Flexible Modellierung von Domänen in ALICA

UNIKASSEL
VERSITÄT

BACHELORARBEIT

dem Fachbereich Elektrotechnik/Informatik
der Universität Kassel

vorgelegt von

Philipp Faßheber

Matrikelnummer: 29201903

Kassel, April 2014

Erstgutachter:	Prof. Dr. Kurt Geihs
Zweitgutachter:	Prof. Dr. Gerd Stumme
Betreuer:	Stephan Opfer, M.Sc.

Plagiatserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig, ohne unerlaubte Hilfe Dritter angefertigt und andere als die in der Bachelorarbeit angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Dritte waren an der inhaltlich-materiellen Erstellung der Bachelorarbeit nicht beteiligt.

Kassel, den 02.04.2014

Philipp Faßheber

Inhaltsverzeichnis

Abbildungsverzeichnis

Kurzfassung

Für die Verhaltensmodellierung von Agenten existieren verschiedene grafische und textuelle Rahmenwerke. ALICA ist ein solches grafisches Rahmenwerk. Um ein Verhalten zu modellieren, verwendet ALICA unter anderem Bedingungen, die an eine konkrete Domäne angepasst sind. Diese Bedingungen werden während der Verhaltensmodellierung formuliert und zur Laufzeit ausgewertet. Ein Werkzeug, durch das die Verhaltensmodellierung mit ALICA stattfindet, ist der Plan Designer. Im Rahmen dieser Arbeit wurde der Plan Designer dahin gehend erweitert, dass der Anwender für die Modellierung von Bedingungen verschiedene Formalismen verwenden kann. Als erster neuer Formalismus wurde die Aussagenlogik hinzugefügt. Durch die Erweiterung ist es in Zukunft einfach möglich weitere, bisher nicht feststehende, Formalismen hinzuzufügen.

Kapitel 1

Einleitung

In der heutigen Zeit sind Roboter dazu in der Lage Aufgaben eigenständig auszuführen. Sie können ihre Umwelt erfassen und auf Veränderungen der Umwelt reagieren. Ein Roboter kann dabei als ein Agent betrachtet werden. Ein Agent ist ein System, welches innerhalb einer Umgebung zu einem autonomen und dynamischen Verhalten fähig ist. Nutzt ein System verschiedene Agenten, so ist es ein Multi-Agenten-System. Diese finden in verschiedenen Domänen Anwendung. Ein Beispiel für ein Multi-Agenten-System sind die Fußballroboter des Teams *Carpe Noctem Cassel* der Universität Kassel. Mit ihren Robotern tritt das Team jährlich bei verschiedenen *RoboCup Wettbewerben* in der *Middle Size*-Klasse an. In dieser Klasse haben die Roboter eine maximale Seitenlänge von 52 cm und treten in Teams, welche aus bis zu fünf Robotern bestehen, gegeneinander an. Während der Auseinandersetzung handeln die Roboter entsprechend der Spezifikation ihres Verhaltens. Diese kann der Entwickler durch verschiedene textuelle und grafische Rahmenwerke formulieren. Textuelle Rahmenwerke, wie *3APL* oder *AgentSpeak(L)* bieten die Möglichkeit, das Verhalten ausschließlich durch Text zu formulieren, welcher einem bestimmten Formalismus entspricht. In grafischen Rahmenwerken wie *XABSL* oder *ALICA* modelliert der Entwickler, mit grafischen Elementen und Beziehungen zwischen diesen Elementen das Verhalten. ALICA - *A Language for Cooperative and Interactive Agents* wurde von Hendrik Skubch entwickelt und ist für beliebige Anwendungsdomänen einsetzbar. So spielt es keine Rolle, ob die Verhaltensmodellierung zum Beispiel für Fußballroboter oder Roboter zur Erkundung unbekannter Gebiete stattfindet. Ein wichtiges Element von ALICA sind Bedingungen, welche das Verhalten eines Roboters mit bestimmen. Diese Bedingungen beinhalten domänenspezifisches Wissen. So kann in der Anwendungsdomäne der Fußball spielenden Roboter eine solche Bedingung lauten: *"Wenn du den Ball siehst, fahre zum Ball"*. Mit dem Plan Designer wurde von Andreas Scharf ein Werkzeug entwickelt, welches die Verhaltensmodellierung mit ALICA ermöglicht. Für die Formulie-

rung der Bedingung existiert kein fester Formalismus, sondern sie erfolgt durch einen beschreibenden Text.

Die Motivation dieser Arbeit ist, die Verwendung verschiedener Formalismen, zur Modellierung von domänenspezifischen Bedingungen im Plan Designer. Dabei soll die bisherige Modellierungsmöglichkeit nicht verloren gehen. Als erster neuer Formalismus für die Modellierung von Bedingungen, soll die Aussagenlogik hinzukommen. Diese Arbeit beschäftigt sich mit der Umsetzung dieser Anforderung. Dafür erhält der Plan Designer eine Pluginschnittstelle. Neue Formalismen zur Beschreibung von Bedingungen können so als Plugins in den Plan Designer integriert werden. Die bisherige Modellierungsmöglichkeit bleibt durch Auslagerung der Funktionalität in ein Plugin erhalten. Die Unterstützung der Aussagenlogik sichert ein neues Plugin.

Der Aufbau der Arbeit ist folgender: Kapitel 2 verdeutlicht die Probleme, welche durch die bisherige Modellierungsmethode von Bedingungen entstehen und zeigt die daraus resultierende Aufgabenstellung. Kapitel 3 geht kurz auf wichtige Grundlagen für die Arbeit ein. Das Kapitel 4 zeigt, wie andere Rahmenwerken zur Verhaltensmodellierung die Formulierung von Bedingungen ermöglichen. Kapitel 5 beschreibt die Implementierung und den Umgang mit der Pluginschnittstelle und den Plugins. Eine Evaluation der Ergebnisse dieser Arbeit bietet Kapitel 6. Abschließend liefert Kapitel 7 eine Zusammenfassung und bietet einen Ausblick auf mögliche zukünftige Arbeiten.

Kapitel 2

Problemstellung

Für die grafische Verhaltensmodellierung spezifiziert ALICA unter anderem Pläne und Bedingungen, die erfüllt sein müssen, damit ein Agent einen Plan ausführt (Ausführliche Behandlung im Abschnitt 3.2). Allgemein betrachtet ist ALICA domänenunabhängig, aber die Beschreibung einer Bedingungen ist nur mit domänenspezifischem Wissen möglich. Die Bedingung, dass der Roboter den Ball besitzen muss, kann zum Beispiel in der Anwendungsdomäne der Fußballroboter nicht ohne domänenspezifisches Wissen modelliert werden. Der Roboter wertet die Bedingungen zur Laufzeit aus und je nachdem ob sie gilt oder nicht, wählt er den Plan aus.
Der Plan Designer, das Werkzeug zum Erstellen von ALICA-Programmen, bietet bereits die Möglichkeit domänenspezifisches Wissen zu integrieren. Für Bedingungen kann der Entwickler einen beliebigen Text verfassen, der die Bedingungen beschreibt. Bezogen auf das bereits genannte Beispiel, könnte die Beschreibung der Bedingung lauten: “Der Roboter hat den Ball”. Dieser Text entspricht keinen Formalismus, sondern wird frei vom Anwender bestimmt.

