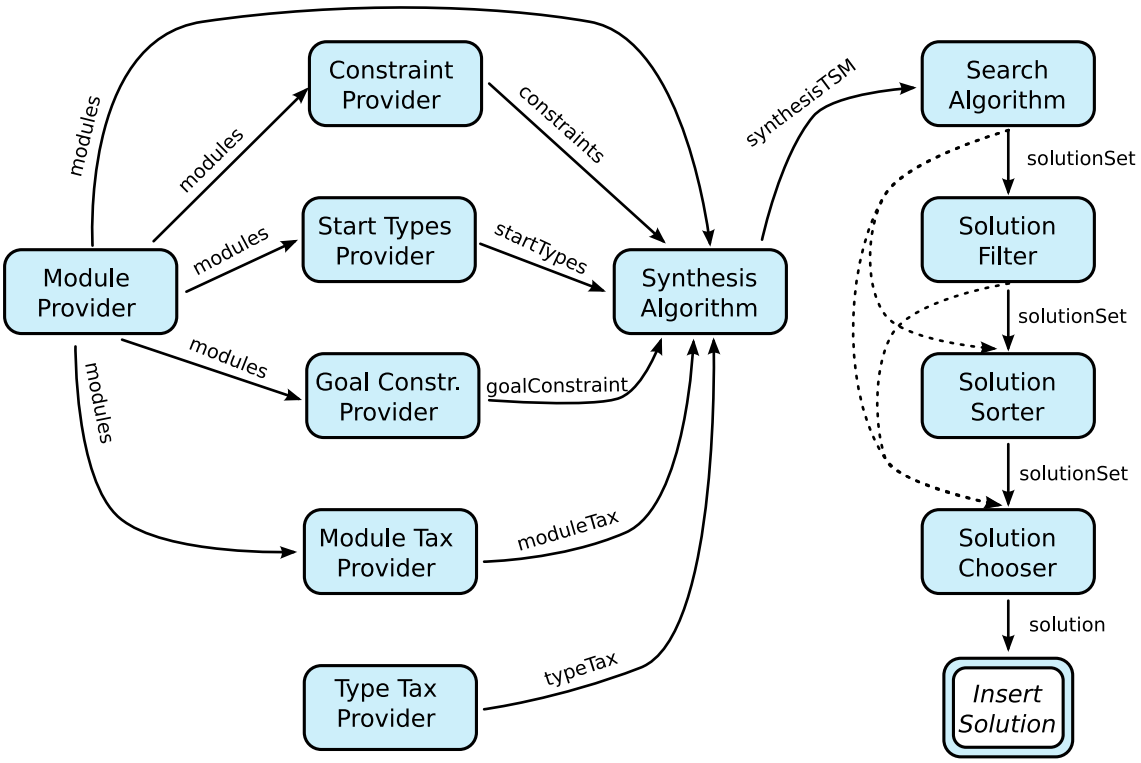


Abbildung 3.5: Abhängigkeiten für den verallgemeinerten Syntheseprozess

Einige konkrete Implementierungen werden in Abschnitt 5.2 „SIB-Bibliothek“ ab Seite 43 vorgestellt.

Module Provider

Aufgabe des Modulproviders ist es, die Modulinformationen, wie sie der Domänenmodellierer spezifiziert hat, zur Verfügung zu stellen. Er muss eine Menge von Modulen liefern, die mit dem symbolischen Namen **MODULES** bezeichnet ist. Da die Modulinformationen von nahezu allen weiteren SIBs benötigt wird, bildet er in der Regel den Anfang des Syntheseprozesses.

Constraint Provider

SIBs dieser Klasse liefern immer eine Menge von SLTL-Formeln. Diese können zum Beispiel über eine Eingabe des Benutzers erfolgen oder aus einer Datei gelesen werden. Bei Benutzereingaben kann das zuvor beschriebene Konzept der Templates zum Einsatz kommen.

Start Types Provider

Die Synthese benötigt als Startvoraussetzung eine Menge von Typen, die von Anfang an verfügbar sind. Die Start-Types-Provider liefern diese Typen anhand des Quell-SIBs

der lose spezifizierten Kante. Dabei sind unterschiedliche Implementierungen denkbar. So können die initialen Typen zum Beispiel ausschließlich über das Quell-SIB bestimmt werden (lokale Betrachtung). Alternativ wird mittels *Datenflussanalyse* von den Start-SIBs aus bestimmt, welche Typen am Quell-SIB verfügbar sind (globale Betrachtung).

Goal Constraint Provider

Die Definition des Syntheseziels wird als ein zusätzliches Constraint in die Synthese gegeben. Im einfachsten Fall ist dies ein „Eventually“ mit allen Typen, die das Ziel-SIB der lose spezifizierten Kante als Eingabe definiert. Es können aber auch komplexere Konstruktionen zum Einsatz kommen.

Type/Module Taxonomy Provider

Neben Universum und Zielformel verwendet die Synthese noch Strukturwissen über Typen und Module in Form von Taxonomien. Diese Taxonomien zu laden oder zu generieren ist Aufgabe der Taxonomieprovider.

Synthesis Algorithm

Durch die Synthesealgorithmen wird bestimmt, wie die Synthese letztendlich durchgeführt wird. Sie benötigen als Eingabe die zuvor bereitgestellten Daten (Module, Taxonomien, Constraints, Starttypen, Goal-Constraint) und produzieren als Ausgabe ein *Synthese-TSM*. Abstrahiert handelt es sich dabei um einen Graphen, der an den Knoten mit einer Menge von Typen und an den Kanten mit Modulen gelabelt ist. Alle Pfade von einem Startknoten zu einem akzeptierenden Knoten erfüllen die in die Synthese gegebene Zielformel. Dabei ist ein Pfad eine Sequenz von Modulen.

Search Algorithm

Die Aufgabe, Lösungen in dem durch die Synthese gelieferten TSM zu suchen, fällt den SIBs dieser Klasse zu. Dabei ist nicht festgelegt, ob sie den Graph vollständig durchsuchen, alle Lösungen ausgeben oder nach sonstigen Kriterien optimieren.

Solution Filter

Diese SIBs dienen dazu, die Lösungen, wie sie vom Suchalgorithmus gefunden wurden, nach bestimmten Kriterien weiter zu filtern. Wie und was gefiltert wird, hängt vom implementierenden SIB ab. Es wird lediglich gefordert, dass die ausgegebenen Lösungen eine Teilmenge der Eingabe sind. Die Verwendung einer oder mehrerer dieser Filter ist optional.

Solution Sorter

SIBs dieser Klasse bringen die Menge von Lösungen in eine definierte Ordnung. Denkbar ist hier eine Sortierung nach Lösungslänge, Kosten oder anderen Faktoren.

Solution Chooser

Der letzte Schritt im modellierten Syntheseprozess ist in der Regel das Auswählen einer Lösung aus der zuvor bestimmten (und gegebenenfalls gefilterten/sortierten) Menge von Lösungen. Dies kann automatisch oder durch Interaktion mit dem Endanwender geschehen.

Insert Solution

Mit dieser Bezeichnung wird der statische Teilprozess benannt, der nach dem Auswählen einer Lösung diese in dem ursprünglichen Graphen einsetzt. In der Regel sollte hier keine Anpassung seitens des Syntheseprozessdesigners geschehen müssen.

3.4 Zusammenfassung der Rollen

Wie in den vorangegangenen Abschnitten herausgestellt wurde, gibt es viele unterschiedliche Bereiche, die ein Anwender von PROPHETS abdecken muss. Aus diesem Grund existiert die zuvor bereits eingeführte Unterteilung in Rollen. Im Folgenden wird noch einmal zusammengefasst, was die Aufgaben der jeweiligen Rollen sind, und welches Wissen benötigt wird, um eine Rolle erfolgreich zu erfüllen.

3.4.1 Endanwender

Der Endanwender bei PROPHETS stellt eine Erweiterung des Anwendungsexperten des *Extreme Model Driven Development* dar. Daher benötigt er auf jeden Fall Kenntnisse der Basiskonzepte des jABC und dessen Bedienung. Darüber hinaus muss er die Unterschiede und Gemeinsamkeiten von SIBs und Modulen kennen. Gleichzeitig ist es von Vorteil, wenn er Typ- und Modultaxonomien lesen kann, da diese durch ihre Klassifizierungen zur Übersicht beitragen können. Zur Synthese muss er nur wissen, dass diese mit Constraints eingeschränkt werden und nur lineare Sequenzen erzeugen kann. Kompliziertere Programmlogik wie Schleifen und Verzweigungen müssen daher von ihm modelliert werden.

3.4.2 Domänenmodellierer

Der Domänenmodellierer benötigt am meisten Einarbeitung und Wissen, um seiner Arbeit mit PROPHETS nachgehen zu können. Er muss sich in der zu modellierenden Domäne auskennen und eine adäquate Abbildung dieses Wissens in die PROPHETS-Modellierung finden. Hierzu muss er Moduldefinitionen zu vorhandenen Services schreiben, die auf geeigneten symbolischen Typen operieren. Dieses Wissen sollte zudem mittels Taxonomien weiter strukturiert werden. Zusätzliches Wissen muss in Form von allgemeingültigen Constraints formuliert werden. Sollte die Domäne darüber hinaus noch zusätzliche Formel-Templates benötigen, so muss der Domänenmodellierer auch in der Lage sein, diese

Templates in SLTL auszudrücken.

Diese gesamte Arbeit des Domänenmodellierers geschieht jedoch im Idealfall nur einmal bei der initialen Konfiguration des Projektes. Nachträgliche Änderungen können vorgenommen werden, lassen sich aber leicht in die bereits bestehenden Informationen einbringen. Daher ist der erhöhte anfängliche Aufwand für den Domänenmodellierer damit kompensiert, dass er später weniger benötigt wird.

3.4.3 Syntheseprozessdesigner

Neben allgemeinem Wissen zur Modellierung mit dem jABC sind für den Syntheseprozessdesigner im Wesentlichen zwei Aspekte wichtig. Zum Einen muss er mit den SIBs der PROPHETS-SIB-Bibliothek vertraut sein. Noch wichtiger ist aber zum Anderen, dass er den verallgemeinerten Syntheseprozess kennt und bei seiner Modellierung beachtet, da er hauptsächlich Variationen von diesem erstellen muss. Er muss sich allerdings nicht explizit um Datenhaltung kümmern. Die PROPHETS-SIBs arbeiten alle automatisch auf dem `ProphetsDataBean`.

4 Umsetzung auf Benutzerebene

Im vorangegangenen Kapitel wurden die allgemeinen Konzepte erläutert, die der Implementierung des PROPHETS-Plugins zugrunde liegen. Darauf aufbauend werden in den folgenden Abschnitten einige Details zu deren Umsetzung aus Sicht der Benutzer näher erläutert. Als Benutzer werden hier sowohl Endanwender als auch Domänenmodellierer verstanden. Die davon losgelöste Arbeit des Syntheseprozessdesigners wird in Kapitel 5 „Syntheseprozessmodellierung“ ab Seite 41 separat behandelt.

Die Darstellung beginnt zunächst erneut mit dem Arbeitsbereich des Endanwenders, indem die Umsetzung der losen Spezifikation mit dem jABC vorgestellt wird. Die Implementierung des Template-Konzeptes sowie dessen Anwendung für Endanwender und Domänenmodellierer werden daraufhin betrachtet. Danach werden die Möglichkeiten zur Moduldefinition diskutiert und die durchgeführte Implementierung begründet, gefolgt von der Erstellung der Taxonomien mit dem OntED-Plugin. Den Abschluss bildet ein Abschnitt zur Modellverifikation unter Verwendung der Ein- und Ausgabeinformationen der Synthese.

4.1 Markierte Kanten

Die Modellierung von Prozessen mit dem jABC wird durch das PROPHETS-Plugin um die Möglichkeit zur *Losen Spezifikation* erweitert. Dazu kann der Endanwender eine Kante über das Kontextmenü markieren. Diese daraufhin rot dargestellte Kante erlaubt ihm auszudrücken, dass die Modellierung zwischen deren Quell- und Ziel-SIB nicht genau vorgegeben ist.

Ein jABC-Modell mit einer oder mehreren dieser markierten Kanten bildet die Grundlage der Eingabe für die Synthese mit PROPHETS. Dabei wird für jede dieser Kanten separat der *Syntheseprozess* ausgeführt. Die Reihenfolge dieser Ausführungen wird mit einer *Breitensuche*, die vom Start-SIB ausgeht, bestimmt. Dadurch wird sichergestellt, dass es keinen Pfad vom Start-SIB zur gerade synthetisierten lose spezifizierten Kante gibt, der noch nicht konkretisierte Kanten enthält[1].

Bei erfolgreicher Synthese wird die markierte Kante durch die Lösung ersetzt. Hierbei treten in Abhängigkeit der Länge der Lösung drei unterschiedliche Fälle ein (vergleiche Abbildung 4.1 auf der nächsten Seite):

[1] Dies ist nur dann wichtig, wenn die Starttypen mittels Datenflussanalyse bestimmt werden.

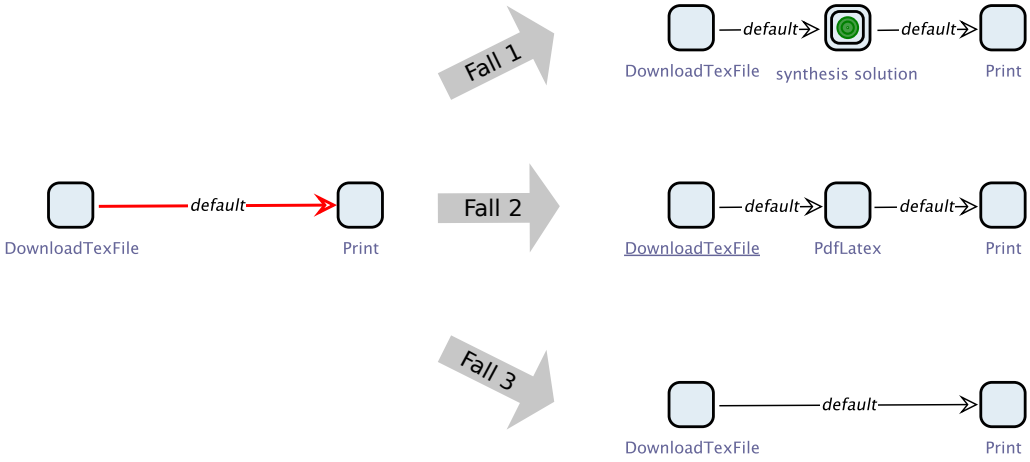


Abbildung 4.1: Einfügen des Syntheseergebnisses

Fall 1 Wenn die Lösung aus mindestens zwei SIBs besteht, wird sie als eigenständiges Modell in eine Datei gespeichert. An die Stelle der zuvor markierten Kante tritt ein `MakroSIB`, dem dieses Modell zugeordnet wird. Dies sorgt dafür, dass der Endanwender auch bei sehr langen Lösungen nicht die Übersicht in dem Modell verliert, da sein Layout grundsätzlich erhalten bleibt.

Fall 2 Besteht die Lösung aus nur einem SIB, so wird es direkt anstelle der markierten Kante in das Modell eingefügt, da der Platzverbrauch ebenso groß wie bei der Verwendung eines `MakroSIB`s ist.

Fall 3 War die vom Endanwender lose spezifizierte Kante bereits vollständig, die Länge der Lösung der Synthese also null, so wird einfach die Markierung der Kante entfernt.

4.2 Implementierung der Moduldefinition

Das zuvor für den verallgemeinerten Syntheseprozess eingeführte Konzept des *Module Provider*s gibt vor, welche Informationen über Module zur Verfügung gestellt werden müssen. Die Quelle dieser Informationen kann je nach Implementierung des SIBs unterschiedlich sein. Aus diesem Grund wurden zunächst verschiedene Möglichkeiten der Implementierung untersucht, die im Folgenden vorgestellt werden.

4.2.1 Analyse der Optionen

Für die Implementierung des Modulproviders standen im Wesentlichen drei Möglichkeiten zur Auswahl:

1. Die Syntheseinformationen werden gemeinsam mit den SIBs ausgeliefert.

2. Die Syntheseinformationen werden über einen Onlinedienst heruntergeladen.
3. Eine im Projekt abzulegende Datei enthält die Syntheseinformationen.