Für die vollständige Modellierung einer Bedingung ist Entwicklung auf verschiedenen Ebenen notwendig. Der Entwickler wählt im Plan Designer eine Bedingung aus. Für diese Bedingung verfasst er einen beschreibenden Text. Anschließend verwendet er die Quelltextgenerierung des Plan Designers. Diese analysiert die Bedingung und erstellt daraus den Quelltext. In dem Quelltext taucht auch als Kommentar der vom Anwender verfasste Text auf. Mit der Hilfe dieses Textes setzt der Entwickler die Programmierung der Bedingung mit C# von Hand fort.

Zusammenfassend wird das domänenspezifische Wissen also durch einen beliebigen Text integriert, welcher die Bedingung beschreibt. Daraus ergeben sich verschiedene Nachteile für den Entwickler. Da der Text keinen konkreten Normen entsprechen

muss, kann der Plan Designer ihn nicht auf Fehler analysieren. So können sich Fehler durch den kompletten Modellierungsverlauf ziehen und zum Beispiel erst bei der Programmierung auf der C#-Ebene auffallen. Außerdem kann auf diese Weise keine effektive Quelltextgenerierung stattfinden. Wünschenswert ist also, dass die Modellierung von domänenspezifischen Bedingungen auf der Basis von fest definierten Formalismen, wie zum Beispiel der Aussagenlogik oder Prädikatenlogik durch den Anwender geschieht. Auf diese Weise ist es möglich, den Anwender bereits während der Modellierung im Plan Designer zu unterstützen. Außerdem könnte der Plan Designer Bestandteile des Formalismus, wie zum Beispiel aussagenlogische Formeln nutzen, um die Quelltextgenerierung zu erweitern und dem Entwickler so Aufwand bei der späteren C#-Entwicklung zu ersparen. Der Anwender soll dabei nicht auf einen festen Formalismus beschränkt sein, sondern er soll die Wahl zwischen verschiedenen Formalismen haben. Diese Formalismen integrieren Entwickler in den Plan Designer.

2.1 Aufgabenstellung

Aus der Problemstellung ergibt sich die folgende Aufgabenstellung: Der Plan Designer soll dem Entwickler verschiedene Formalismen anbieten, um eine Bedingung zu modellieren. Dadurch ist es dem Anwender möglich, Domänen auf flexible Art und Weise zu modellieren. Dazu soll der Plan Designer eine Pluginschnittstelle erhalten, welche die Aufnahme von neuen Formalismen ermöglicht. Das heißt für jeden Formalismus erstellt der Entwickler ein zur Pluginschnittstelle passendes Plugin. Die Integration der Plugins soll zur Laufzeit des Plan Designers möglich sein. Dies senkt den Entwicklungsaufwand, da es nicht nötig ist, den Plan Designer neu zu kompilieren oder neu zu starten. Für die Integration muss die Pluginschnittstelle in der Lage sein, die grafische Oberfläche und die Generierung von Quelltext zur Laufzeit zu erweitern. Ebenfalls muss die Schnittstelle eine Möglichkeit zur Persistierung pluginspezifischer Inhalte anbieten. Ein erster neuer Formalismus ist die Aussagenlogik. Dazu ist die Erstellung eines Plugins mit der entsprechenden Funktionalität nötig. Außerdem soll der Plan Designer abwärtskompatibel bleiben, die Modellierung mit einem beliebigen Text also erhalten bleiben. Dafür ist es nötig die zugehörige Funktionalität ebenfalls in ein Plugin auszulagern.

Kapitel 3

Grundlagen

Für das Verständnis der Arbeit ist es nötig, vorher verschiedene Grundlagen zu thematisieren. Dieses Kapitel erläutert Multi-Agenten-Systeme sowie ALICA als Rahmenwerk zur Verhaltensmodellierung von Agenten. Es verdeutlicht relevante Technologien, welche der Plan Designer nutzt und greift abschließend mit der Aussagenlogik und dem Kellerautomaten kurz Inhalte aus der theoretischen Informatik auf.

3.1 Multi-Agenten-Systeme

Bisher existiert keine einheitliche Definition für den Begriff *Agent*. Diese Arbeit nutzt Wooldriges Definition, die einen Agenten als System beschreibt, welches sich in einer bestimmten Umgebung befindet und imstande ist, in dieser Umgebung eigenständig und ohne Fremdeinwirkung Aktionen durchzuführen, um seine vorgegebenen Ziele zu erreichen [4, S. 5 ff.]. Verwendet ein System mehrere Agenten, wird von einem Multi-Agenten-System gesprochen. In diesem System ist eine selbstständige Organisation und Kommunikation notwendig. Nimmt zum Beispiel ein Agent eine spezielle Aufgabe an, muss er die restlichen Agenten darüber informieren. Das verhindert, dass die Agenten die Aufgabe doppelt ausführen. Außerdem kann für das Ausführen einer Aufgabe die Hilfe eines anderen Agenten nötig sein, oder eine andere Aufgabe muss zuvor fertiggestellt sein. Agenten müssen des Weiteren auf ihre Umwelt reagieren können. So kann zum Beispiel ein, mit bestimmter Sensorik ausgestatteter, Agent die anderen Agenten über die aktuelle Situation informieren. Für die Erfüllung dieser Anforderungen muss jeder Agent bestimmte Eigenschaften besitzen. Wooldridge und Jennings definieren die folgenden Eigenschaften [5, S. 116]:

Autonomität: Agenten können eigenständig und ohne fremde Hilfe handeln.

Kommunikationsfähigkeit: Ein Agent besitzt die Fähigkeit, mit anderen Agenten zu kommunizieren.

Reaktionsfähigkeit: Agenten sind dazu in der Lage Änderungen ihrer Umgebung wahrzunehmen und darauf zu reagieren. Die Umgebung kann zum Beispiel die reale Umwelt oder andere Agenten sein.

Proaktivität: Agenten reagieren nicht nur auf Änderungen in ihrer Umwelt, sondern führen aktiv Handlungen aus, um ein Ziel zu erreichen.

Multi-Agenten-Systeme finden in verschiedenen Domänen Anwendung. So sind zum Beispiel die Fußballroboter des Teams Carpe Noctem Cassel ein Multi-Agenten-System. Die Agenten, in dem Fall die Roboter, führen eigenständig verschiedene Handlungen gemeinsam aus, mit dem Ziel das Fußballspiel zu gewinnen. Dabei reagieren sie auf die Position des Balls und die Position der eigenen sowie gegnerischen Roboter und starten aktiv Angriffe, um ein Tor zu schießen.

3.2 A Language for Interactive Cooperative Agents

ALICA ist ein Formalismus zur Verhaltensmodellierung und Kontrolle von kooperativen Robotern. Er wurde von Hendrik Skubch im Rahmen seiner Doktorarbeit entwickelt [6]. Zur Strukturierung und Modellierung von Verhalten verwendet ALICA Hierarchien von endlichen Automaten. Eine Nutzenfunktion weist den Agenten ihre Aufgaben zu. ALICA ist eine domänenunabhängige Sprache und kann deswegen für jede Anwendungsdomäne von Multi-Agenten-Systemen eingesetzt werden. Es spielt keine Rolle, ob die Anwendungsdomäne zum Beispiel Fußballroboter oder Roboter zur Auskundschaftung unbekannter Gebiete ist. Sie bietet jedoch die Möglichkeit, domänenspezifisches Wissen zu integrieren. Im Folgenden wird auf wichtige Elemente von ALICA eingegangen, die zum Verständnis der Arbeit nötig sind.

3.2.1 Endliche Zustandsautomaten

Endliche Zustandsautomaten stellen die unterste Schicht in ALICA dar. Sie dienen zur Beschreibung der Aktionen, die ein Agent ausführen soll. Ein endlicher Zustandsautomat besitzt immer einen Startzustand, eine beliebige Anzahl von Zuständen und eine beliebige Anzahl von Endzuständen. Die Endzustände sind unterteilt in Erfolgs- und Misserfolgszustände. Durchläuft ein Agent einen Automaten und endet im Erfolgszustand, spricht man von erfolgreicher Ausführung des Automaten, beim Enden in

einem Misserfolgszustand von nicht erfolgreicher Ausführung des Automaten. Endzustände besitzen immer eine Nachbedingung. Wenn ein Agent diesen Endzustand erreicht, muss die Nachbedingung gelten. Abbildung 3.1 zeigt beispielhaft einen solchen Automaten für Fußballroboter. Er besteht aus dem Startzustand *Z0*, den drei Zuständen *Z1* - *Z3* und dem Endzustand *Z4*. Die Nachbedingung für den Endzustand ist, dass bei einem Schuss das Tor getroffen wurde.

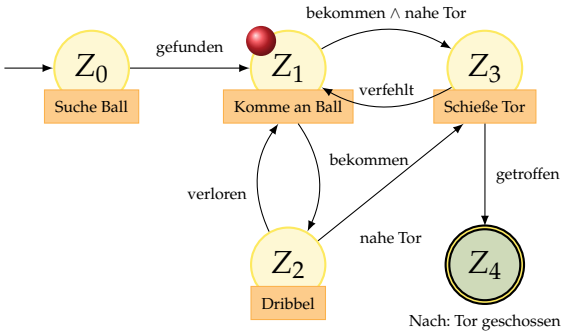


Abbildung 3.1: Endlicher Zustandsautomat in ALICA [1]

3.2.2 Verhalten

Ein Zustand ist in der Regel mit verschiedenen Verhalten verknüpft. In Abbildung 3.1 ist ein Verhalten als Rechteck unter den Zuständen gekennzeichnet. Ein Verhalten beschreibt die auszuführende Aktion, wenn der Agent in diesem Zustand ist und ist ein atomares, domänenspezifisches Element. Typische Verhalten in der Domäne der Fußballroboter sind zum Beispiel zu einem bestimmten Punkt fahren, stehen oder schießen. Für die Modellierung von Verhalten werden in ALICA atomare Verhalten erstellt und dem jeweiligen Zustand zugewiesen. So kann ein Anwender zum Beispiel die Aktion *schießen* modellieren, indem er ein Verhalten *Schieße* erstellt und mit dem gewünschten Zustand verknüpft.

3.2.3 Transitionen

Die verschiedenen Zustände des Automaten sind über Transitionen miteinander verbunden. Diese Transitionen sind mit Bedingungen verknüpft. Ein Agent kann nur den Zustand wechseln, wenn die Bedingung erfüllt ist. In Abbildung 3.1 sind die Zustände *Z0* und *Z1* über eine Transition verbunden. Ein Wechsel von *Z0* in *Z1* kann nur über die Transition geschehen, wenn die Bedingung erfüllt ist, dass der Ball gefunden wurde.

3.2.4 Pläne

Für das Erreichen des übergeordneten Ziels kann es auch nötig sein, dass verschiedene Agenten verschiedene Automaten ausführen müssen. Für Gruppierung von Automaten mit dem gleichen Ziel bietet ALICA Pläne an. Abbildung 3.2 zeigt als Beispiel einen solchen Plan. Die unterschiedlichen Automaten des Plans werden mit einer Aufgabe annotiert. Diese bezeichnen die Aktion, welche der Automat realisiert. Zusätzlich wird mit einer Kardinalität angegeben, wie viele Agenten die Aufgabe ausführen können. So können im Beispiel beliebig viele Agenten die Aufgabe *Verteidigen* ausführen. Im Gegensatz dazu kann die Aufgabe *Angriff* nur ein Agent ausführen. Ein Plan ist ausführbar, wenn seine Vorbedingung erfüllt ist und ausreichend Agenten zur Verfügung stehen. Der Plan *Golden Goal* wird also nur ausgeführt, wenn mindestens zwei Agenten zur Verfügung stehen. Die Ausführungsdauer des Plans ist abhängig von seiner Laufzeitbedingung. Sobald diese nicht mehr erfüllt ist, endet die Ausführung des Plans.

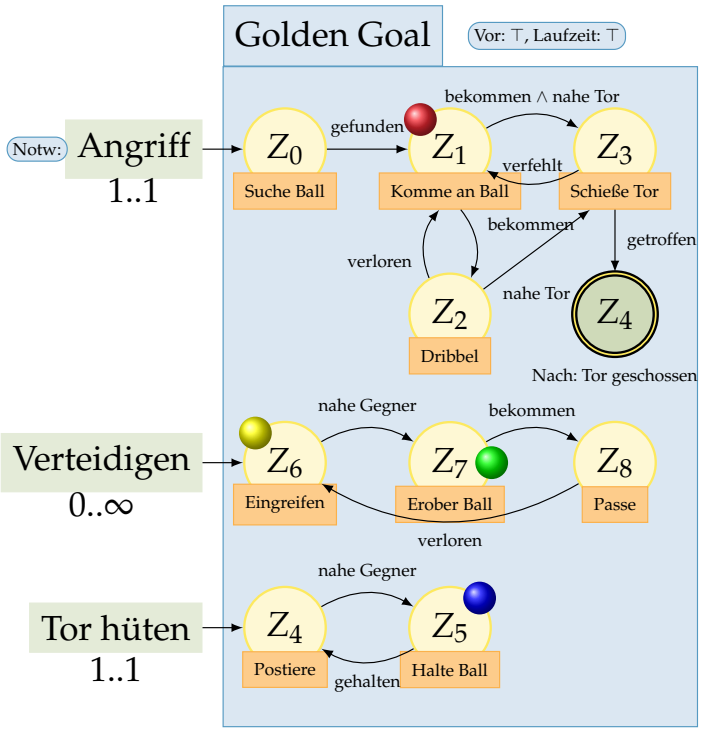


Abbildung 3.2: Grafische Darstellung eines Plans [1]

3.2.5 Bedingungen

Bedingungen werden in ALICA an verschiedenen Stellen verwendet. Es gibt insgesamt drei Arten von Bedingungen.

Vorbedingung: Der Einsatz von Vorbedingungen findet bei Transitionen und Plänen statt. Die erfüllte Vorbedingung einer Transition macht diese passierbar. Für einen Plan bedeutet die erfüllte Vorbedingung, dass dieser ausführbar ist, sofern auch ausreichend Agenten zur Verfügung stehen.