Bei der ersten Möglichkeit werden die Syntheseinformationen mittels Annotation oder dediziertem Interface direkt an die SIB-Klasse angehängt. Der Vorteil dieses Ansatzes besteht hauptsächlich in der verringerten Arbeit für den Domänenmodellierer und der leichteren Verteilbarkeit von synthetisierbaren SIB-Bibliotheken „als Gesamtes“. Der große Nachteil, der zum Ausschluss dieser Möglichkeit führt, liegt darin, dass das Hinzufügen oder Bearbeiten dieser Informationen immer ein erneutes Kompilieren und Verteilen der gesamten SIB-Biliothek notwendig gemacht hätte. Große Bibliotheken wie die *Common-SIBs* hätten entweder komplett überarbeitet und neu verteilt werden müssen, oder wären nicht für die Synthese verfügbar gewesen. Als Alternative zu Annotation und Interface hätten diese Informationen auch als Metadaten in dem SIB-Bibliotheks-Archiv abgelegt werden können. Dies würde zwar ein erneutes Kompilieren der SIBs unnötig machen, aber dennoch müsste das Archiv neu erstellt und verteilt werden.

Der Onlineverzeichnisdienst der zweiten Option liefert auf Anfrage zu gegebener SIB-UID die Modulinformationen. Dies setzt voraus, dass projektübergreifend die Syntheseinformationen zu einem gegebenen SIB eindeutig sind. Grundsätzlich verhalten sich die SIBs zwar immer gleich, aber die symbolischen Typnamen müssten konsistent gehalten werden. Es würde also eine globale Pflege dieses zentralen Dienstes vonnöten sein. Die Arbeit des Domänenmodellierers wäre demnach nicht mehr auf sein Projekt beschränkt. Daher wurde im Rahmen der Entwicklung von PROPHETS diese Möglichkeit zunächst nicht implementiert.

Betrachtet man die dritte Option, bei der eine Datei im Projektverzeichnis abgelegt wird, hinsichtlich derselben Aspekte, so sieht man, dass Vor- und Nachteile im Wesentlichen umgekehrt werden. Der Domänenmodellierer kann sich auf sein Projekt beschränken, erhält aber auch keine Erleichterungen durch mitgelieferte oder online abrufbare Syntheseinformationen. Dafür wird der Verwaltungsaufwand lokal beschränkt und ein andauerndes Neukompilieren von SIB-Bibliotheken entfällt.

Nach Abwägung der Vor- und Nachteile ist die dritte Option in PROPHETS umgesetzt worden. Durch das allgemeine Konzept des Modulproviders und die dadurch entstandene Entkopplung von Syntheseinformationen und SIB-Klasse kann dies aber bei Bedarf nachträglich noch mit geringem Aufwand erweitert werden. Die damit unter Umständen einhergehende Erweiterung der Verantwortung des Domänenmodellierers muss dann in Kauf genommen oder durch die Einführung einer vierten Rolle kompensiert werden.

4.2.2 Umsetzung mit XStream

Für die Umsetzung des Konzeptes, die Modulinformationen in einer Datei im Projekt abzulegen, bieten sich zwei Möglichkeiten an. Die erste verwendet ein eigenes Dateiformat, bei welchem die Syntax bedarfsgerecht angepasst ist. Das zweite besteht aus einer Serialisierung der Daten durch eine externe Bibliothek.

```
<module>
 <sibUID>latex-example/Latex</sibUID>
 <outputTypes>
  <string>dvi</string>
 </outputTypes>
 <inputTypes>
  <string>tex</string>
 </inputTypes>
 <branch>default</branch>
 <paramTypeMapping>
  <entry>
    <string>dviOutput</string>
    <string>dvi</string>
  </entry>
  <entry>
    <string>texFile</string>
    <string>tex</string>
  </entry>
 </paramTypeMapping>
 <displayName>Latex</displayName>
</module>
```

Abbildung 4.2: Syntheseinformationen des `Latex`-SIBs

Ein eigenes Dateiformat benötigt einen speziell dafür geschriebenen Parser. Dies ist zum Einen fehleranfälliger als externe fertige Lösungen zu verwenden. Zum Anderen muss der Anwender, in diesem Fall der Domänenmodellierer, eine eigene Sprache dafür lernen. Daher hat sich eine mit *XStream* [XSt09] eingelesene XML-Datei als insgesamt sinnvollste Lösung herausgestellt.

XStream ist eine Bibliothek, die Java-Beans in XML serialisiert und deserialisiert. Hat man einfache Java-Beans mit wenigen Feldern, wie es bei den Modulbeschreibungen der Fall ist, kann die XML-Datei auch problemlos manuell editiert werden. Das Erlernen einer neuen Syntax ist nicht erforderlich, sofern XML bekannt ist. Da sich XML in einer Vielzahl von Domänen als Standard von Konfigurationsdateien und Datensätzen etabliert hat, kann davon allerdings ausgegangen werden.

Abbildung 4.2 zeigt die Moduldefinition für das `Latex`-SIB. Der eindeutige Name für das Modul wird aus SIB-UID und Branch-Name erzeugt. Da die SIB-UID teilweise sehr lang wird und insbesondere bei automatisch generierten SIBs auch nicht unbedingt menschenlesbare Informationen enthält, kann zusätzlich noch ein `displayName` angegeben werden. Dieser wird für Interaktion mit dem Endanwender statt des echten Namens verwendet.

Den Kern der Moduldefinition bilden hingegen sein Ein- und Ausgabeverhalten. Damit werden die Namen der Typen angegeben, die von dem SIB auf dem definierten Branch gelesen beziehungsweise geschrieben werden.

Eine Sonderrolle spielt die Map `paramTypesMapping`. Diese wird benötigt, damit nach

der Synthese die SIB-Parameter der durch die Synthese erzeugten SIBs mit Werten gefüllt werden können. Für jeden SIB-Parameter wird hierfür eine Zuordnung zum in ihm verwendeten Typen angegeben.

4.3 Spezifikation der Constraints

Die Beschreibung zur Spezifikation von Constraints mithilfe von Formelvorlagen trennt sich in zwei Abschnitte. Zunächst wird die Darstellung aus Nutzersicht vorgestellt, gefolgt von Details zur Implementierung und Datenhaltung.

4.3.1 Darstellung

Die Umsetzung des zuvor beschriebenen Konzeptes, Constraints mit Hilfe von Formel-Templates zu erzeugen, ist als `TemplateConstraintsEditor` implementiert. Abbildung 4.3 auf der nächsten Seite zeigt diesen Editor mit den Standard-Templates und einigen daraus instanziierten Constraints für das LaTeX-Beispiel. Die Anzeige ist in zwei Hauptbereiche aufgeteilt. Im oberen Teil werden die verfügbaren Templates aufgelistet, im unteren die gebildeten Instanzen. Klickt der Anwender auf das `[i]` neben einem Template, wird dessen Beschreibung eingeblendet. Beim Klick auf `[+]` wird das Template mit leeren Variablenwerten den instanziierten Templates hinzugefügt.

In dem natürlichsprachlichen Satz der Vorlage werden für jede Variable Comboboxen erstellt, die mit den möglichen Werten gefüllt werden. Diese Werte ergeben sich durch den im Template spezifizierten Wertebereich für die Variable. Dabei sind folgende Angaben möglich:

Konkrete Module schränken die Elemente in der Combobox auf die Blattknoten der Modultaxonomie ein. Bei konsistenter Domäne entspricht dies genau den in der Moduldatei definierten Modulen.

Abstrakte Module werden über die inneren Knoten in der Modultaxonomie gebildet.

Module bilden die Vereinigung aus den zuvor benannten abstrakten und konkreten Modulen. Zur besseren Unterscheidbarkeit werden die abstrakten Module kursiv dargestellt.

Typen verhalten sich analog zu den Modulen. Auch hier existieren die Varianten „abstrakt“ und „konkret“ sowie die Vereinigung aus beidem.

Freitext beschreibt einen vom Anwender einzugebenden Text, der unverändert in die Formelvorlage eingefügt wird.

Durch Klicken auf das geöffnete Schloss werden die Eingaben des Benutzers überprüft und das Constraint geschlossen. Wenn alle Constraints geschlossen sind, kann der Vorgang mit Klick auf „Next“ abgeschlossen werden.

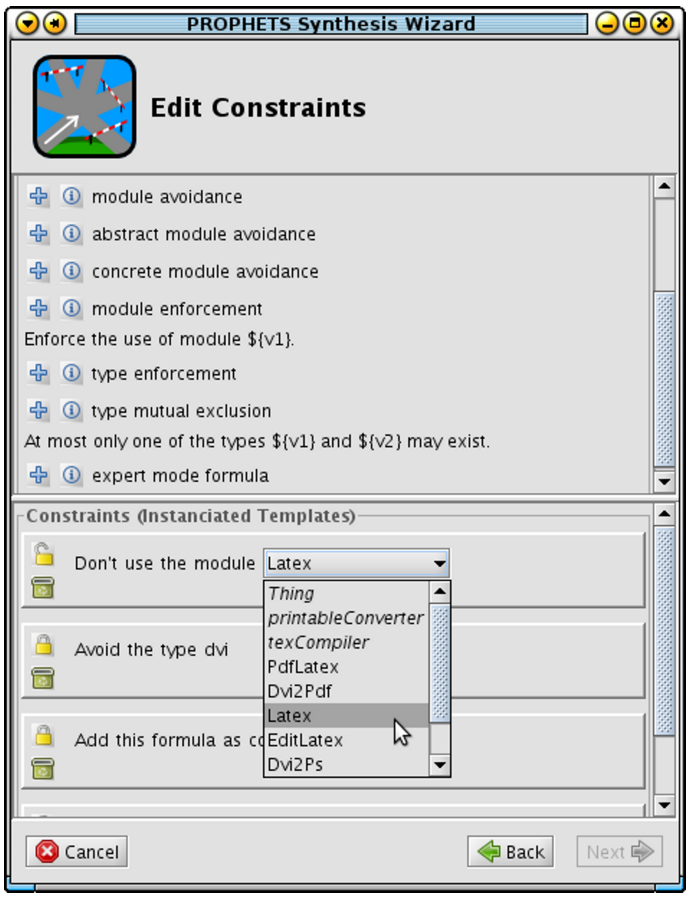


Abbildung 4.3: Der `TemplateConstraintsEditor` im LaTeX-Beispiel

Der `TemplateConstraintsEditor` wird sowohl vom Endanwender als auch vom Domänenmodellierer verwendet. Der Endanwender bearbeitet damit die Constraints zu einer einzelnen Synthese, der Domänenmodellierer hingegen verwendet ihn, um globale Constraints für das gesamte Projekt zu spezifizieren.

4.3.2 Backend

Für die Persistierung von Templates und Instanzen wurde auch hier die Serialisierung mit *XStream* gewählt. Die Gründe dafür sind analog zur Verwendung von XStream bei der Moduldefinitionsdatei zu sehen. Hinzu kommt, dass die Umsetzung mit einem anderen Mechanismus an dieser Stelle dazu geführt hätte, dass sich die Domänenmodellierung inkonsistent gestaltet.

Die Vorlagen müssen vom Domänenmodellierer ins Projekt eingefügt werden. Dazu kann er die Standardvorlagen, die PROPHETS mitliefert, bei Bedarf um domänenspezifische ergänzen. Erstellt er globale Constraints, werden diese dann bei Ausführung des Syntheseprozesses automatisch eingelesen und können vom Endanwender wiederum bearbeitet

werden. Sowohl die Templates als auch die daraus instanziierten projektspezifischen Constraints werden unter festgelegten Dateinamen[2] im Projektverzeichnis abgelegt.

4.4 Erstellen der Taxonomien

Gemäß dem in Abschnitt 3.2.2 „Taxonomien“ auf Seite 23 vorgestellten Konzept werden die Taxonomien als Ontologien im OWL-Format erstellt und mit der OntED-API beim Ausführen des Syntheseprozesses eingelesen. Auch hier sind die Dateien unter festen Namen[3] im Projektverzeichnis abgelegt. Das Erstellen dieser OWL-Dateien kann mit einem der zahlreichen freien OWL-Editoren wie Protégé [Sta09] oder Swoop [KPH05] erfolgen.

Darüber hinaus bietet das PROPHETS-Plugin eine nahtlose Integration der Erstellung der Taxonomien in das jABC. Zu diesem Zweck wird das OntED-Plugin verwendet und durch PROPHETS um spezielle Menüeinträge erweitert, welche den lesenden und schreibenden Zugriff auf die im Projekt liegenden OWL-Dateien verwalten. Auch werden bei Bedarf neue Dateien erstellt und bestehende Dateien automatisch aktualisiert.

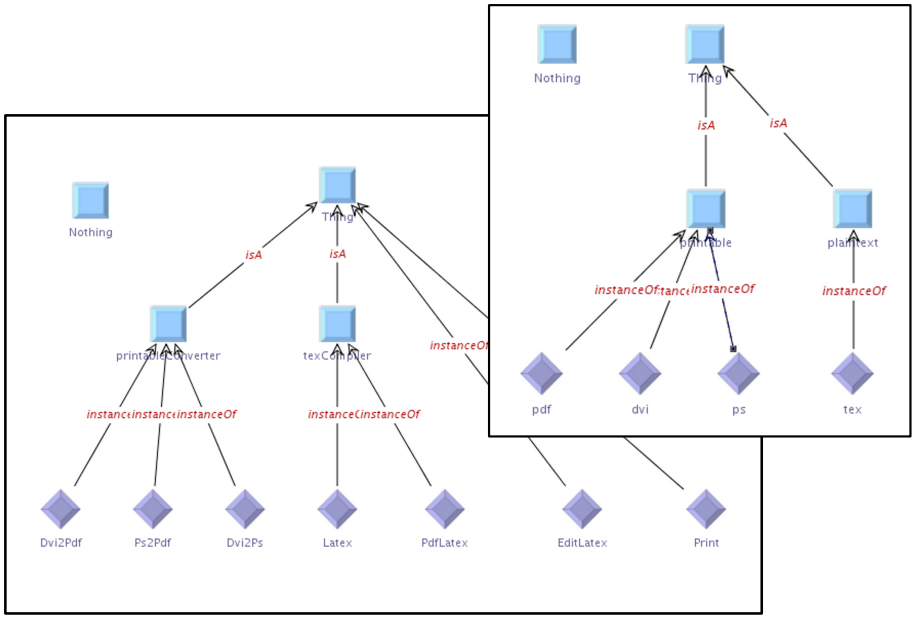


Abbildung 4.4: Modul- und Typtaxonomie im LaTeX-Beispiel

Die konkreten Elemente – also die Typen und Module, die in der Moduldefinition auftauchen – werden in der Ontologie als *Individuen* dargestellt. Die abstrakten Elemente werden mit *Klassen* modelliert. Abbildung 4.4 zeigt die Modul- und Typtaxonomie für das

[2] `prophets/templates.xml`, `prophets/templateInstances.xml`
[3] `prophets/moduleTaxonomy.owl`, `prophets/typeTaxonomy.owl`

LaTeX-Beispiel. Zur optischen Unterscheidung verwendet das OntED-Plugin ein blaues Quadrat für Klassen sowie ein violettes gedrehtes Quadrat für Individuen.

Da alle Klassen und Individuen gemäß der OWL-Semantik Instanz der speziellen Klasse „Thing“ sind, wird diese als Wurzel für die Taxonomie festgelegt. Ein taxonomiespezifisches Wurzelelement mit dem Namen „Module“ oder „Type“ könnte nur zusätzlich zu „Thing“ definiert werden. Eine nach dem Einlesen erfolgende Umbenennung ist aus Gründen der Konsistenz nicht umgesetzt worden. Die Bezeichnung in der Taxonomie wäre von der im `TemplateConstraintsEditor` abgewichen. Daher müssen Endanwender und Domänenmodellierer mit dieser Konvention vertraut sein.

Die Klasse „Nothing“ wird für Klassen benötigt, die über Inferenzregeln erstellt werden. Wenn diese widersprüchlich sind, kann es eine solche Klasse nicht geben. Dies wird in OWL als Instanz von „Nothing“ dargestellt. Da bei den Taxonomien keine inferierten Relationen vorkommen, wird die Klasse „Nothing“ nicht benötigt. Als fester Bestandteil einer Ontologie kann sie dennoch nicht entfernt werden.