Laufzeitbedingung: Über die Laufzeitbedingung wird gesteuert, ob die Ausführung eines Plans abgebrochen werden soll.

Nachbedingung: Die Nachbedingung ist mit einem Endzustand verknüpft und muss gelten, wenn der Endzustand erreicht wird.

3.3 Plan Designer

Der Plan Designer ist ein grafisches Werkzeug für die Entwicklung von ALICA-Programmen. Die Abbildung 3.3 zeigt ein Teil der grafischen Oberfläche des Plan Designers, mit der es möglich ist, Pläne und ihre Automaten zu modellieren. Die erste Versi-

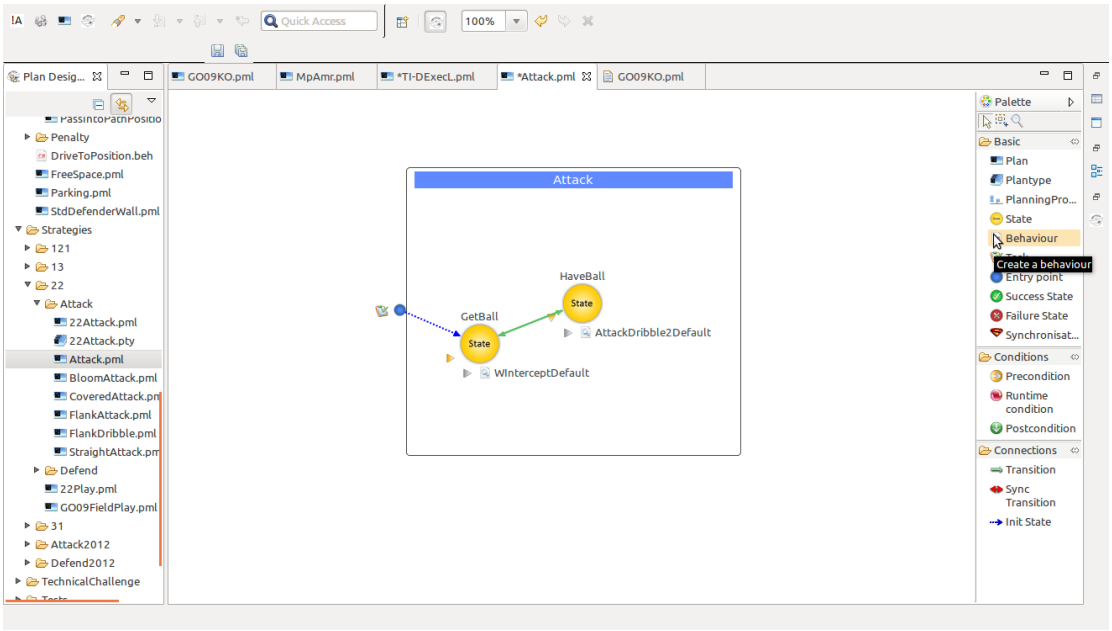


Abbildung 3.3: Oberfläche des Plan Designers

on entwickelte Andreas Scharf 2008 im Rahmen seiner Bachelorarbeit [2]. Seitdem wird der Funktionsumfang des Plan Designers ständig erweitert. Architektonisch baut der Plan Designer auf verschiedenen Technologien auf, welche Abbildung 3.4 vereinfacht auflistet. Die unterste Schicht bildet die OSGi-Spezifikation. Darüber Equinox als Implementierung der Spezifikation. Eine weitere Grundlage ist die Eclipse Rich-Client-Platform, welche durch verschiedene Plugins, wie denen vom Eclipse Modeling

Project erweitert wurde. Dieser Abschnitt erklärt kurz die wichtigsten Technologien des Plan Designers.

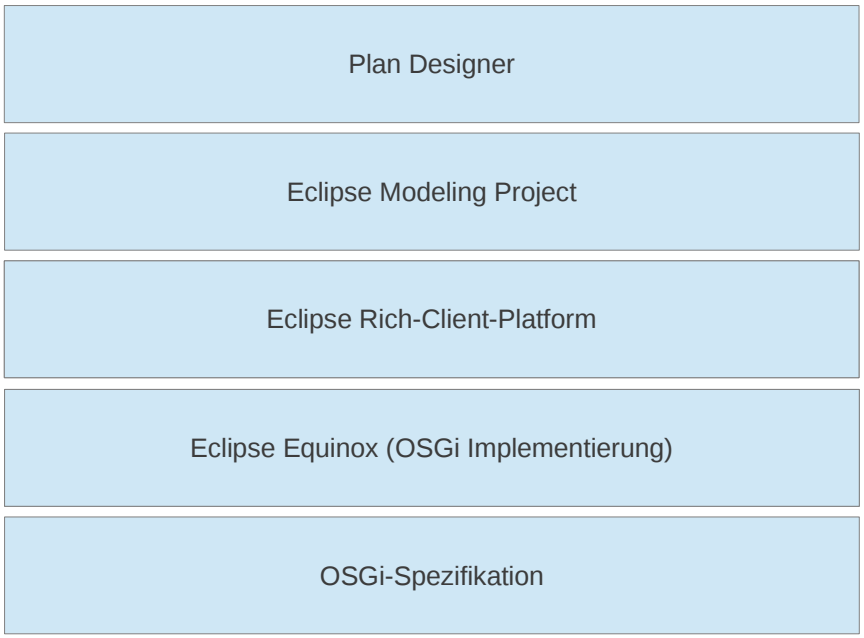


Abbildung 3.4: Architektur des Plan Designers

3.3.1 OSGi-Spezifikation

Die OSGi-Spezifikation wurde erstmalig im Jahr 2000 von der OSGi-Alliance veröffentlicht. Die OSGi-Alliance ist ein Konsortium, bestehend aus verschiedenen großen und kleinen Unternehmen. Mitglieder sind zum Beispiel IBM oder die Deutsche Telekom. Mittlerweile befindet sich die Spezifikation seit 2012 in der Version 5. Sie definiert einen Rahmen für die Entwicklung und Ausführung von Anwendungen. OSGi ist eine serviceorientierte[1] Architektur. Sie besteht aus der *OSGi Service Plattform* an der sich Service-Anwendungen, genannt *Bundles* an- und abmelden. Angemeldete Bundles können gestartet oder gestoppt werden. Diese Vorgänge finden zur Laufzeit der Plattform statt. Es ist kein Neustart der Plattform notwendig. Diese Eigenschaft nennt sich *"Hot Plugging"*. Die detaillierte Beschreibung findet sich in [7].

3.3.2 Eclipse Equinox

Eclipse Equinox [8] ist eine im Jahr 2003 von der Eclipse Foundation entwickelte Referenzimplementierung der OSGi-Spezifikation. Eine zusätzliche Eigenschaft von Eclipse

[1] "Service" engl. für "Dienst"

Equinox ist die Unterstützung von Erweiterungen, genannt *Extensions*. Durch eine Extension ist es möglich, Erweiterungen zu definieren oder anzubieten. Ein Bundle wird in Eclipse Equinox als *"Eclipse Plugin"* oder einfach *"Plugin"* bezeichnet. Die Implementierung besitzt alle nötigen Mechanismen, um Abhängigkeiten zwischen Plugins aufzulösen und die Plugins zur Laufzeit zu verwalten. Die Begriffe *Bundle* und *Eclipse Plugin*, beziehungsweise *Plugin* werden im Folgenden synonym verwendet.