Wählt nun der Domänenmodellierer über das PROPHETS-Menü das Bearbeiten der Modul- oder Typtaxonomie[4], so wird direkt die dazugehörige OWL-Datei mit dem OntED-Plugin geöffnet. Sollte die Datei noch nicht vorhanden sein, wird anhand der Moduldefinition eine neue erstellt. Dabei wird jedes Element (je nach Datei also Typ oder Modul) als Instanz von „Thing“ eingefügt. Bei vorhandener Moduldefintion muss der Domänenmodellierer die Taxonomie also nicht vollständig manuell erstellen. Er kann direkt mit der Gruppierung durch Einfügen von Klassen beginnen.

Beim Öffnen einer bereits bestehenden Taxonomie werden die Individuen mit der Moduldefinition abgeglichen. Elemente, die nicht in der Taxonomie, aber in der Moduldefinition enthalten sind, werden wie bei der Neuerstellung der Datei als Instanz von „Thing“ eingefügt. Ein Mitteilungsfenster zeigt dem Domänenmodellierer an, welche Elemente so erzeugt wurden. Zusätzlich können in der Taxonomie Individuen vorhanden sein, die nicht in der Moduldefinition existieren. Diese werden allerdings nicht automatisch entfernt. Der Modellierer erhält darüber jedoch eine Warnmeldung.

4.5 Verifikation von Modellen

Projekte, die für die Verwendung mit PROPHETS aufgesetzt sind, enthalten implizite Informationen über mögliche Reihenfolgen von SIBs. Die Idee liegt nahe, diese Informationen nicht nur zum Synthetisieren neuer Modelle zu verwenden, sondern auch zur Verifikation bereits vorhandener Modelle. Bei diesem *Model Checking* werden einzuhaltende Anforderungen und Eigenschaften als temporallogische Formeln ausgedrückt und gegen ein vorhandenes Modell geprüft. Je nachdem, ob es sich um eine Linear- oder Branching-Time-Logik handelt, wird das Modell unterschiedlich gebildet (vergleiche Abschnitt 2.2.2 auf Seite 12). Mit *GEAR* [Bak06] steht dem jABC ein Model-Checking-Plugin zur Verfügung,

[4] Dieses ist über das jABC-Menü unter „`Plugins/PROPHETS/Domain Setup`“ zu finden.

welches SIB-Graphen verifizieren kann. Im Rahmen von PROPHETS wird damit überprüft, ob SIBs die Typen verfügbar haben, die sie für ihre Ausführung benötigen.

Mit den Ein- und Ausgabeinformationen der SIBs, die aus der Moduldefinition entnommen werden, wird eine *Datenflussanalyse* auf dem Modell durchgeführt, wie sie in Abschnitt 2.3 auf Seite 15 vorgestellt wurde. Die resultierenden Typen werden als *Atomare Propositionen* an die entsprechenden SIBs annotiert. Dadurch kennt GEAR für jedes SIB die Typen, die an seinem Eingang unabhängig vom vorherigen Ausführungsverlauf verfügbar sind. Zusätzlich dazu wird an jedes SIB der dazugehörige Modulname annotiert. Gibt es aufgrund unterschiedlicher Branches verschiedene Module zu demselben SIB, so werden alle diese Modulnamen hinzugefügt.

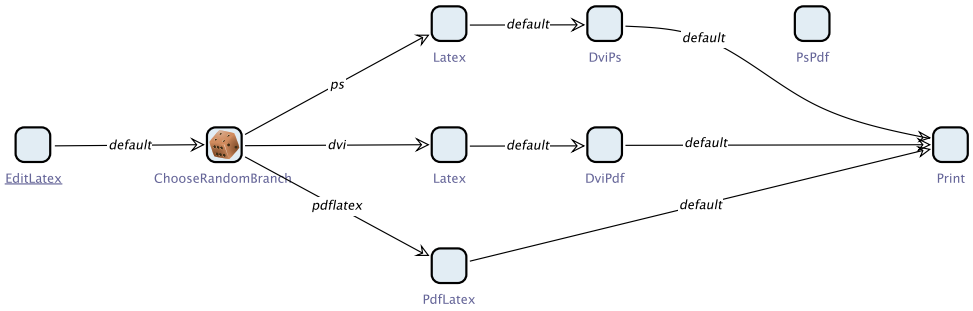


Abbildung 4.5: Beispiel zur Modellverifikation

Abbildung 4.5 zeigt einen Graph, der mit den SIBs aus dem LaTeX-Beispiel modelliert ist. Man erkennt leicht, dass der oberste Pfad nicht gültig ist, da dort der Schritt, der **PS** in **PDF** umwandelt, übersprungen wird. Demnach gibt es einen Pfad, der in `Print` endet, ohne dass eine pdf-Datei generiert wurde. Die Darstellung dieses Fehlers mit GEAR zeigt Abbildung 4.6 auf der nächsten Seite. Im oberen Teil ist der ursprüngliche Graph inklusive Formelerfüllungsrahmen in rot und grün zu sehen. Unten ist die Ansicht mit GEAR auf Anzeige der atomaren Propositionen umgeschaltet. Auch hier kann man erkennen, dass das `Print`-SIB nicht den geforderten Typ **PDF** als Atomare Proposition enthält.

Die Formeln für das Modell werden wie folgt generiert: Für jedes Modul wird eine zu erfüllende Implikation als Formel hinzugefügt. Für das im Beispiel verwendete `Print` wird folgende Formel an das Modell gehängt[5]:

$$\texttt{'module Print'} \Rightarrow \texttt{'pdf'}$$

Bei mehreren Eingabetypen würden diese mittels Konjunktion auf der rechten Seite aufgezählt. Da nun die Datenflussanalyse für das `Print`-SIB nicht die definitive Verfügbarkeit von **PDF** ermittelt hat, ist diese Formel im `Print`-SIB nicht erfüllt und wird in GEAR rot markiert (mittlerer Teil der Abbildung 4.6 auf der nächsten Seite).

[5]Hier wird aus Gründen der besseren Lesbarkeit bewusst auf die Mixfix-Notation von GEAR verzichtet. Die hinzugefügte Formel entspricht der GEAR-Syntax.

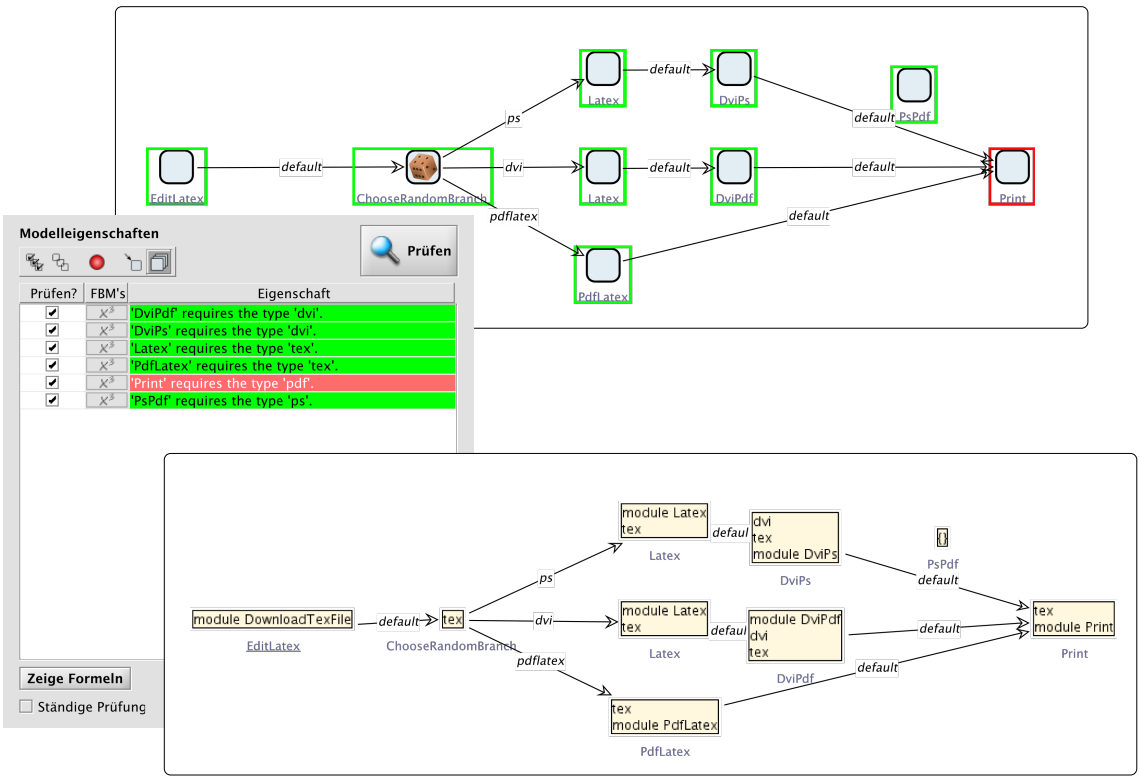


Abbildung 4.6: Modellverifikation mit PROPHETS und GEAR

Eine Alternative für die Verfikation der Eingabetypen kommt ohne Datenflussanalyse aus, benötigt allerdings komplexere Formeln. Dort wird für jeden Typ t folgende Formel erzeugt:

$$\left(\bigwedge_{u \in U(t)} \neg < u > true\right) \quad \mathbf{WU} \quad \left(\bigvee_{g \in G(t)} < g > true\right)$$

Dabei seien $U(t)$ und $G(t)$ die Module, die t als Eingabe („use“) beziehungsweise Ausgabe („gen“) haben. Wichtig ist hierbei, dass diese Formeln nur im Startknoten gültig sein müssen. Ist eine Formel nicht erfüllt, benötigt man zudem die Anzeige des Fehlerpfades, da sonst nur darauf geschlossen werden kann, dass *ein* SIB, nicht *welches*, seine benötigte Eingabe t nicht immer verfügbar hat. Diese Information kann aber auch durch Aufteilen jeder Typformel in $|U(t)|$ Einzelformeln wieder direkt darstellbar gemacht werden.

Im Rahmen dieser Arbeit wurde nur die zuerst vorgestellte Variante implementiert, da sie von der ohnehin für die Bestimmung der Starttypen implementierten Datenflussanalyse profitieren konnte.

5 Syntheseprozessmodellierung

Der Aufgabe des Syntheseprozessdesigners kommt eine besondere Rolle zu, da sie im Rahmen dieser Arbeit erfolgt ist und mit der Enwicklung der PROPHETS-SIB-Bibliothek einherging. In diesem Kapitel werden ausgewähle SIBs dieser Bibliothek vorgestellt sowie Einzelheiten zur Modellierung von Syntheseprozessen motiviert und erläutert. Den Abschluss bildet die Präsentation jener Syntheseprozesse, die bei der Entwicklung von PROPHETS entstanden und in das Plugin integriert sind.

5.1 Wizard

Die Entscheidung, den *Syntheseprozess* mit dem jABC zu modellieren, bringt die zuvor dargestellten Vorteile in Bezug auf Verständnis, Flexibilität und technischen Hintergrund mit sich. Durch die Unabhängigkeit von SIBs untereinander ist es aber erst einmal nicht so leicht möglich, eine gemeinsame GUI zu verwenden. Die Common-SIBs für GUI sind so implementiert, dass jedes SIB seine eigene Darstellungskomponente (zum Beispiel Dialog oder Fenster) öffnet. Dort findet dann die Interaktion mit dem Nutzer statt. Beim Verlassen des SIBs wird diese Komponente dann wieder entfernt. Dieses Verhalten verhindert das Modellieren von Prozessen, die dem Anwender eine konsistente und einheitliche GUI präsentieren sollen. Bei den Common-SIBs wird aber zu Gunsten der besseren Integration in beliebige Prozesse auf diese Konsistenz verzichtet.

Die Modellierung von Syntheseprozessen hingegen findet ausschließlich mit den SIBs der PROPHETS-SIB-Bibliothek statt. Da diese SIBs auch in keinem anderen Kontext agieren müssen, war es möglich, für PROPHETS und die dort verwendeten SIBs ein einheitliches GUI-Konzept zu entwickeln. Aus Sicht des *Endanwenders* soll sich die Ausführung des Syntheseprozesses wie ein *Wizard* [Eck05] gestalten. Das heißt, er sieht während der Ausführung permanent dasselbe Fenster. Mit Buttons wie „Weiter“ und „Zurück“ kann er sich durch eine Sequenz von Seiten navigieren. Mit dem letzten „Weiter“ in der Sequenz wird der Wizard geschlossen.

Um dieser Anforderung gerecht zu werden, wurde für PROPHETS ein eigenes Wizard-Konzept entworfen. Implementierende SIBs greifen über das `ProphetsDataBean` auf ein dediziertes Objekt der Klasse `ProphetsWizard` zu und können von ihm eine leere Zeichenfläche (`JPanel`) anfordern. Abbildung 5.1 auf der nächsten Seite zeigt einen Wizard, der noch nicht mit Inhalt gefüllt wurde. Der rot markierte Bereich ist das eben benannte `JPanel`. Bei Bedarf kann das SIB auch Buttons des Wizards deaktivieren (siehe blaue Markierung im Bild), wenn zum Beispiel ein „Next“ erst dann möglich sein soll,

Abbildung 5.1: Leere Instanz eines `ProphetsWizard`

wenn bestimmte Eingaben getätigt sind. Zudem können Icon und Überschrift im oberen Teil angepasst werden. Klickt der Anwender nun auf einen (nicht deaktivierten) Button, übermittelt der Wizard dem SIB ein zum Button passendes Event. Das SIB speichert dann die in den Elementen der Zeichenfläche eingegebenen Daten im `ProphetsDataBean` und gibt den Wizard wieder frei.

Je nachdem auf welchen Button der Anwender geklickt hat, resultiert das SIB in dem Branch „next“, „back“ oder „cancel“. Somit ist es dem Syntheseprozessdesigner möglich, den Workflow innerhalb des von dem Endanwender durchlaufenen Wizards durch die Modellierung im jABC zu beinflussen. Hierbei tauchen allerdings Probleme im Zusammenhang mit der Datenverwaltung auf. Der Endanwender erwartet beim Klicken auf „Zurück“ sicherlich, dass vorher getätigte Eingaben rückgängig gemacht werden, damit alternative Pfade im Modell mit denselben Daten starten, wie wenn sie direkt gewählt worden wären. Dies ist aber im jABC derzeit so nicht möglich. Die eher technischen Gründe dafür werden in Anhang A.1 auf Seite 63 erläutert. Hier bleibt festzuhalten, dass sich der Syntheseprozessdesigner beim Modelllieren dieser Problematik bewusst sein muss.

5.2 SIB-Bibliothek

Bei der Entwicklung des PROPHETS-Plugins ist eine Vielzahl von SIBs entstanden, von denen die meisten die in Abschnitt 3.3 auf Seite 25 vorgestellten Klassen des *verallgemeinerten Syntheseprozesses* implementieren. Durch die Existenz von Auswahlmöglichkeiten erhält der Syntheseprozessdesigner die Flexibilität in seiner Modellierung. Im Folgenden werden einige ausgewählte SIBs in ihrer Funktion und Implementierung detailliert vorgestellt.

Die SIBs der PROPHETS-SIB-Bibliothek lassen sich anhand zweier Eigenschaften in vier Gruppen unterteilen. Zunächst können *Hilfs-* und *Haupt-SIBs* unterschieden werden. Die Haupt-SIBs sind diejenigen, die eine Funktionalität des verallgemeinerten Syntheseprozesses implementieren. Hilfs-SIBs erledigen Aufgaben zur Vor- und Nachbereitung von Daten oder stellen Kontrollflussmechanismen zur Verfügung. Die zweite Eigenschaft klassifiziert die SIBs bezüglich Benutzerinteraktion. *Interaktive SIBs* verlangen eine Eingabe vom Endanwender, indem sie das zuvor vorgestellte Wizard-Konzept implementieren. Demgegenüber existieren *passive SIBs*, die interaktionslos auf den Daten arbeiten, welche über das `ProphetsDataBean` zur Verfügung stehen.

5.2.1 Bestimmung von Starttypen

Innerhalb der Synthese sind einmal generierte Typen für alle später ausgeführten Module verfügbar. Daher lag die Implementierung eines *Start Types Provider*s nahe, dessen Verhalten sich daran orientiert. Im Gegensatz zum `LocalStartTypesProvider`, der als Starttypen die Ausgaben des Quell-SIBs der markierten Kante wählt, werden beim `DFAStartTypesProvider` die Starttypen mittels *Datenflussanalyse* bestimmt. Der implementierte *Worklist-Algorithmus* ermittelt für jedes SIB des Modells die Menge an Typen, die an seinem Eingang definitiv vorhanden sind, unabhängig über welchen Ausführungspfad dieses SIB erreicht wurde. Zur Bestimmung der Synthesestarttypen werden dann die so ermittelten Typen des Quell-SIBs mit seinen Ausgabetypen vereinigt.

5.2.2 Bestimmung von Zieltypen

Wie bereits in Abschnitt 2.3 angedeutet wurde, erwartet die verwendete Synthesebibliothek keine explizit angegebene Menge von Zieltypen. Stattdessen muss das Ziel in die Formel kodiert werden. Dies geschieht in PROPHETS durch Erzeugung eines speziellen *Goal Constraints*. Im Rahmen dieser Arbeit sind zwei SIBs entstanden, die dieses Constraint erzeugen.

Der `SimpleGoalConstraintProvider` verwendet ein „Eventually“ mit der Konjunktion der Eingabetypen des Ziel-SIBs. Das im Beispiel verwendete `Print`-SIB würde zu dem Constraint

```
F(pdf)
```

führen. Tests mit der Synthesebibliothek haben jedoch gezeigt, dass häufig Lösungen gefunden werden, die nach Erreichen des Ziels noch weitere Module enthalten. Diese sind zwar korrekt im Sinne der Synthese, aber nicht unbedingt von Interesse für den Endanwender. Daher hat er die Möglichkeit, in der Plugin-Konfiguration von PROPHETS die Verwendung des `TerminatingGoalConstraintProviders` auszuwählen. Dieser kodiert die Eingabetypen des Ziel-SIBs so, dass nach ihrem Erreichen keine weiteren Module mehr folgen dürfen. Im Beispiel mit dem Print-SIB resultiert dies in der Formel:

$$\neg \text{ pdf } \mathbf{U} \ (\text{ pdf } \wedge \neg < \mathit{true} > \mathit{true})$$

5.2.3 Suche nach Lösungen

Der `AllPathsExtractor` ist eine naive Implementierung des im verallgemeinerten Syntheseprozess definierten Suchalgorithmus. Er führt eine vollständige *Tiefensuche* auf dem Synthese-TSM durch. Anwendung auf Domänen, die größer sind als das LaTeX-Beispiel, haben aber gezeigt, dass das TSM zu groß werden kann, um es vollständig zu durchsuchen. Aus diesem Grund wurde der `BoundedPathsExtractor` als weiterer Suchalgorithmus hinzugefügt. Dieser führt eine *iterative Tiefensuche* [RN04, S. 111ff.] auf dem Synthese-TSM durch. Dabei ist es möglich, die Suchtiefe mit einem festen Wert zu beschränken. Zusätzlich kann mit einem *Threshold Factor* angegeben werden, bis zu welchem Vielfachen der Länge der ersten gefundenen Lösung der Algorithmus suchen soll.

Durch die Verwendung der iterativen Tiefensuche werden die Vorteile der Tiefensuche mit denen einer Breitensuche verbunden. So garantiert der Aspekt der Breitensuche das Finden der kürzesten Lösung ohne vollständiges Durchsuchen des Graphen. Die Tiefensuche hingegen enthält bei gefundenem Zielknoten die Lösung bereits auf dem Stack. Das Verwalten einer expliziten Vorgängerrelation kann damit entfallen. Da darüber hinaus auch keine besuchten Knoten im Sinne einer Closed-Menge gespeichert werden müssen, ist der Speicherplatzbedarf der Suche deutlich geringer als bei einer Breitensuche. Ein Nachteil besteht allerdings im mehrfachen Durchsuchen der oberen Ebenen. Unter der Annahme, dass der Verzweigungsfaktor durchgängig ähnlich ist, sind aber auf jeder Ebene ohnehin mehr Knoten als auf allen vorangegangenen Ebenen. Die dadurch nur geringfügig erhöhte Laufzeit wird zu Gunsten der genannten Vorteile in Kauf genommen[1].

5.2.4 Filtern von Lösungen

Es hat sich gezeigt, dass die Synthese häufig viele Lösungen liefert, die bis auf die Reihenfolge der verwendeten Module identisch sind. Allgemein kann angenommen werden, dass Permutationen von Lösungen für den Anwender gleichwertig sind. Daher wurde der `PermutationSolutionFilter` implementiert, der die Lösungsmenge so filtert, dass nur die jeweils erste der bis auf Permutation identischen Lösungen übernommen wird.

[1] Für die formale Laufzeitanalyse siehe [RN04, S. 112].

Der dahinterstehende Algorithmus arbeitet so, dass er einmal vollständig über die Menge von Lösungen iteriert. Zu jeder Lösung wird ein Hashwert berechnet, der für Permutationen identisch und ansonsten eindeutig ist. Beim ersten Vorkommen eines Hashwertes wird die Lösung der gefilterten Menge hinzugefügt und der Hashwert gespeichert. Wird später eine weitere Lösung mit demselben Hashwert gefunden, so wird sie übersprungen.

Zur Berechnung des Hashwertes wird der konkatenierte String aus SIB-UID und Branch-Name aller Module einer Lösung herangezogen. Nach diesem wird die Lösung alphabetisch sortiert und dann ein Gesamt-String daraus gebildet. Die Laufzeit wird maßgeblich durch die Anzahl der Lösungen (n) und das Sortieren der Lösung bestimmt. Geht man von einer maximalen Lösungslänge m aus, dann lässt sich die Gesamtlaufzeit durch $O(n \cdot m \log m)$ abschätzen. Da die Lösunglänge aber üblicherweise sehr viel kleiner als die Anzahl der Lösungen ist, kann man die Laufzeit insgesamt als linear in der Anzahl der Lösungen betrachten.

5.2.5 Kontrollfluss

Damit der Syntheseprozess nicht auf eine einfache Abfolge von SIBs beschränkt ist, wurden spezielle Komponenten implementiert, die den Kontrollfluss beeinflussen. Sie fallen damit in die Gruppe der Hilfs-SIBs.

Das passive `ConfigBranching`-SIB liest die Konfigurationsdatei von PROPHETS[2] ein und verzweigt dann anhand des Wertes eines Konfigurationsschlüssels. Der Schlüssel kann im SIB als SIB-Parameter gesetzt werden. Die erwarteten Werte können mittels *Mutable Branches* modelliert werden. Sollte der Schlüssel nicht vorhanden oder ein unerwarteter Wert konfiguriert sein, so wird zu der Kante verzweigt, die mit dem Branch „default“ belegt wurde.

Ein vergleichbares Verhalten zeigt das `WizardBranching`-SIB. Der Unterschied hierbei besteht darin, dass es den Nutzer nach dem zu wählenden Branch fragt. Somit ist es ein interaktives SIB. Eine vergleichbare Funktionalität existiert mit dem `ShowBranchingDialog`-SIB bereits in den Common-SIBs. Folgende zwei Gründe motivieren die Neuimplementierung:

1. Wie bereits zuvor dargelegt, wird durch die Verwendung des `ProphetsWizard` die einheitliche GUI gewahrt.

2. Das WizardBranching-SIB liefert dem Modellierer zusätzlich die Möglichkeit, eine Beschreibung zu jeder Option anzugeben.

Abbildung 5.2 auf der nächsten Seite vergleicht am Beispiel der Auswahl des Start-Types-Providers die beiden SIBs.

[2]Diese ist unter in ${HOME}/.JavaABC/PROPHETS.properties zu finden.

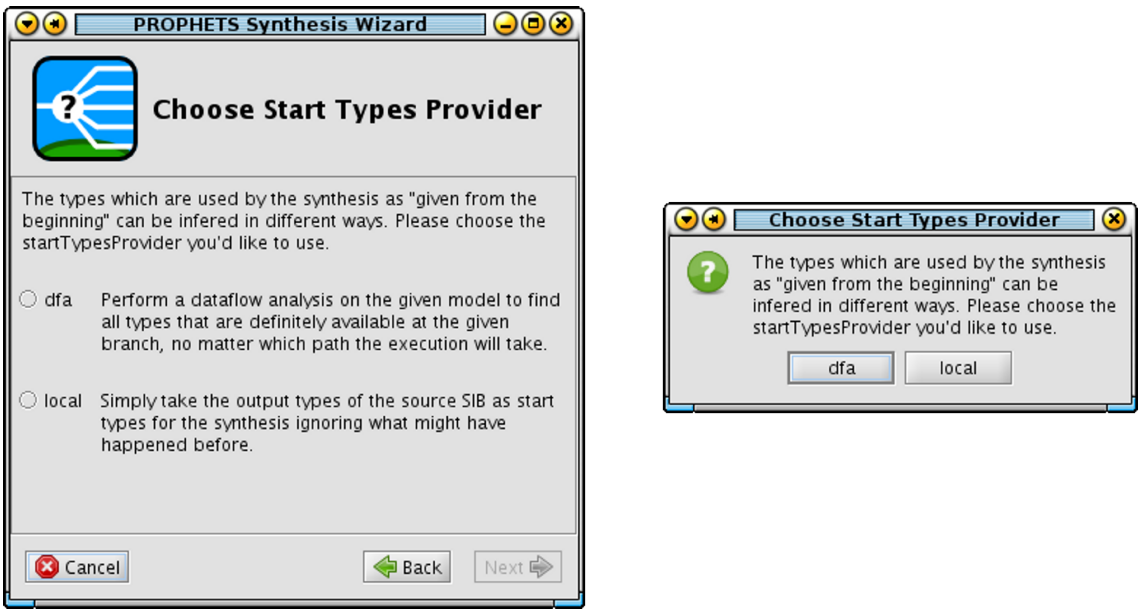


Abbildung 5.2: Vergleich: `WizardBranching` und `ShowBranchingDialog`

5.3 Umsetzung des Syntheseprozesses

Das ursprüngliche Konzept sah genau einen Syntheseprozess vor, der mit dem jABC modelliert und nach der Generierung mit *Genesys* ins Plugin eingebunden wird. Es kann aber verschiedene Benutzerprofile des Endanwenders hinsichtlich Anforderung und Kenntnis geben, sodass dieser Prozess weiter untergliedert werden musste. Da aber die Codegenerierung den einen Syntheseprozess mit genau einem Menüeintrag verbindet, muss die Verzweigung innerhalb dieses Prozesses stattfinden.

An dieser Stelle kommen die zuvor beschriebenen Kontrollfluss-SIBs `ConfigBranching` und `WizardBranching` zum Einsatz. Abbildung 5.3 auf der nächsten Seite zeigt den Syntheseprozess, der anhand der Konfiguration und durch Einblendung des Wizards mit Auswahloptionen zu dem gewünschten Prozess führt.

Nach Vorbereitung der Daten durch zwei initialisierende SIBs wird zunächst geprüft, ob ein Standardprozess konfiguriert ist. Wenn dies der Fall ist, wird direkt zu dem entsprechenden Prozess gesprungen. Ansonsten wird der Anwender gefragt, welchen Prozess er ausführen möchte. Hier ist auch zu sehen, dass mit der Modellierung im jABC beeinflusst werden kann, was beim Klick auf „Zurück“ geschehen soll. Selbst wenn durch das SIB „config: default process“ ein Standardprozess ausgewählt wird, so kommt der Anwender dennoch zum Auswahldialog „zurück“. Dies funktioniert natürlich nur dann, wenn der Prozess noch weitere interaktive SIBs enthält.

Wie in der Abbildung zu sehen ist, sind drei Prozesse als fester Bestandteil von PROPHETS in das Plugin integriert worden. Diese werden im Folgenden kurz vorgestellt. Die wichtigsten verwendeten SIBs dieser drei Prozesse sind in Abbildung 5.4 auf der nächsten Seite

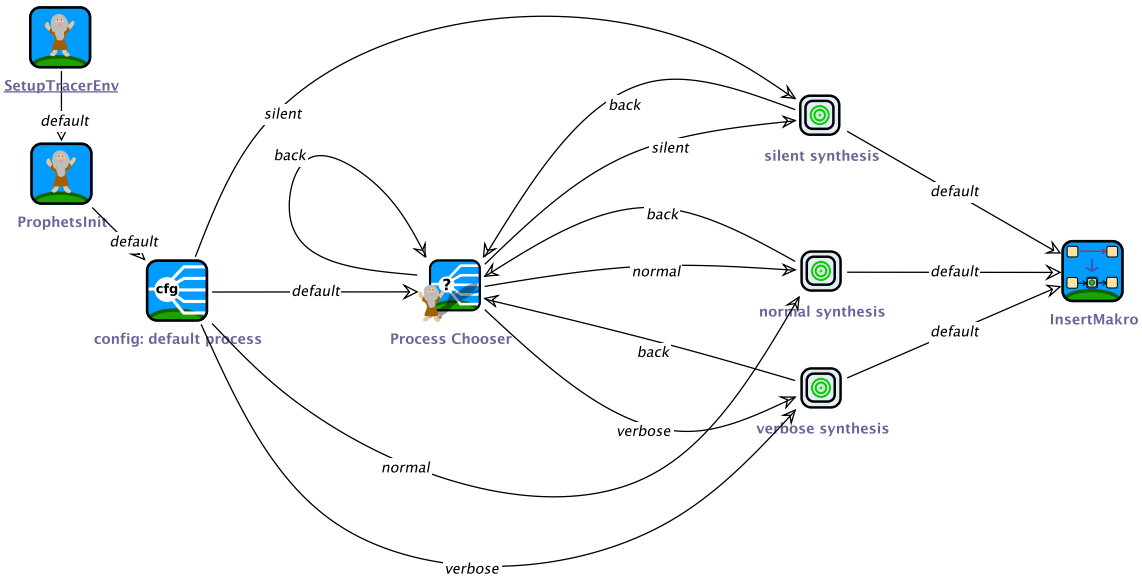


Abbildung 5.3: Auswahl des Syntheseprozesses

Klasse / Prozess	silent	normal	verbose
Module Provider	XStreamModuleProvider		
Constraint Provider	*config*	*config*	*user selection*
Start Types Provider	DFAStartTypesProvider		*user selection*
Goal Constraint Provider	*config*	*config*	*user selection*
Module Tax Provider	OntEDModuleTaxonomyLoader		*user selection*
Type Tax Provider	OntEDTypeTaxonomyLoader		*user selection*
Synthesis Algorithm	*config*	*config*	*user selection*
Search Algorithm	*config*	*config*	*user selection*
Solution Filter	*none*	*config*	*user selection*
Solution Sorter	*none*	SolutionSorterBySize	*user selection*
Solution Chooser	ShortestSolutionChooser	TextualSolutionChooser	*user selection*

Abbildung 5.4: Vergleich der Syntheseprozesse hinsichtlich verwendeter SIBs

anhand der Klassen des *verallgemeinerten Syntheseprozesses* einander gegenübergestellt. Abbildungen der vollständigen Prozesse sind in Anhang A.2 ab Seite 65 zu finden.

normal

In dem Prozess „normal“ ist das Standardverhalten des Plugins abgebildet. Der Endanwender spezifiziert zunächst Constraints und wählt dann aus den von der Synthese gefundenen Lösungen eine aus. Falls keine Lösung gefunden wurde oder keine der gefundenen Lösungen dem Anwender zusagt, kann er mit Klick auf „Zurück“ zum Constraints-Editor zurückkehren und dort die Anforderung verfeinern. Ob die Constraints mit Templates oder direkt in SLTL spezifiziert werden, kann über die Plugin-Konfiguration eingestellt werden.

silent

Dieser Prozess ist für Endanwender gedacht, die weniger den experimentellen Charakter von PROPHETS ausnutzen wollen, sondern schnell nach ausführbaren Möglichkeiten

suchen. Sofern eine Lösung existiert, wird einfach die zuerst gefundene kürzeste ohne weitere Interaktion eingefügt.

verbose

Ursprünglich war dieser Prozess für Anwender gedacht, die von den in den Prozessen „normal“ und „silent“ verwendeten Standardeinstellungen abweichen wollten. Er verwendet für jede Entscheidung ein `WizardBranching`-SIB, welches nach dem nächsten auszuführenden Schritt fragt. Es hat sich aber herausgestellt, dass nur bei wenigen Entscheidungen sinnvolle Standard-SIBs angegeben werden konnten, sodass der „verbose“-Prozess häufig gebraucht wurde. Für eine regelmäßige Verwendung war er allerdings nicht geeignet, da er die vielen Fragen bei jeder Ausführung erneut stellt. Aus diesem Grund wurden für die beiden anderen Prozesse die wichtigsten Entscheidungen mittels `ConfigBranching`-SIB modelliert. Deren Verhalten kann über die Plugin-Konfiguration beeinflusst werden. Der Prozess „verbose“ verbleibt für Fälle, die nicht durch die Konfiguration abgedeckt werden, und als Demonstration der PROPHETS-SIB-Bibliothek weiterhin im Plugin enthalten. In den meisten Fällen wird allerdings eine Verwendung des „normal“-Prozesses mit geeigneter Konfiguration sinnvoller sein.