3.3.3 Eclipse Plugin Architektur

Ein Eclipse Plugin ist in Eclipse Equinox das Äquivalent zu einem Bundle in der OSGi-Spezifikation. Als Identifikationsmerkmal erhält das Plugin einen eindeutigen Namen. Eine häufige Konvention für die Namensgebung ist die, welche auch bei Java-Paketen Anwendung findet. Ein Plugin besteht aus einem JAR-Archiv, welches Quelltext, Ressourcen, wie Bilder oder Fremdbibliotheken und verschiedene Dateien zur Konfiguration. Wichtige Konfigurationsdateien sind:

Manifest Diese Datei legt für ein Bundle verschiedene Eigenschaften, wie Name oder Version fest. Zusätzlich definiert das Manifest, welche Quellen das Bundle anderen Bundles zur Verfügung stellt und welche Quellen das Bundle wiederum von anderen Bundles benötigt.

Extension-Points Ein Extension-Point, zu deutsch Erweiterungspunkt, definiert eine Erweiterung. Jeder Extension-Point befindet sich in einer eigens dafür angelegten Datei. Diese beschreiben auf Basis von XML den Extension-Point.

Extensions Extensions geben an, für welche Extension-Points Funktionalität hinzugefügt wurde. Dies können grafische Elemente oder nur Quelltext sein. Zum Start der Anwendung analysiert Eclipse Equinox die Extension-Points und Extension und fügt sie entsprechend zusammen.

Das Schaubild in Abbildung 3.5 verdeutlicht die Zusammenhänge zwischen den Dateien. *PluginA* benötigt Quelltext, welchen *PluginB* bereitstellt. Diese Abhängigkeit ist im *Manifest* definiert. Des Weiteren definiert *PluginA* einen Extension-Point, welcher es anderen Plugins ermöglicht einen Button[2] hinzuzufügen. In der Abbildung wird dieser Extension-Point als *Button-Extension-Point* bezeichnet. Die dazu passende Extension ist die *Button-Extension*. Beim Start der Anwendung löst Equinox die in der Manifest definierten Abhängigkeiten auf. Weiterhin fügt Equinox die Extensions mit den Extension-Points zusammen.

[2] "Button" engl. für "Knopf"

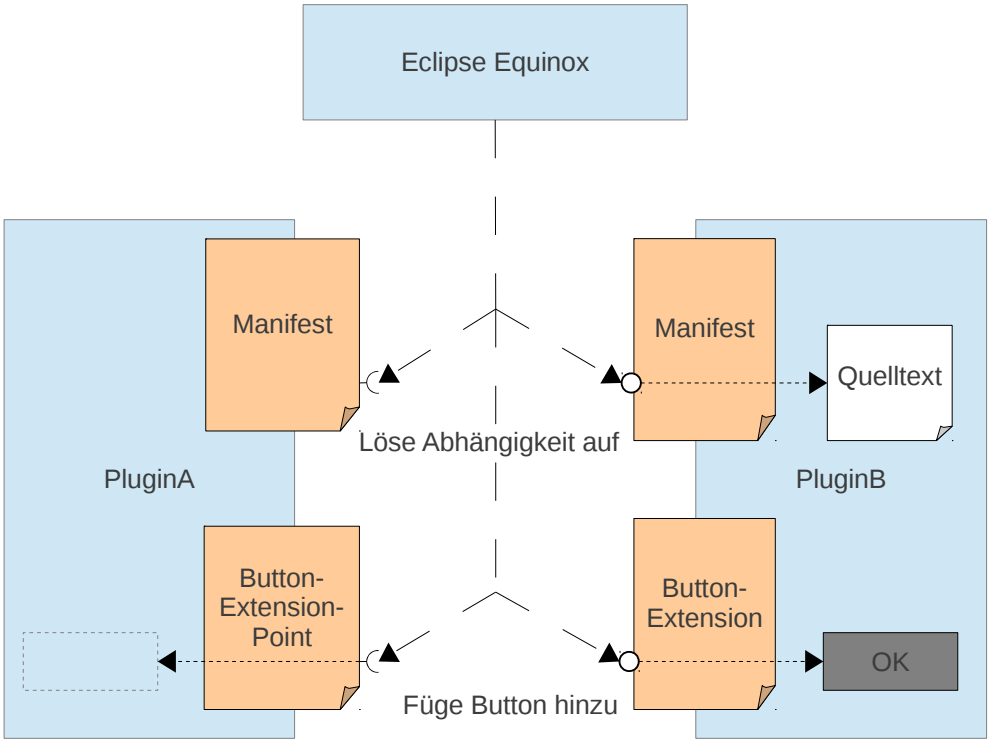


Abbildung 3.5: Darstellung des Pluginmechanismus

3.3.4 Eclipse Rich Client Platform

Die Eclipse Rich Client Platform [9], im folgenden *"Eclispe RCP"* genannt, ist eine Ansammlung diverser Eclipse Plugins, auf deren Basis die Entwicklung von Anwendungen stattfinden kann. So stellen verschiedene Plugins zum Beispiel grafische Bedienelemente, wie Buttons oder Textfelder zur Verfügung. Ebenfalls als Plugin integriert, ist die OSGi-Implementierung. Um eine konkrete Anwendung zu entwickeln, wird die Eclipse RCP um Plugins erweitert. Dabei besteht die Möglichkeit vorhandene Plugins zu nutzen oder eigene zu entwickeln. Auf diese Art und Weise ist es möglich, die Eclipse RCP auf verschiedene Anwendungsdomänen zuzuschneiden. Ein populäres Beispiel ist die *Eclipse Platform* eine integrierte Entwicklungsumgebung für diverse Programmiersprachen. Die Abbildung 3.6 zeigt einen vereinfachten Aufbau der Eclipse Platform und macht deutlich, um welche Plugins die Eclipse RCP erweitert wurde. Hinzugefügte Plugins sind zum Beispiel jene, zum Debuggen oder Zusammenarbeiten mit anderen Entwicklern.