6 Fallstudien

Als Mittel zur Veranschaulichung der Konzepte hat das LaTeX-Beispiel seinen Zweck erfüllt. Da man es wegen der geringen Größe der Domäne und der wenigen möglichen Lösungspfade nicht als ernstzunehmenden Anwendungsfall ansehen kann, werden im Folgenden zwei größere Fallstudien in der Anwendung mit PROPHETS vorgestellt. Das erste Beispiel befasst sich mit der Syntheseprozessdomäne von PROPHETS selbst. Es zeigt, wie mithilfe des PROPHETS-Plugins *Syntheseprozesse* erzeugt werden können, um dadurch die Arbeit des *Syntheseprozessdesigners* zu erleichtern. Die zweite Fallstudie verlässt die rein informatischen Anwendungsfälle und geht über zur Bioinformatik. Hier werden mit PROPHETS Prozesse zur Analyse von biologischen Sequenzen erzeugt.

6.1 Synthese von Syntheseprozessen

Die Domäne, in der Syntheseprozesse mit der PROPHETS-SIB-Palette modelliert werden, erwies sich als gutes Anwendungsbeispiel für die Synthese. Dies liegt vor allem daran, dass die SIBs sehr typenorientiert konzipiert sind. Durch den verallgemeinerten Syntheseprozess ergeben sich Abhängigkeiten, die durch Synthese aufgelöst werden können.

6.1.1 Modellierung der Domäne

Um die Syntheseprozesse mit PROPHETS synthetisieren zu können, reicht es aus, die Moduldefinitionsdatei zu erstellen. Durch den *verallgemeinerten Syntheseprozess* und das `ProphetsDataBean` ergeben sich unmittelbar die symbolischen Typnamen, sodass der eigentliche Aufwand auf das Erstellen der Datei beschränkt ist.

Mit dieser Grundlage kann ohne Taxonomien schon synthetisiert werden. Abbildung 6.1 auf der nächsten Seite zeigt ein einfaches Beispiel. Durch den `XStreamModuleProvider` ergibt sich **MODULES** als Starttyp für die Synthese. Da das `Solution2Graph`-SIB eine **SOLUTION** als Eingabe benötigt, wird

```
F(solution)
```

als Zielformel erzeugt. Rechts ist ein Ausschnitt der Lösungsmenge abgebildet. Dabei ist zu erkennen, dass die Synthese sinnvolle und unterschiedliche Varianten des verallgemeinerten Syntheseprozesses liefert. Hier wurden insgesamt 192 Lösungen gefunden. Filtert man die bis auf Permutation doppelten Lösungen nicht heraus, so ergeben sich 80.640 Lösungen.

(a) Spezifikation (b) Lösungen

Abbildung 6.1: Beispiel einer Syntheseprozesssynthese

Folgende Aussage stellt eine für diese Domäne sinnvolle allgemeine Information dar: „Wenn der `TextualSolutionChooser` verwendet wird, sollen die Lösungen vorher immer anhand ihrer Länge sortiert werden.“ Dies lässt sich durch die Formel

```
(!<TextualSolutionChooser>true) WU (<SolutionSorterBySize>true)
```

darstellen. Ähnlich lässt sich formulieren, dass bei einer automatischen Auswahl der kürzesten Lösung ein Sortieren nicht sinnvoll ist. Da die in dem verallgemeinerten Syntheseprozess definierten Klassen *Solution Sorter* und *Solution Filter* aus Sicht der Synthese keine Veränderungen an den Daten vornehmen, werden sie nicht automatisch in die Lösungen aufgenommen. Es könnte also noch gefordert werden, dass eine oder beide dieser Klassen auftauchen müssen. Allerdings stellt dies eher eine anfragespezifische Bedingung statt globalem Domänenwissen dar.

Dennoch ergibt sich daraus die Motivation für die Erstellung der Taxonomien. Teilt man mit der Modultaxonomie die SIBs gemäß ihrer Klassen, die durch den verallgemeinerten Syntheseprozess festgelegt werden, können Constraints wie

```
F(solutionSorter)
```

verwendet werden. Weiterhin ist eine Klassifizierung anhand der in Abschnitt 5.2 vorgenommenen Unterteilung in „interaktive“ und „passive“ SIBs sinnvoll. Damit können unter Verwendung des Constraints

```
G(!<interactive>true)
```

Prozesse erzeugt werden, die sich wie der „silent“-Prozess verhalten. PROPHETS findet dann für das obige Beispiel $16 = 2^4$ permutationsfreie Lösungen. Diese Zahl ergibt sich

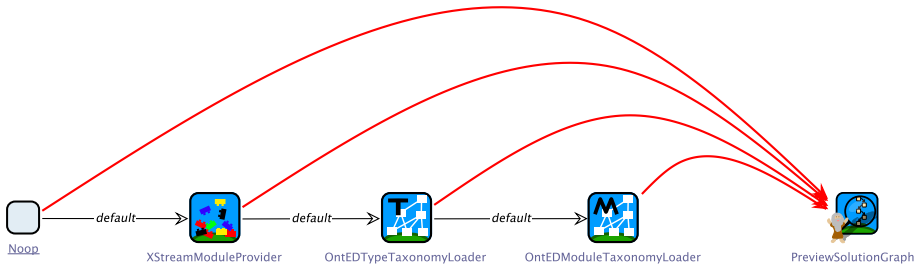


Abbildung 6.2: Beispiel für Laufzeitvergleiche

daraus, dass für vier SIB-Klassen[1] jeweils zwei Varianten existieren, die keine Interaktion benötigen.

6.1.2 Laufzeiten

Obwohl die Domäne mit nur 26 Modulen relativ klein ist, hat sich bei größeren Lösungslängen schnell gezeigt, dass die Laufzeiten sehr lang werden. Ein bequemes Experimentieren durch iterative Verfeinerung der Anforderungen und erneute Suche nach Lösungen ist kaum möglich. Im Folgenden werden Laufzeiten bezüglich unterschiedlicher erwarteter Lösungslängen verglichen. Abbildung 6.2 zeigt die vier durchgeführten Synthesen. Je mehr vom Prozess schon vorgegeben ist, desto kürzer ist die erwartete Lösung. Bei der lose spezifizierten Kante ganz rechts hat diese eine Länge von sieben. Folgende Module werden dabei erwartet: *Constraint Provider*, *Goal Constraint Provider*, *Start Types Provider*, *Synthesis Algorithm*, *Search Algorithm*, *Solution Chooser* und `Solution2Graph`.

Abbildung 6.3 auf der nächsten Seite stellt die beiden Synthesealgorithmen hinsichtlich Laufzeit[2] und Suchraumgröße gegenüber. Dabei ist die Zeile der erwarteten kürzesten Lösungslänge jeweils fett markiert. Die Testläufe wurden mit der iterativen Tiefensuche durchgeführt. Obwohl der gesamte Suchraum durchlaufen werden sollte, wurde sie verwendet, da sie die pro Suchtiefe besuchte Anzahl an Knoten protokolliert. Die aufgeführten Knoten müssen allerdings nicht alle verschieden sein. Da die Tiefensuche aber außerhalb der Kreiserkennung nicht auf Duplikate prüft, zeigen die dargestellten Zahlen die Anzahl der Knotenexpansionen bis zur jeweiligen Suchtiefe.

Im Folgenden werden Beobachtungen auf diesen Daten zusammengetragen. Allerdings ist zu beachten, dass dies nicht zwingend für anders strukturierte Domänen gelten muss. Die hier verwendeten Beispiele zeichnen sich durch eine hohe erwartete Lösungslänge bei relativ kleiner Domäne aus. Gleichzeitig ist die Reihenfolge der Module vor Ausführung des Synthesealgorithmus nahezu beliebig, sodass viele Permutationen möglich sind.

[1] Diese Klassen sind *Start Types Provider*, *Goal Constraint Provider*, *Synthesis Algorithm* und *Search Algorithm*.

[2] Die Zeitmessungen wurden mit einem Intel Core 2 Duo (3,15 GHz) und 2GB RAM unter Linux durchgeführt.

Suchtiefe	Mosel	ETI	ETI/Mosel	Suchtiefe	Mosel	ETI	ETI/Mosel
1	9	26	2,89	1	9	25	2,78
3	223	1.158	5,19	3	303	1.279	4,22
5	1.663	14.598	8,78	5	2.751	21.322	7,75
7	**8.575**	**105.318**	**12,28**	7	15.231	156.682	10,29
8	29.311	234.918	8,01	**8**	**32.511**	**400.522**	**12,32**
9	146.239	351.846	2,41	9	103.551	886.282	8,56
Summe	336.936	1.104.810	3,28	10	548.991	1.331.722	2,43
Zeit (s)	1,1	2,1	1,91	Summe	1.260.060	4.195.541	3,33
Lösungen	4.608	4.608	1,00	Zeit (s)	3,5	6,4	1,83
Perm-Frei	192	192	1,00	Lösungen	17.280	17.280	1,00
				Perm-Frei	192	192	1,00

Suchtiefe	Mosel	ETI	ETI/Mosel	Suchtiefe	Mosel	ETI	ETI/Mosel
1	10	26	2,60	1	7	19	2,71
3	456	1.674	3,67	3	203	731	3,60
5	6.720	40.518	6,03	5	4.479	20.427	4,56
7	35.840	333.638	9,31	7	34831	285937	8,21
8	80.640	839.878	10,42	8	79.311	739.633	9,33
9	**170.240**	**2.103.238**	**12,35**	9	177.167	1.851.633	10,45
10	510.720	4.620.998	9,05	**10**	**372.879**	**4.609.393**	**12,36**
11	2.849.280	6.959.558	2,44	11	1.066.767	10.107.121	9,47
Summe	6.521.022	21.998.542	3,37	12	6.226.447	15.266.801	2,45
Zeit(s)	19,1	38,3	2,01	Summe	14.204.409	48.240.007	3,40
Lösungen	89.600	89.600	1,00	Zeit (s)	44,3	83,6	1,89
Perm-Frei	192	192	1,00	Lösungen	195.712	195.712	1,00
				Perm-Frei	192	192	1,00

Abbildung 6.3: Vergleich von Laufzeiten und Suchraumgrößen

- Die ETI-Synthese braucht fast doppelt so lange wie die Mosel-Synthese, wertet aber insgesamt das 3,3-fache der Knoten aus. Auch wenn die Verwendung der Mosel-Synthese hier sinnvoller erscheint, expandiert die ETI-Synthese ihre Nachfolger offensichtlich schneller. Dies könnte dazu führen, dass in anderen Domains die ETI-Synthese sinnvoller ist.
- Die gefundenen Lösungen unterscheiden sich nicht zwischen den Algorithmen.
- Da für die Taxonomie- und Modulprovider keine Alternativen existieren, bleibt die Gesamtzahl der gefundenen Lösungen gleich, auch wenn man diese vorher nicht festlegt. Lediglich die Anzahl der Permutationen wird dadurch erhöht.
- Der Verzweigungsfaktor sinkt mit der Suchtiefe. Bei ETI ist er anfangs deutlich höher. Bei Mosel steigt er nach dem Finden einer Lösung wieder deutlich an.

6.2 Sequenzanalyse in der Bioinformatik

Die erste Fallstudie hat sich automatisch bei der Implementierung des PROPHETS-Plugins ergeben. Im Folgenden wird eine weitere Domäne vorgestellt, die als Fallstudie vollständig neu aufgesetzt wurde. Zunächst findet eine inhaltliche Einführung in Bioinformatik statt, um die Grundlage zum Verständnis der hier verwendeten Beispiele zu schaffen. Danach folgt ein Abschnitt zur Modellierung der Domäne, in dem die verwendeten Tools, aber auch Größenordnungen, Abbildungen und Erfahrungen präsentiert werden. Den Abschluss bilden konkrete Beispielprozesse, die mit Synthese konkretisiert werden können.

6.2.1 Bioinformatik

Bioinformatik ist ein interdisziplinärer Forschungsbereich, der in den letzten Jahren stark an Bedeutung gewonnen hat. Sie bezeichnet die Anwendung von Methoden der Informatik in der Biologie. Hier profitiert besonders die Genomik von Algorithmen und immer schneller werdenden Rechnern. Bioinformatik trägt einen großen Anteil an der Entschlüsselung des menschlichen Genoms durch das Humangenomprojekt. Im Rahmen des Wachstums dieses Bereichs haben sich eine Reihe von Diensten und Bibliotheken entwickelt. Eine sehr große frei verfügbare Sammlung von Tools zur Analyse von Basensequenzen wird vom EMBOSS-Projekt [RLB00] zur Verfügung gestellt. Aktuell fasst EMBOSS etwa 170 Tools [EMB09]. Für diese Fallstudie wurde ein Großteil dieser Tools als PROPHETS-Domäne modelliert. Darüber hinaus wurden zusätzliche Tools integriert, die Zugriff auf öffentliche Datenbanken ermöglichen.

Die Sequenzanalysetools arbeiten auf einer kodierten Repräsentation der *Nukleinsäuren DNA* und *RNA*. Diese bestehen aus einer Kette von *Nukleotiden*, welche wiederum aus einer *Nukleobase* sowie einem Zucker/Phosphat-Anteil bestehen. Letzterer bildet das sogenannte Rückrad der Sequenz und ist in jedem Nukleotid innerhalb einer Nukleinsäure identisch. Daher sind für Analysen nur die Nukleobasen relevant. Bei der DNA, welche die Erbinformationen enthält, sind *Adenin*, *Guanin*, *Cytosin* und *Thymin* die vorkommenden Nukleobasen. Die RNA enthält *Uracil* statt Thymin. Ihre Anfangsbuchstaben werden zur Kodierung innerhalb einer Sequenz verwendet.

Statt die Sequenz direkt mit den vier Basen zu kodieren, können auch die aus ihnen erzeugten Aminosäuren verwendet werden, da diese zum Teil aussagekräftigere Analysen erlauben. Hierzu werden jeweils Dreiergruppen der Nukleobasen zu einer Aminosäure übersetzt. Insgesamt gibt es 20 Aminosäuren, die dann auch mit einem Einbuchstabencode in sogenannten *Proteinsequenzen* repräsentiert sind (siehe [SMR08, S. 34]).

Eine häufig benötigte Analyse ist das *Sequenzalignment*, bei dem die Ähnlichkeit zweier Sequenzen bestimmt wird. Dies wird zum Beispiel bei der DNA-Sequenzierung oder der Untersuchung von Artenverwandtschaften benötigt. Dabei wird zwischen globalem und lokalem sowie paarweisem und multiplem Alignment unterschieden. *Globales Alignment* vergleicht vollständige Sequenzen, wohingegen *lokales Alignment* ähnliche Bereiche in den Sequenzen findet. Beim *paarweisen Alignment* werden dabei genau zwei Sequenzen analysiert. Das *multiple Alignment* analysiert mehrere Sequenzen auf einmal.

Vergleicht man zwei gleich lange Sequenzen direkt miteinander, so können zwei Fälle auftreten. Sind die Basen beider Sequenzen an derselben Position identisch, spricht man von einem „Match“. Ein „Mismatch“ bezeichnet dagegen, dass die beiden Basen verschieden sind. Die Gesamtähnlichkeit eines Alignments wird „Score“ genannt und ergibt sich aus der Summe von Einzelscores für die Positionen. Entscheidend für deren Bestimmung ist eine Kostenmatrix, die den Score für jede mögliche Paarung angeben. Vereinfacht sei im Folgenden hierfür die Einheitsmatrix gewählt, bei der ein Match einen Score von 1 und ein Mismatch einen Score von 0 hat. Die Idee beim Alignment ist nun, durch Einführung von „Gaps“ (Lücken) die Anzahl der Matches zu erhöhen. Dabei bekommt ein Gap in

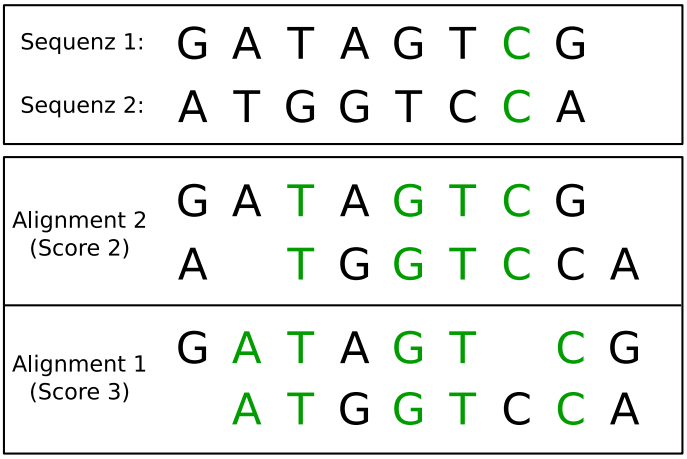


Abbildung 6.4: Beispiel: Sequenzalignment

der Regel einen negativen Score, der im Folgenden auch vereinfacht auf -1 festgelegt ist. Abbildung 6.4 zeigt ein Beispiel mit zwei Sequenzen, die direkt verglichen einen Score von 1 haben, da sie nur auf der vorletzten Position übereinstimmen. Darunter sind zwei mögliche Alignments abgebildet, die durch Gaps einen Score von 2 beziehungsweise 3 erzielen. Die Laufzeit zur Bestimmung eines Alignments mit maximalem Score ist $O(n \cdot m)$, wobei n und m die Längen der beiden Sequenzen bezeichnen[3]. Die Bestimmung von optimalen multiplen Alignments ist hingegen exponentiell in der Anzahl der Sequenzen, sodass hier häufig heuristisch arbeitende Algorithmen zum Einsatz kommen.

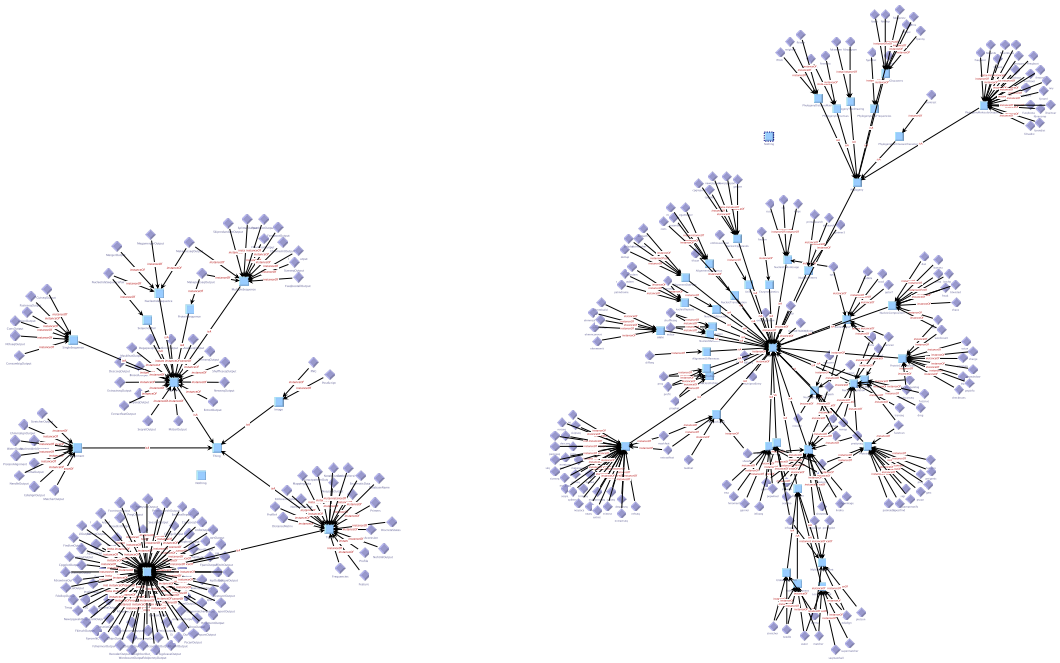
6.2.2 Modellierung der Domäne

Bei der zuvor vorgestellten Modellierung der Syntheseprozessdomäne ergaben sich die symbolischen Typnamen aus der Spezifikation des verallgemeinerten Syntheseprozesses. Hier hingegen mussten zunächst gemeinsame Ein- und Ausgaben der siebenfachen Menge an Services identifiziert und klassifiziert werden. Allerdings hat sich gezeigt, dass dies zum Teil nicht möglich ist, weil häufig keine einheitlichen Datenformate verwendet werden.

Viele der Tools sind zudem schwer zu beschreiben, da sie sehr universell sind. Die technische Beschränkung auf konkrete Ein- und Ausgabetypen sorgt dafür, dass unterschiedliches Verhalten nur durch Verfielfältigung der SIBs zu modellieren ist. Konzeptuell ist dies zwar kein Problem, führt aber zu deutlich erhöhtem Aufwand für den Domänenmodellierer.

Ein weiteres Problem ist, dass verschiedene Variablen desselben Typs nicht existieren können. Gerade bei der Sequenzanalyse werden aber häufig mehrere Sequenzen benötigt. In gewissem Grade kann dies durch Einführung symbolischer Typen wie **sequencePair** oder **multipleSequence** simuliert werden. Allerdings kann ein Tool, welches **sequencePair** als Eingabe benötigt, nicht verwendet werden, wenn zuvor zwei Sequenzen mit doppelter

[3] Dies geschieht durch dynamische Programmierung, siehe Smith-Waterman-Algorithmus [SW81].



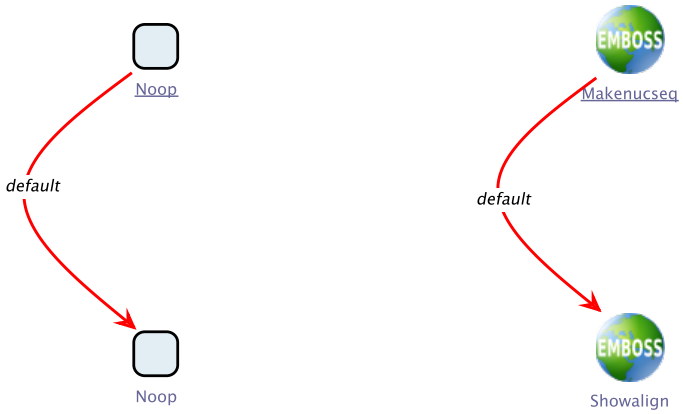
(a) Typ-Taxonomie (b) Modul-Taxonomie

Abbildung 6.5: Taxonomien der Bioinformatikdomäne

Anwendung eines Tools generiert wurden, welches **SEQUENCE** erzeugt.

Im Folgenden werden die Tools aus der Domäne vorgestellt, die für die im nächsten Abschnitt beschriebenen Beispiele benötigt werden. Einen Eindruck von der gesamten Domäne liefert Abbildung 6.5.

`Makenuqseq`	erzeugt mehrere zufällige Nukleotidsequenzen.
`Seqret`	ermöglicht das Lesen von Sequenzen aus einer Vielzahl von Quellen und Datenformaten.
`Emma`	ist die EMBOSS-Kapselung von ClustalW, einem verbreiteten Algorithmus für globales multiples Alignment.
`Edialign`	implementiert einen Algorithmus für lokales multiples Alignment.
`Showalign`	zeigt ein Alignment an.
`GetSeq (DDBJ)`	holt anhand einer ID eine Nukleotidsequenz aus der DNA Data Bank of Japan (DDBJ).
`GetSeq (UniProt)`	holt anhand einer ID eine Proteinsequenz aus der DDBJ.
`BLAST`	ist ein Algorithmus, der in einer Datenbank nach ähnlichen Sequenzen sucht. Das hier verwendete SIB führt zu einer gegebenen Nukleotidsequenz in dem Datenbestand der DDBJ eine Suche nach ähnlichen Proteinsequenzen durch.



(a) Beispiel ohne Rahmenmodell (b) Beispiel mit Rahmenmodell

Abbildung 6.6: Einfache Prozessbeispiele in der Bioinformatikdomäne

6.2.3 Prozessbeispiele

Eine der einfachsten Anforderungen eines Anwenders, der mit der zuvor beschriebenen Domäne zu arbeiten beginnt, ist: „Ich möchte einen Prozess haben, der mir eine Sequenz erzeugt.“ Stellt er zunächst keine weiteren Rahmenanbedingungen, so lässt sich dies in einem Modell ausdrücken, welches aus zwei `Noop`-SIBs als Rahmen für eine markierte Kante besteht (vergleiche Abbildung 6.6(a)). Die Anforderung wird dann über eine Instanz des Templates „type enforcement“ formuliert, für welches er aus der Auswahlliste den abstrakten Typ „Sequence“ auswählt. Den resultierenden kürzesten Lösungen kann entnommen werden, dass es zur Generierung von Sequenzen drei SIBs gibt, die einzeln operieren können. Nach dieser ersten Evaluation kann der Anwender also sein Rahmenmodell konkretisieren. Der Prozess soll durch Verwendung von `Makenuqseq` mit einer zufälligen Nukleotidsequenz starten und im `Showalign`-SIB enden. Abbildung 6.6(b) zeigt diese Abwandlung des Prozesses. Die Synthese liefert hier als kürzeste Lösungen `Edialign` und `Emma`, die beiden zuvor vorgestellten Algorithmen für multiples Alignment.

Problematisch äußert sich schon bei diesem Beispiel die Größe der Domäne. Die Synthese kann ohne weitere Constraints nicht entscheiden, wie sinnvoll oder zielführend ein Pfad ist. Alles, was typkonsistent ist, wird als gültig angenommen. Dadurch ergibt sich ein extrem großer Suchraum, der bei der Mosel-Synthese auf einer Suchtiefe von 4 bereits 18 Millionen Knoten enthält. Bei der ETI-Synthese sind dies sogar 47 Millionen Knoten. Eine Suchtiefe von 5 ist bei beiden nicht mehr durchführbar. Es lässt sich aber abschätzen, dass etwa 1-2 Milliarden Knoten auf dieser Tiefe liegen.

Ein weiteres Beispiel ist in Abbildung 6.7 auf der nächsten Seite dargestellt. Hier ist der vorgegebene Rahmenprozess schon komplexer. Zunächst wird aus einer Datenbank eine Nukleotidsequenz geladen, die dann dafür verwendet wird, mit BLAST eine Suche nach ähnlichen Proteinsequenzen durchzuführen. Die Ausgabe von BLAST enthält die IDs der gefundenen Sequenzen. Über diese IDs wird iteriert, um die Sequenzen aus der Protein-

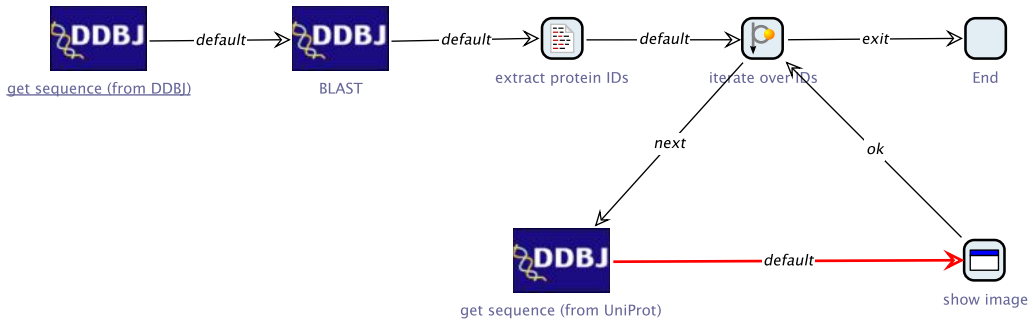


Abbildung 6.7: Komplexeres Beispiel in der Bioinformatikdomäne

datenbank zu laden. Die lose spezifierte Kante, die im `ShowImage`-SIB endet, drückt nun aus, dass irgendwie mit dieser geladenen Sequenz ein Bild erzeugt und angezeigt werden soll. Die Synthese liefert hier unter anderem einige einschrittige Lösungen (vergleiche Abbildung 6.8 auf der nächsten Seite).

Wünschenswert wäre an dieser Stelle ein Ausschluss von Modulen, die auf Nukleotidsequenzen arbeiten. Damit könnte erreicht werden, dass sicher die zuvor geladene Proteinsequenz angezeigt wird, und nicht die Nukleotidsequenz, die zu Beginn geladen wurde. Hierzu müsste es in der Modultaxonomie eine Klassifizierung nach Nukleotidsequenz als Eingabetyp geben. Diese manuell hinzuzufügen würde allerdings bedeuten, dass der Domänenmodellierer redundante Informationen zwischen Moduldefinition und -taxonomie verwalten muss. Ein Mechanismus, der diese Informationen automatisch erzeugt, wäre möglich, ist aber im Rahmen dieser Arbeit nicht implementiert worden.

6.3 Erfolge und Grenzen

Schon an den beiden vorgestellten Fallstudien lässt sich erkennen, dass der Aufwand, den der Domänenmodellierer betreiben muss, sehr stark von der Struktur der Domäne abhängen kann. Schwierig wird es dann, wenn keine eindeutigen Datentypen identifizierbar oder nicht hinreichend genau spezifiziert sind. Ist die Domäne einmal erstellt, kann PROPHETS aber wie geplant eingesetzt werden, um den Endanwender beim Experimentieren zu unterstützen.

Als primäres Problem hat sich jedoch die Größe der Domäne herausgestellt. So ist zum Beispiel das Bearbeiten der Taxonomien mit OntED relativ träge. Bei der Syntheseprozessdomäne fällt dies noch nicht ins Gewicht; bei der Bioinformatikdomäne schon deutlicher. Dies ist vermutlich darauf zurückzuführen, dass der in OntED verwendete Reasoner bei jeder Änderung am Modell die vollständige Ontologie auf Konsistenz prüft.

Noch viel deutlicher hat sich allerdings ein Optimierungsbedarf hinsichtlich der Suchstrategie aufgetan. Obwohl mit der iterativen Tiefensuche eine Verbesserung im Vergleich zur vollständigen Tiefensuche erzielt wurde – und dadurch in der Bioinformatikdomäne

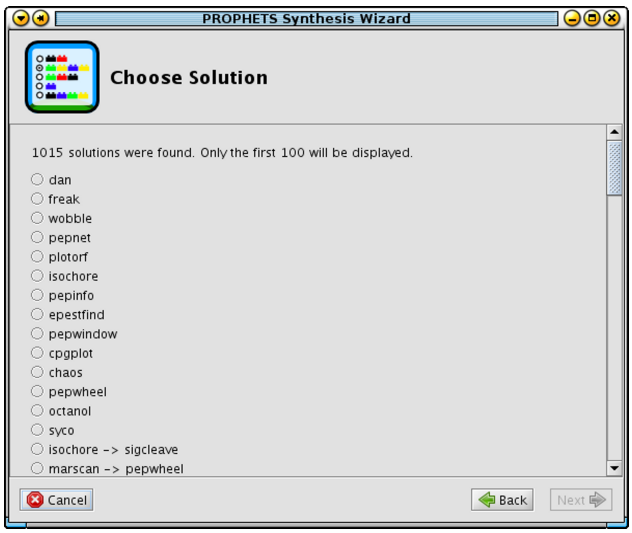


Abbildung 6.8: Lösungen für die Suche nach bilderzeugenden Modulen

überhaupt erst Lösungen zu finden waren – sind die Laufzeiten häufig noch sehr lang. Dies liegt an dem recht hohen Verzweigungsfaktor des Synthese-TSM. Wenn nun die kürzeste Lösung schon lang oder aber der *Threshold Factor* (siehe Seite 44) zu groß gewählt ist, wird die Beschränkung der Suchtiefe so hoch, dass die Anzahl zu besuchender Knoten in nicht akzeptable Bereiche wächst. Als Abbruchkriterium könnte noch die maximale Anzahl zu besuchender Knoten hinzugefügt werden. Dies würde allerdings dazu führen, dass vielleicht gar keine Lösung gefunden wird. Sinnvoller wäre, die Suche zielgerichtet mit mit einer Heuristik vorgehen zu lassen. Das Finden einer geeigneten Heuristik dürfte dabei das größte Problem darstellen. Zu untersuchen wäre, ob bessere Laufzeiten erzielt werden können, wenn der Abstand zum Ziel in einem abstrakten Suchraum als Heuristik herangezogen wird. Eine Möglichkeit dieser Abstraktion wäre zum Beispiel, das ursprüngliche Universum unter Vernachlässigung der Formelauswertung zu verwenden. Da eine reale Lösung niemals kürzer sein kann als der Weg in diesem Universum, könnte man den dortigen Abstand als untere Schranke für die Kostenabschätzung nehmen[4]. Auf ähnliche Weise könnte bei der Mosel-Synthese zusätzlich der Automat, der aus der Formel erzeugt wird, als Abstraktion herangezogen werden.