3.3.5 Eclipse Modeling Framework

In der Softwareentwicklung kommen zur Beschreibung und Verarbeitung von Daten einer Anwendungsdomäne häufig Datenmodelle zum Einsatz. Das *Eclipse Modeling*

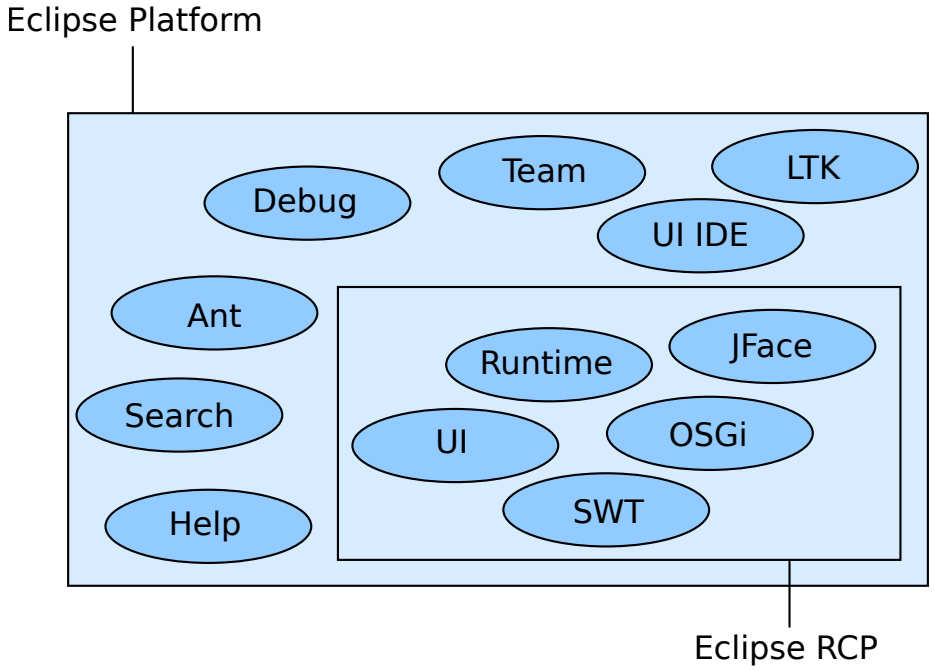


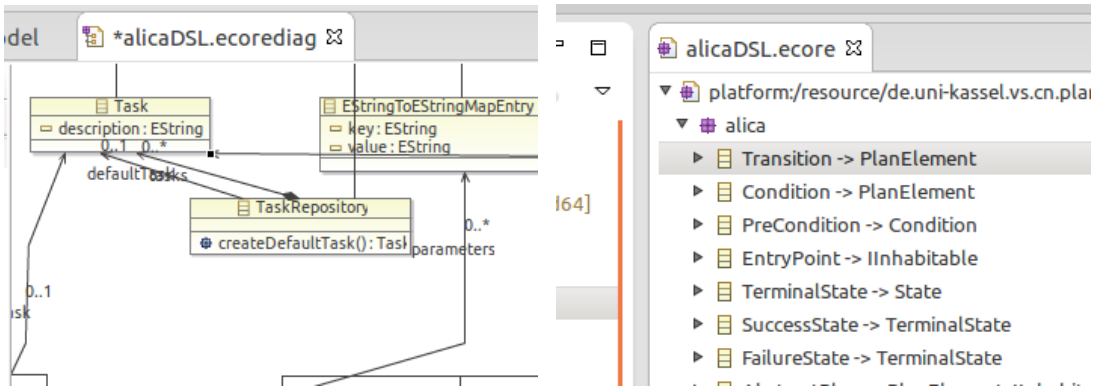
Abbildung 3.6: Vereinfachter Aufbau der Eclipse Platform [2]

Framework [10], im Folgenden *EMF* genannt, bietet eine Möglichkeit zum Erstellen und Bearbeiten von Datenmodellen innerhalb der Eclipse Platform. Zusätzlich ist es mit EMF möglich, aus dem Datenmodell Quelltext zu generieren. EMF ist Teil des Eclipse Modeling Projects[3]

EMF macht eine Unterscheidung zwischen dem eigentlichen Modell und dem sogenannten Meta-Modell. Das Meta-Modell beschreibt die Struktur und Eigenschaften des Modells, d.h. es definiert, welche Attribute die Elemente besitzen und wie deren Beziehungen untereinander sind. Zu unterscheiden sind zwei verschiedene Meta-Modelle. Der sogenannte *Ecore* und das *Genmodel*. Der Ecore definiert Struktur und Eigenschaften der Elemente. Dies kann über verschiedene Methodiken geschehen. EMF bietet einen diagrammatischen und einen baumbasierten Editor zum Erstellen des Ecores an. Abbildung 3.7a zeigt den diagrammatischen Editor, welcher wie ein typischer Editor für UML-Klassendiagramme funktioniert. Die Definition von Elementen und deren Beziehungen erfolgt dort über grafische Elemente, wie Linien und Rechtecke. Beim baumbasierten Editor, welchen Abbildung 3.7b zeigt, findet die Definition durch diverse Eingabefelder statt.

Zusätzlich zu den Editoren gibt es die Möglichkeit verschiedene andere Modelldefinitionen zu importieren und daraus ein *Ecore* zu generieren. Unterstützt werden die

[3]Sammlung von Rahmenwerken und Werkzeugen für die modellgetriebene Softwareentwicklung



(a) Diagrammbasierter Editor **(b)** Baumbasierter Editor

Abbildung 3.7: Diagrammatischer und baumbasierter Editor zum Erstellen des Ecores

folgenden Definitionen:

- Java Interfaces
- XML
- UML

Das *Genmodel* basiert auf dem Ecore. Es beinhaltet alle nötigen Informationen, die für die Generierung des Quelltextes auf Basis des Meta-Modells notwendig sind.
Mit der Nutzung von *EMF* ergeben sich für die Entwicklung einige Vorteile. Änderungen am Modell sind durch Neugenerierung sofort im Quelltext verfügbar. Durch den grafischen Editor erhält der Entwickler einen guten Überblick über die einzelnen Elemente des Systems und deren Beziehungen. EMF bietet außerdem eine Implementierung des *Kommando*-Entwurfmusters an und ermöglicht so auf einfache Art und Weise Änderungen am Modell wieder aufzuheben. Des Weiteren verfügt EMF über einen Mechanismus zur Serialisierung und Persistierung. Dieser ermöglicht das Serialisieren und somit Persistieren von Objektstrukturen auf der Basis von XML oder XMI.

3.3.6 Open Architecture Ware

Open Architecture Ware [11] war ursprünglich ein eigenständiges Projekt, dessen Zielstellung Modelltransformation und die Generierung von Quelltext auf Basis eines Modells war. Seit 2009 gehören die Komponenten von Open Architecture Ware jedoch zum *Eclipse Modeling Project*. Die Generierung von Quelltext ist templatebasiert[4], das heißt, in einem Template wird über Ausdrücke festgelegt, wie der Quelltext für ein Element des Modells auszusehen hat. Dabei können Templates auf Ausdrücke anderer

[4] "Template" engl. für "Vorlage"

Templates zurückgreifen. Das Rahmenwerk lässt sich mit EMF kombinieren. Innerhalb eines Templates ist dadurch der Zugriff auf Objekte und deren Eigenschaften möglich.