[4]Es handelt sich dabei um eine „konsistente“ und „zulässige“ Heuristik, sodass sich eine Suche mit „Iterative Deepening A*“ anbieten könnte. Siehe hierzu auch [RN04, S. 139].

7 Abschließendes

In diesem abschließenden Kapitel wird zunächst zusammengefasst, was mit dieser Arbeit erreicht wurde. Der zweite Abschnitt gibt dann einen Ausblick auf mögliche Erweiterungen der Konzepte und Implementierung.

7.1 Zusammenfassung

Mit PROPHETS ist eine Erweiterung des jABC enstanden, die den Anwendungsexperten in vielerlei Hinsicht unterstützen kann. Über die übliche Modellierung im jABC hinaus kann er nun lose spezifizierte Kanten einsetzen, die vom Plugin automatisch konkretisiert werden. Durch die Generierung von Verifikationsformeln kann er aber auch ohne Anwendung der Synthese von einer modellierten Domäne profitieren.

Aus technischer Sicht erwies sich die Entscheidung, den Syntheseprozess mit dem jABC zu modellieren, als sehr positiv. Bei der Entwicklung des Plugins und dem Experimentieren mit den Anwendungsfällen ergaben sich häufig neue Anforderungen, die problemlos in die bestehenden Prozesse integriert werden konnten. Zu nennen wäre hier der `PermutationSolutionFilter`. Dieser war usprünglich nicht vorgesehen, konnte aber schnell und unabhängig vom bereits bestehenden Code implementiert werden. Zur Erweiterung des Syntheseprozesses um diese Funktion musste dann nur das dazugehörige SIB eingefügt werden. Andere Beispiele hierfür sind die Implementierung effizienterer Suchalgorithmen oder des `ConfigBranching`-SIBs, welches Kontrollfluss ohne Benutzerinteraktion ermöglichte.

Darüber hinaus ergab sich durch die Modellierung im jABC die erste Fallstudie. Das Bootstrapping – also die Synthese von Syntheseprozessen – war nicht von vornherein geplant. Durch die ohnehin erfolgte Modellierung im jABC und die typenorientierte Sichtweise auf die Anforderungen der SIBs untereinander hat sich die Anwendbarkeit allerdings automatisch ergeben. Der Aufwand zur Umsetzung bestand lediglich im Erstellen der Moduldefinition, was nur wenige Minuten in Anspruch nahm. Damit wurde auch gleichzeitig die Möglichkeit geschaffen, die zu diesem Zeitpunkt bereits bestehenden Syntheseprozesse „silent“, „normal“ und „verbose“ hinsichtlich Typkonsistenz zu verifizieren.

PROPHETS unterscheidet sich in wesentlichen Aspekten von der ETI-Plattform. Auch wenn die Ideen grundsätzlich auf ETI basieren, so wurden doch einige entscheidende Erweiterungen vorgenommen.

- Der Synthesealgorithmus ist nicht mehr auf Tools begrenzt, die genau eine Ein- und

Ausgabe besitzen. Durch die Zustandsdefintion als Typmenge statt eines einzelnen Typs können zudem Daten verwendet werden, die weiter zurückliegend generiert wurden. Damit lassen sich reale Anwendungsfälle deutlich besser modellieren.

- Da die Integration und Pflege von Tools im ETI-System aufwändig war, wurden nur wenige Tools eingebunden. PROPHETS hingegen kann zur Synthese alle Dienste einbeziehen, die als SIB vorliegen. Dies können lokale Funktionen sein, aber auch zum Beispiel Web Services, die mithilfe von SIBs aufgerufen werden.
- Durch das Template-Konzept und dessen einfache Erweiterbarkeit kann der Anwender die Synthese auch dann verwenden, wenn er keine Formeln in SLTL formulieren kann. Es gab zwar bei ETI bereits einige vereinfachende Pattern, aber mit diesen mussten dennoch Formeln ausgedrückt werden. Der Schritt zum Lückentext mit selektierbaren Werten löst sich komplett von der Formelsicht.

7.2 Ausblick

Über die in Abschnitt 6.3 „Erfolge und Grenzen“ auf Seite 57 dargestellten notwendigen Optimierungen hinaus haben sich bei der Bearbeitung des Projekts weitere Ideen und Erweiterungen ergeben, die das Arbeiten mit PROPHETS verbessern und vereinfachen können.

Mehr Übersicht für den Endanwender

Mit dem Taxonomy-Editor-Plugin können die SIBs eines Projektes strukturiert und umbenannt werden. Diese Information wird daraufhin vom jABC zur Anzeige von SIBs statt der Klassen- und Paketnamen verwendet. Durch die Existenz einer Domänenmodellierung für PROPHETS könnten diese SIB-Taxonomien automatisch erstellt werden. Dies führt zum Einen dazu, dass die SIBs automatisch so strukturiert werden, wie es in der Domäne sinnvoll ist. Zum Anderen erscheinen dann in der SIB-Ansicht genau die Bezeichnungen, wie sie auch in der Synthese auftauchen.

Weiterhin wäre ein *Solution Chooser* sinnvoll, der Informationen zu den Modulen und den dazugehörigen SIBs anzeigen kann. Der Endanwender kann dann besser experimentieren, weil er zu den Lösungen direkt die Dokumentation einsehen kann. Dadurch sollte sich die Qualität einer Lösung deutlich besser einschätzen lassen.

Zusätzlich zu dem `PermutationSolutionFilter` wäre noch ein weiterer *Solution Filter* denkbar, der längere Lösungen herausfiltert, wenn bereits eine kürzere Lösung existiert, die vollständig in der längeren enthalten ist.

Mehr Flexibilität für den Domänenmodellierer

Es hat sich gezeigt, dass die im Konzept festgelegte Zuordnung von SIBs zu Modulen anhand von SIB-UID und Branch-Name in der Praxis nicht immer ausreichend ist. Häufig ist der Funktionsumfang eines SIBs vielfältiger und unterscheidet sich in Abhängigkeit

von Parametern oder tatsächlichem Typ der Eingabe. Das heißt, dass für eine Kombination aus SIB-UID und Branch-Name wiederum mehrere Module zu erzeugen sein sollten. In diesem Zusammenhang ist es auch sinnvoll, die Moduldefinition um statische Parameter zu erweitern. Dies sind SIB-Parameter, die unabhängig von Ein- und Ausgabe, und daher nicht für die Synthese relevant sind. Sie können vom Domänenmodellierer mit festen Werten belegt werden, um die Funktionen des SIBs zu steuern. Dadurch muss der Endanwender nicht alle Parameter eines SIBs detailliert kennen. Zur Zeit ist dies nur möglich, indem mehrere SIBs für dieselbe Funktionalität angeboten werden, bei denen diese statischen Parameter nicht im SIB deklariert sind, sondern unmittelbar an den Service Adapter übergeben werden.

Lose Integration des Syntheseprozesses

Der Syntheseprozess ist aktuell fest in das PROPHETS-Plugin hineinkompiliert. Besser wäre hier, wenn das Syntheseprozessmodell in eine existierende jABC-Installation eingefügt werden könnte, ohne dass das Plugin neu kompiliert und verteilt werden muss. Vorstellbar wäre auch eine Platzierung im ohnehin exisitierenden jABC-Konfigurationsverzeichnis. Existiert dort bei erster Ausführung der Syntheseprozess nicht, wird die im Plugin-Archiv vorhandene Variante dorthin kopiert.

Semantische Datenflussanalyse

Durch eine mögliche Erweiterung, dass Module auch abstrakte Typen als Ein- oder Ausgabe haben dürfen, ergibt sich die Notwendigkeit, dass auch die Datenflussanalyse für die Verifikation und Starttypenbestimmung damit umgehen kann. Interessant wäre hier zu recherchieren, ob es bereits Ansätze einer solchen semantischen Datenflussanalyse gibt, bei der die Daten durch Taxonomien weiter strukturiert sind.

A Anhänge

A.1 Modellierung eines Wizard mit dem jABC

Um ein einheitliches Erscheinungsbild der GUI-Elemente zu erzeugen, aber auch um eine einzige Instanz eines Frames wiederverwenden zu können (damit Anwenderaktionen wie Größenveränderung und Verschieben erhalten bleiben), wurde das in Abschnitt 5.1 „Wizard“ ab Seite 41 beschriebene Konzept für die PROPHETS-SIB-Bibliothek entwickelt. Dabei ist die Idee entstanden, die Buttons des Wizards als mögliche Branches eines Wizard-SIBs heranzuziehen. Die Abfolge der vom Anwender gesehenen Wizard-Seiten lässt sich damit vom Syntheseprozessdesigner im Modell festlegen. Zwischen einzelnen Wizard-SIBs können zusätzlich interaktionsfreie SIBs modelliert werden.

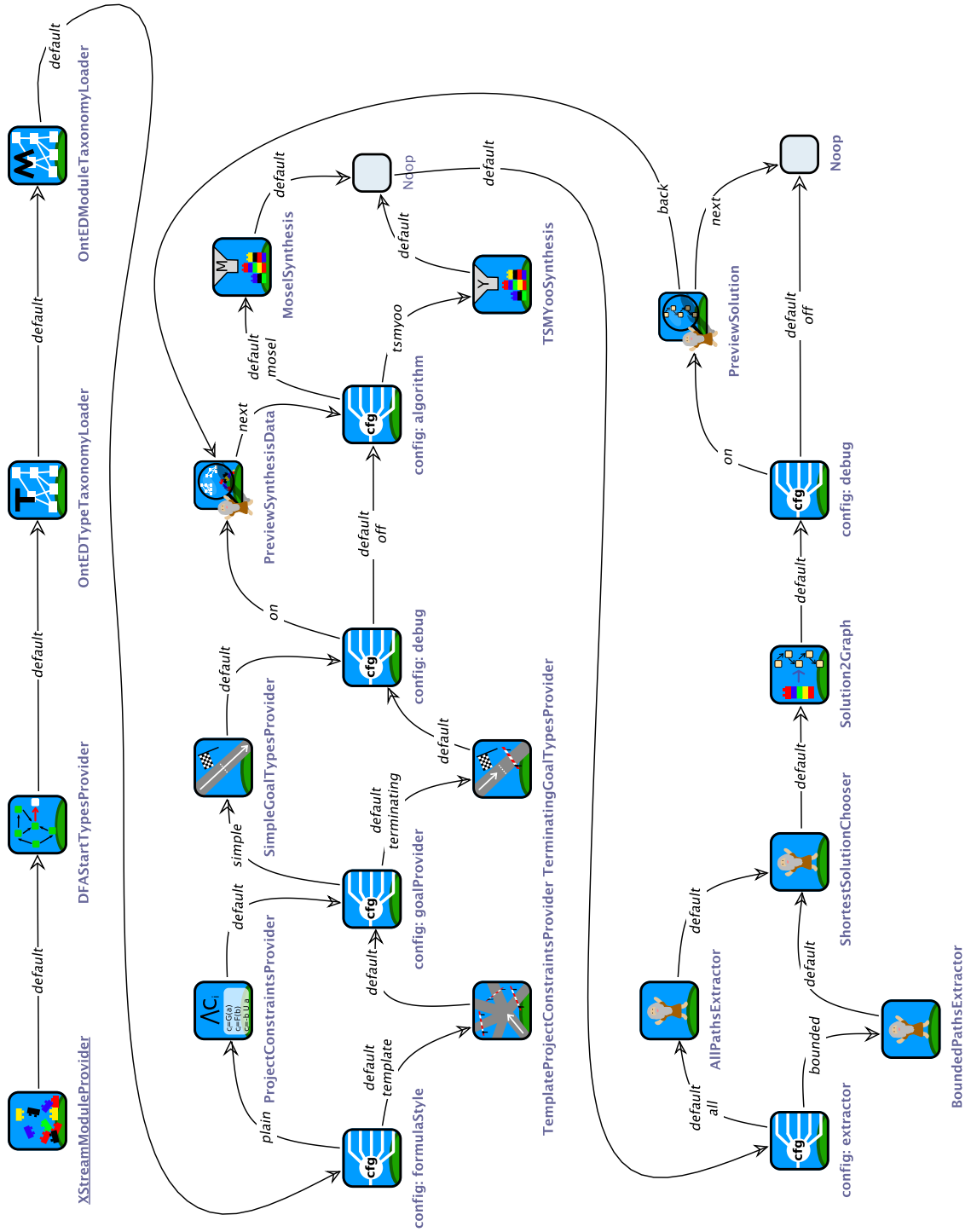
Daraus ergibt sich aber die Fragestellung, wie mit bereits erstellten Daten verfahren werden soll, wenn der Anwender auf „Zurück“ klickt. Er erwartet sicherlich, dass seine Eingaben rückgängig gemacht werden, um alternative Pfade nicht zu beeinflussen. Verwendet man hierfür einen Undo-Manager, der Veränderungen auf dem `ProphetsDataBean` rückgängig machen kann, dann müsste ermittelt werden, wie viele Schritte rückgängig gemacht werden müssen, da eine „back“-Kante durchaus mehrere SIBs zurückspringen kann. Insbesondere tritt dies ein, wenn zwischen zwei Wizard-SIBs interaktionslose SIBs vorhanden sind. Ein „Back“ zu einem dieser SIBs würde den Anwender nach dessen erneuter Ausführung wieder zu genau dem Wizard-SIB zurückbringen, wo er bereits auf „Zurück“ geklickt hat. Da aber die Modellstruktur beim Generieren des Codes verloren geht, gibt es keine Möglichkeit dieses zu überwachen.