Zusätzlich unterstützt Open Architecture Ware den Umgang mit sogenannten Aspekten. Mit Aspekten ist es möglich die Ausdrücke der Templates um Funktionalität zu erweitern, ohne die entsprechenden Templates selbst zu verändern. Abbildung 3.8 zeigt

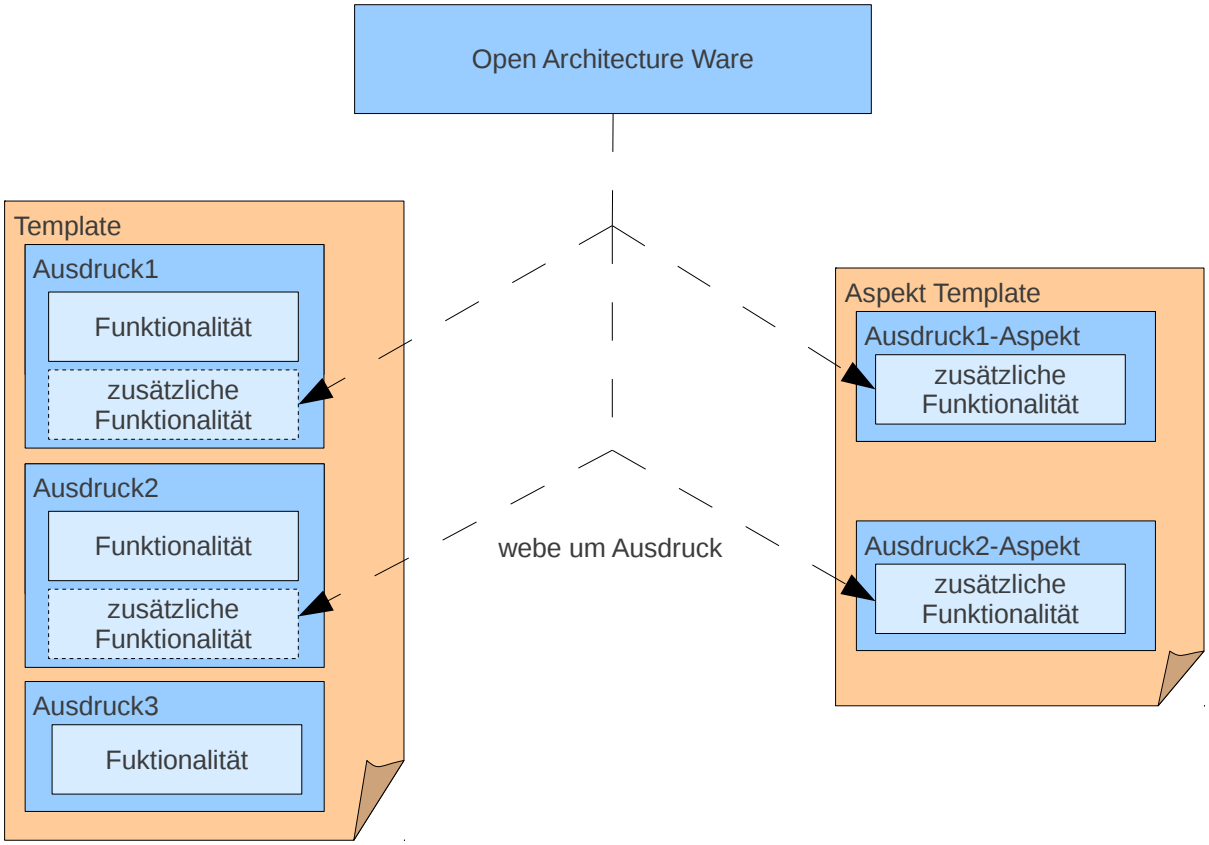


Abbildung 3.8: Funktionsweise von Aspekten in Open Architecture Ware

die Funktionsweise von Aspekten. Zu erkennen ist, dass die Aspekte in einem zusätzlichen Template definiert sind. Innerhalb der Aspekte befindet sich die zusätzliche Funktionalität zur Quelltextgenerierung. Beim Auslösen der Quelltextgenerierung überprüfen entsprechende Komponenten von Open Architecture Ware, für welche Ausdrücke Aspekte vorhanden sind. Anschließend wird die Funktionalität der Aspekte dem Ausdruck hinzugefügt. Dieser Vorgang wird als *weben* bezeichnet. In der Abbildung werden die Aspekte *Ausdruck1-Aspekt* und *Ausdruck2-Aspekt* um die Ausdrücke *Ausdruck1* und *Ausdruck2* gewoben. Bei der Generierung von Quelltext auf Basis des abgebildeten Templates führt Open Architecture Ware die Funktionalitäten der Ausdrücke und die der Aspekte aus, sofern diese für bestimmte Ausdrücke definiert wurden. Im abge-

bildeten Beispiel gilt das für die Ausdrücke *Ausdruck1* und *Ausdruck2*. *Ausdruck3* wird durch keinen Aspekt erweitert.

3.4 Aussagenlogik

Wie in der Problemstellung festgelegt, soll mit einem Plugin die Modellierung der Bedingungen mit Hilfe von Aussagenlogik geschehen. Ein zentraler Bestandteil dieser Arbeit ist also Syntax und Semantik der Aussagenlogik. Dieser Abschnitt definiert diese Begriffe im Bezug zur Aussagenlogik und zeigt mit dem Kellerautomat eine Möglichkeit zu entscheiden, ob ein Text der Syntax der Aussagenlogik entspricht oder nicht. Die Grundlage für diesen Abschnitt bilden [12] und [13].

3.4.1 Grundlagen der Aussagenlogik

Die Aussagenlogik untersucht die Verknüpfung von Aussagen mittels der Operatoren *"und"*,*"oder"*, sowie *"nicht"*. Eine Aussage ist ein atomares sprachliches Gebilde wie:

$$\text{A} = \text{“Ein Kreis ist rund”} \tag{3.1}$$

$$\text{B} = \text{“Ein Kreis ist viereckig”} \tag{3.2}$$

Eine Aussage, auch *atomare Formel* genannt, kann wahr oder falsch sein. Aus der Intuition ergibt sich, dass im obigen Beispiel Aussage A wahr und Aussage B falsch ist. Aufgabe der Aussagenlogik ist nun festzulegen, wie sich die Wahrheitswerte der einzelnen Aussagen zu Wahrheitswerten von verknüpften Aussagen fortsetzen lassen. Eine verknüpfte Aussage, z.B. *A und B* ist eine *Formel*. Die Verknüpfung von Formeln muss einer konkreten Syntax entsprechen, z.B. ist die Formel *und B* keine gültige aussagenlogische Formel, da sie nicht konform zur Syntax ist. Anstelle der Schlüsselwörter *"und"*,*"oder"*, *"nicht"* ist die folgende Notation gebräuchlich:

$$\textit{(1) und: } \wedge \textit{ (2) oder: } \vee \textit{ (3) nicht: } \neg \tag{3.3}$$

Nach Schönig ist die Syntax durch die folgenden Regeln definiert [12, S. 14]:

$$\text{Alle atomaren Formeln sind Formeln} \tag{3.4}$$

$$\text{Für alle Formeln } A \text{ und } B \text{ sind } (A \wedge B) \text{ und } (A \vee B) \text{ Formeln} \tag{3.5}$$

$$\text{Für jede Formel } F \text{ ist } \neg F \text{ eine Formel} \tag{3.6}$$

Die Bezeichnung einer Formel lässt sich aus dem verwendeten Operator ableiten. $\neg A$ ist eine *Negation*, $(A \wedge B)$ eine *Konjunktion* und $(A \vee B)$ eine *Disjunktion*. Taucht die Formel A in B auf, so ist A eine *Teilformel* von B. Bei der Verknüpfung von mehreren Teilformeln durch verschiedene Operatoren, leitet sich die Bezeichnung von dem Operator ab, dessen Operation zuletzt durchgeführt wird. So ist zum Beispiel $A \wedge (B \vee C)$ eine Konjunktion.