Eine weiterer Ansatz wäre, dass alle Wizard-SIBs direkt zu Beginn ihrer Ausführung eine identische Kopie des `ProphetsDataBean` erzeugen und diese wiederherstellen, wenn zu ihnen zurückgesprungen wird. Allerdings lässt sich innerhalb eines SIBs nicht entscheiden, ob es über eine „back“-Kante erreicht wurde oder über einen regulären Vorwärtspfad. Dies kann auch nicht davon abhängig gemacht werden, ob bereits für dieses SIB eine Kopie der Daten exisitiert, da somit vorgesehene Schleifen nicht möglich wären. Auch wenn es solche Schleifen in den für PROPHETS im Rahmen dieser Arbeit modellierten Syntheseprozessen nicht gibt, sollte solch eine Möglichkeit nicht von vornherein ausgeschlossen werden. Um diesem Problem zu entgegnen, müssten also alle SIBs zusätzlich den zuletzt gewählten Branch im Ausführungskontext speichern. Ist dies bei Eintritt in ein SIB „back“, dann muss das SIB die Kopie des `ProphetsDataBeans` wiederherstellen. Aber auch diese Lösung funktioniert nur, wenn zwischen dem Wizard-SIB, bei dem der Anwender auf „Zurück“ geklickt hat, und dem SIB, wohin zurückgesprungen werden soll, kein weiteres SIB auf

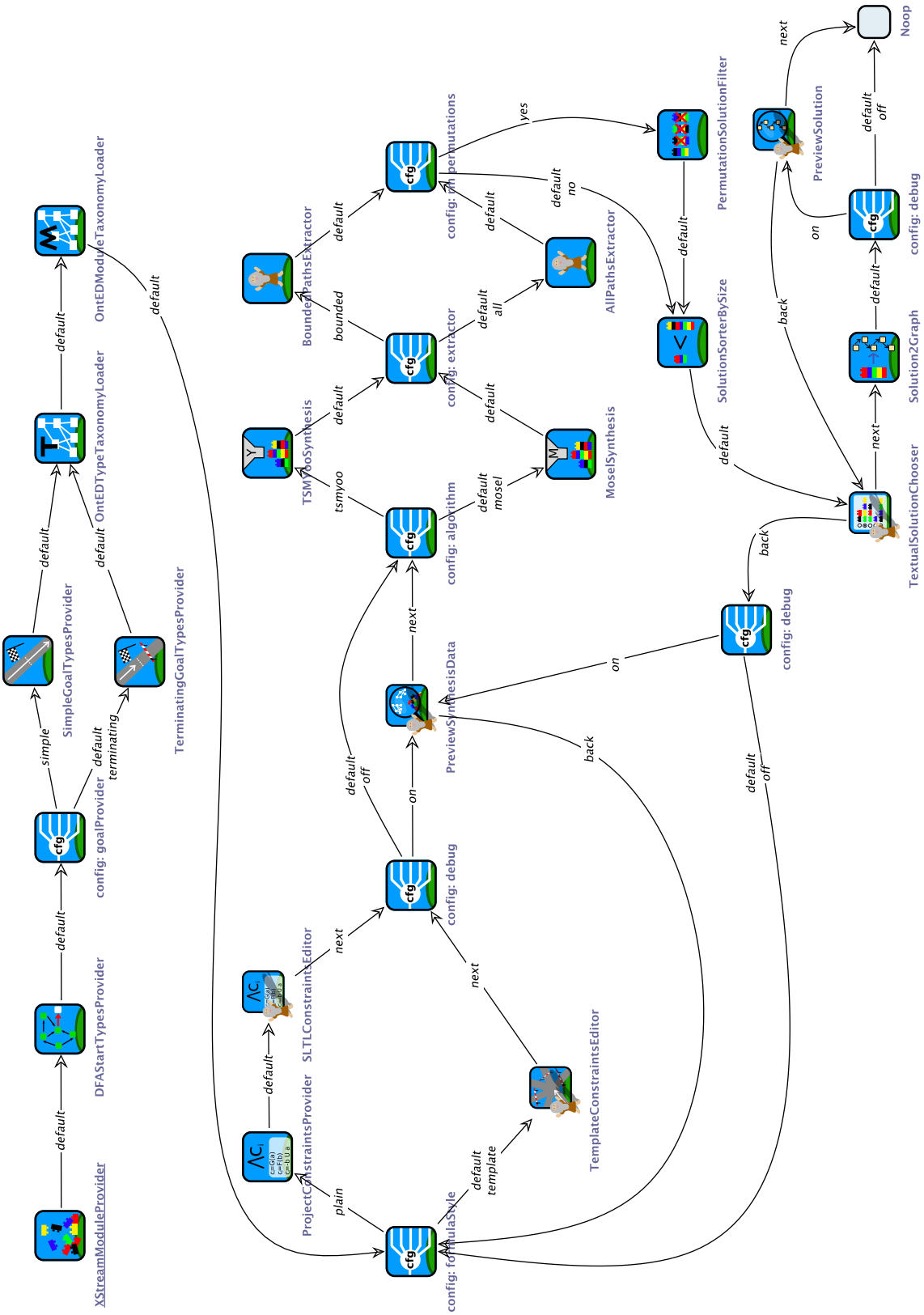
dem „back“-Pfad liegt.

Da keine zufriedenstellende Lösung zu dieser Problematik gefunden werden konnte, ist für PROPHETS keine der vorgestellten Ansätze implementiert worden. In der Regel wird der Anwender keine Probleme feststellen, da die meisten SIBs das Feld im `ProphetsDataBean`, für das sie Daten erzeugen, ohnehin überschreiben statt etwas hinzuzufügen. Dennoch kann es vereinzelt zu Seiteneffekten kommen, die aber zunächst in Kauf genommen werden, da der Implementierungsaufwand zur Behebung in keiner Relation zum Nutzen steht.

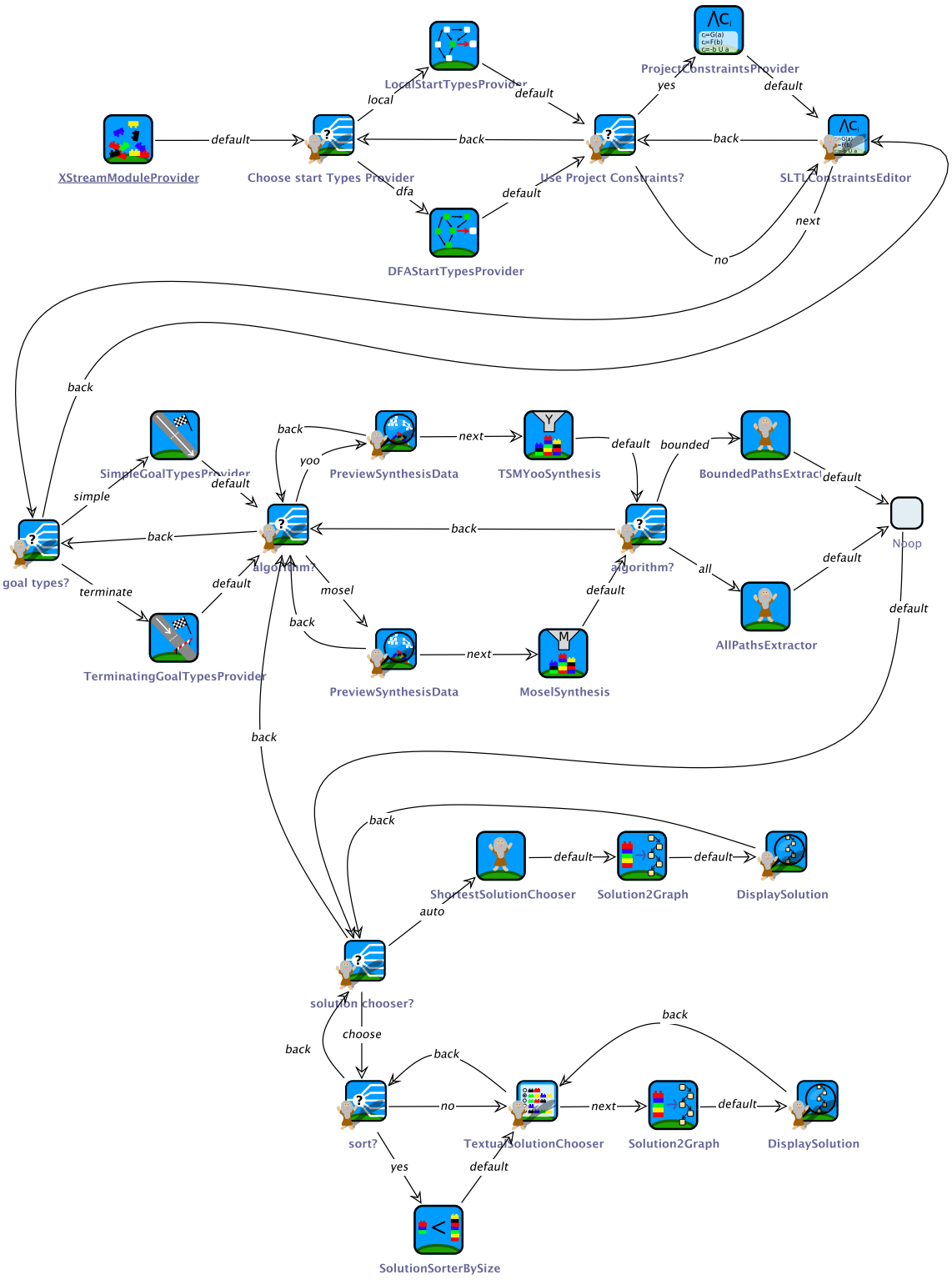
A.2 Syntheseprozesse



Der Syntheseprozess „silent“



Der Syntheseprozess „normal“



Der Syntheseprozess „verbose“

Index

Literaturverzeichnis

[Bak06] Marco Bakera. *Pläne & Spiele. Eine Anwendung spielbasierten Model Checkings*. Diplomarbeit. Aug. 2006. S. S. 7, 38.

[CGP99] Edmund M. Clarke, Orna Grumberg und Doron A. Peled. *Model Checking*. The MIT Press, 1999. S. S. 12.

[Doe06] Markus Doedt. *Erweiterung der jABC-Framework Bibliothek um eine modular anpassbare Ausführungsschicht*. Diplomarbeit. Mai 2006. S. S. 8.

[Doe08] Markus Doedt. *jABC Plugin: Taxonomy Editor*. [Online; Stand 9. September 2009]. 2008. URL: http://jabc.cs.tu-dortmund.de/opencms/en/plugins/taxonomy_editor.html. S. S. 7.

[Eck05] Robert Eckstein. *Creating Wizard Dialogs with Java Swing*. 2005. URL: http://java.sun.com/developer/technicalArticles/GUI/swing/wizard/. S. S. 41.

[EMB09] EMBOSS Developers. *EMBOSS: The Applications (programs)*. [Online; Stand 8. September 2009]. 2009. URL: http://emboss.sourceforge.net/apps/release/6.1/emboss/apps/. S. S. 53.

[Hes02] Wolfgang Hesse. *Ontologie(n)*. 2002. URL: http://www.gi-ev.de/no_cache/service/informatiklexikon/informatiklexikon-detailansicht/meldung/ontologien-57/. S. S. 24.

[JMS08] Sven Jörges, Tiziana Margaria und Bernhard Steffen. "Genesys: service-oriented construction of property conform code generators". In: *ISSE* 4.4 (2008), S. 361–384. S. S. 8.

[Jö09] Sven Jörges. *Genesys Code Generation Framework*. [Online; Stand 24. August 2009]. 2009. URL: http://jabc.cs.tu-dortmund.de/genesys/. S. S. 8.

[KM05] Martin Kempa und Zoltán Ádám Mann. *Model Driven Architecture*. 2005. URL: http://www.gi-ev.de/no_cache/service/informatiklexikon/informatiklexikon-detailansicht/meldung/model-driven-architecture-117.html. S. S. 3.

[KPH05] Aditya Kalyanpur, Bijan Parsia und James A. Hendler. "A Tool for Working with Web Ontologies". In: *Int. J. Semantic Web Inf. Syst.* 1.1 (2005), S. 36–49. S. S. 37.

[Leh09] Lehrstuhl 5 für Programmiersysteme, TU Dortmund. *jABC SIBs*. [Online; Stand 5. September 2009]. 2009. URL: http://jabc.cs.tu-dortmund.de/sib/. S. S. 25.

[Mar+09] Tiziana Margaria u. a. *Synthesizing Semantic Web Service Compositions with jMosel and Golog*. 2009. S. S. 14.

[May09] Christian May. *Entwicklung einer Bibliothek zur service-orientierten Modellierung von Ontologien*. Diplomarbeit. Feb. 2009. S. S. 24.

[MH69] John McCarthy und Patrick J. Hayes. "Some Philosophical Problems from the Standpoint of Artificial Intelligence". In: *Machine Intelligence 4*. Hg. von B. Meltzer und D. Michie. reprinted in McC90. Edinburgh University Press, 1969, S. 463–502. S. S. 10.

[MS04] Tiziana Margaria und Bernhard Steffen. "Lightweight coarse-grained coordination: a scalable system-level approach". In: *Int. J. Softw. Tools Technol. Transf.* 5.2 (2004), 107–123. S. S. 1, 3.

[MS08] Tiziana Margaria und Bernhard Steffen. "Agile IT: Thinking in User-Centric Models". In: *ISoLA*. Hg. von Tiziana Margaria und Bernhard Steffen. Bd. 17. Communications in Computer and Information Science. Springer, 2008, S. 490–502. ISBN: 978-3-540-88478-1. S. S. 5.

[MS09] Tiziana Margaria und Bernhard Steffen. "Business Process Modelling in the jABC: The One-Thing Approach". In: *Handbook of Research on Business Process Modeling*. Hg. von Jorge Cardoso und Wil van der Aalst. IGI Global, 2009. Kap. 1. S. S. 5.

[MS97] Tiziana Margaria und Bernhard Steffen. *Coarse-grain Component Based Software Development: The MetaFrame Approach*. 1997. S. S. 5.

[Nag09] Ralf Nagel. "Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive: Von der Anforderungsanalyse zur Realisierung". Diss. Technische Universität Dortmund, 2009. S. S. 4, 7, 8.

[Nau09] Stefan Naujokat. *jETI: Redesign der ETI-Plattform. Eine Client/Server-Architektur für entfernte Ausführung dateibasierter Tools*. Studienarbeit. Feb. 2009. S. S. 9.

[NNH05] Flemming Nielson, Hanne Riis Nielson und Chris Hankin. *Principles of Program Analysis*. 2. Aufl. Springer-Verlag Berlin Heidelberg, 2005. S. S. 15, 16.

[RHS05] Jan-Peter Richter, Harald Haller und Peter Schrey. *Serviceorientierte Architektur*. 2005. URL: http://www.gi-ev.de/no_cache/service/informatiklexikon/informatiklexikon-detailansicht/meldung/serviceorientierte-architektur-118.html. S. S. 1.

[RLB00] P. Rice, I. Longden und A. Bleasby. "EMBOSS: The European Molecular Biology Open Software Suite". In: *Trends in Genetics* 16.6 (2000), S. 276–277. S. S. 53.

[RN04] Stuart Russel und Peter Norvig. *Künstliche Intelligenz. Ein Moderner Ansatz.* 2. Aufl. Pearson Education Deutschland, 2004. S. S. 10, 44, 58.

[SMB97] Bernhard Steffen, Tiziana Margaria und Volker Braun. “The Electronic Tool Integration Platform: Concepts and Design”. In: *STTT* 1.1-2 (1997), S. 9–30. S. S. 1.

[SMF93] Bernhard Steffen, Tiziana Margaria und Burkhard Freitag. *Module Configuration by Minimal Model Construction*. 1993. S. S. 12, 14.

[SMR08] Paul M. Selzer, Richard J. Marhöfer und Andreas Rohwer. *Applied Bioinformatics. An Introduction*. Springer-Verlag Berlin Heidelberg, 2008. S. S. 53.

[Sta09] Stanford Center for Biomedical Informatics Research. *Protégé*. [Online; Stand 24. August 2009]. 2009. URL: http://protege.stanford.edu/. S. S. 37.

[Ste+06] Bernhard Steffen u. a. “Model-Driven Development with the jABC”. In: *Haifa Verification Conference*. Hg. von Eyal Bin, Avi Ziv und Shmuel Ur. Bd. 4383. Lecture Notes in Computer Science. Springer, 2006, S. 92–108. S. S. 4.

[Sun09] Sun Microsystems. *Java SE at a Glance*. [Online; Stand 31. August 2009]. 2009. URL: http://java.sun.com/javase/. S. S. 6.

[SW81] T. F. Smith und M. S. Waterman. “Identification of common molecular subsequences”. In: *J. Mol. Biol.* 147 (1981), S. 195–197. S. S. 54.

[Top05] Christian Topnik. *jMosel: Ein Toolset für monadische Logik 2. Ordnung auf Strings*. Diplomarbeit. Nov. 2005. S. S. 14.

[W3C04] W3C World Wide Web Consortium. *OWL Web Ontology Language Semantics and Abstract Syntax*. 2004. URL: http://www.w3.org/TR/owl-semantics/. S. S. 24.

[Win06] Christian Winkler. *Entwicklung eines jABC-Plugins zum Design von JDBC-kompatiblen Datenbankschemata*. Diplomarbeit. März 2006. S. S. 8.

[XSt09] XStream Development Team. *XStream Website*. [Online; Stand 5. September 2009]. 2009. URL: http://xstream.codehaus.org/. S. S. 34.