Die Semantik der Aussagenlogik ergibt sich durch das Hinzufügen von Bedeutungen zu den syntaktischen Elementen. Erst dadurch ist es möglich, die Wahrheitswerte der verknüpften Aussagen zu untersuchen. Erk et. al. definieren diese in [13, S. 37 ff.] wie folgt: Ein *Wahrheitswert* ist ein Wert der Menge *{wahr, falsch}*. Für die Menge der atomaren Formeln $\mathcal{E}$ lässt sich dann die Abbildung $\mathcal{A}\colon \mathcal{E} \to \{wahr, falsch\}$ definieren, welche jeder atomaren Formel einen Wahrheitswert zuweist. Diese Abbildung heißt *Belegung* und lässt sich auf die Menge aller Formeln $\mathcal{F}$ erweitern. Durch die daraus resultierende Abbildung $\mathcal{A}$ zu $\mathcal{A}\colon \mathcal{F} \to \{wahr, falsch\}$ ergibt sich die Semantik der Aussagenlogik [13, S. 38]:

$$\mathcal{A}(\neg F) = wahr \text{ gdw. } \mathcal{A}(F) = falsch \tag{3.7}$$

$$\mathcal{A}(F \vee G) = wahr \text{ gdw. } \mathcal{A}(F) = wahr \text{ und } \mathcal{A}(G) = wahr \tag{3.8}$$

$$\mathcal{A}(F \wedge G) = wahr \text{ gdw. } \mathcal{A}(F) = wahr \text{ oder } \mathcal{A}(G) = wahr \tag{3.9}$$

3.4.2 Kellerautomaten

In der theoretischen Informatik entscheiden Automaten, ob ein Wort zu einer Sprache gehört. Eine Sprache $\mathcal{L}$ ist eine Menge von Wörtern über einem bestimmten Alphabet Σ. Sei zur Verdeutlichung $\Sigma = \{a, b\}$ und $\mathcal{L} = \{w = uu | u \in \Sigma\}$. Ein Beispiel für $w \in \mathcal{L}$ ist $\{aa\}$, eins für $w \notin \mathcal{L}$ wiederum ist $\{ab\}$.

Von solchen *formalen Sprachen* existieren verschiedene Sprachklassen. Es gibt die *regulären Sprachen REG*, die *kontextfreien Sprachen CF*, die *kontextsensitiven Sprachen CS* und die *rekursiv aufzählbaren Sprachen RE*. Jede dieser Sprachklassen besitzen unterschiedliche Eigenschaften, die in [13] verdeutlicht sind. Für diese Arbeit sind die *kontextfreien Sprachen* interessant, denn das ist die Sprachklasse, welcher die Aussagenlogik angehört. Diese Sprachklasse hat den Vorteil, dass sie *entscheidbar* ist. Das bedeutet für jede Sprache $\mathcal{L} \subset CF$ ist entscheidbar, ob $w \in \mathcal{L}$ oder $w \notin \mathcal{L}$ gilt. Für einen beliebigen Text ist also entscheidbar, ob er der Syntax der Aussagenlogik entspricht.

In der Klasse der *kontextfreien Sprachen* kann ein Kellerautomat das Entscheidbarkeitsproblem lösen. Dieser besteht aus Zuständen und einem Keller, auf welchem Elemente abgelegt oder entfernt werden. Für die Entscheidung der Zugehörigkeit eines Wortes zu einer Sprache liest der Automat das Wort ein. Anhand der Haltekonfiguration des Automaten nach Einlesen des Wortes ist erkennbar, ob das Wort zur Sprache gehört oder nicht. Erk et. al definieren in [13, S. 134] einen Kellerautomaten durch folgendes Tupel:

K	Endliche Menge von Zuständen. In diese wechselt der Automat während der Verarbeitung des Wortes.
Σ	Eingabealphabet mit den Symbolen, aus denen ein Wort besteht.
D	Kelleralphabet, welches die Symbole enthält, die der Automat in den Keller legt.
$s_0 \in K$	Startzustand, welcher der Automat zum Einstieg betritt.
$Z_0 \in D$	Das Anfangssymbol im Keller.
$F \subseteq K$	Die Menge der finalen Zustände.
$\Delta \subseteq (K \times (\Sigma \bigcup \{\varepsilon\}) \times D) \times (K \times D^*)$	Übergangsrelation, die den Zustandswechsel definiert. ε ist das Symbol für das leere Wort, also das Wort welches keine Eingabe repräsentiert. D^* ist die Menge aller möglichen Kombinationen aus den Elementen, die D enthält. Sei zum Beispiel $D = \{a\}$, dann ist $D^* = \{a, aa, aaa, aaaa, ...\}$.

Zur Veranschaulichung von Kellerautomaten dient ein Zustandsdiagramm, wie es in Abbildung 3.9 beispielhaft dargestellt ist. Eine Transition ist immer mit dem benötigten Symbol des Wortes und dem obersten Kellersymbol beschriftet. Diese Konfiguration ist nötig, um die Transition zu passieren. Zusätzlich erfolgt die Angabe eines Elementes aus dem Kelleralphabet. Dieses Element fügt der Automat beim passieren der Transition dem Keller hinzugefügt. Befindet sich der Automat in der Abbildung zum Beispiel im Zustand Z_0, ließt ein a und der Keller ist leer (# ist eine häufige Notation für das Anfangssymbol), so wird a dem Keller hinzugefügt und in Z_1 gewechselt.

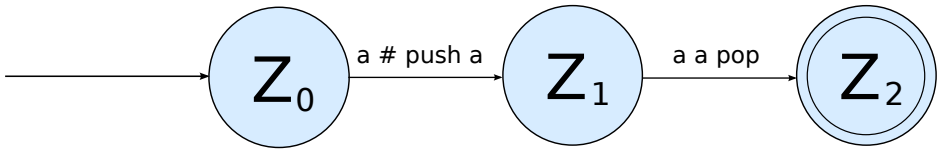


Abbildung 3.9: Zustandsdiagramm eines Kellerautomaten

Sei zum Beispiel $\mathcal{L}$ eine Sprache über einem beliebigen Alphabet Σ und gilt zu entscheiden, ob $w \in \mathcal{L}$ oder $w \notin \mathcal{L}$ gilt. Dazu arbeitet der Kellerautomat nacheinander jedes Symbol, aus dem w besteht ab und wechselt dabei den Zustand entsprechend der Übergangsrelation, beziehungsweise schreibt oder löscht Elemente von seinem Keller. Befindet sich der Automat nach überprüfen des letzten Symbols in einem Endzustand und es befindet sich kein Element in seinem Keller, so gilt $w \in \mathcal{L}$. Erreicht der Automat keinen Endzustand, oder nach dem Durchlauf befindet sich noch mindestens ein Element in seinem Keller, gilt $w \notin \mathcal{L}$.

Kapitel 4

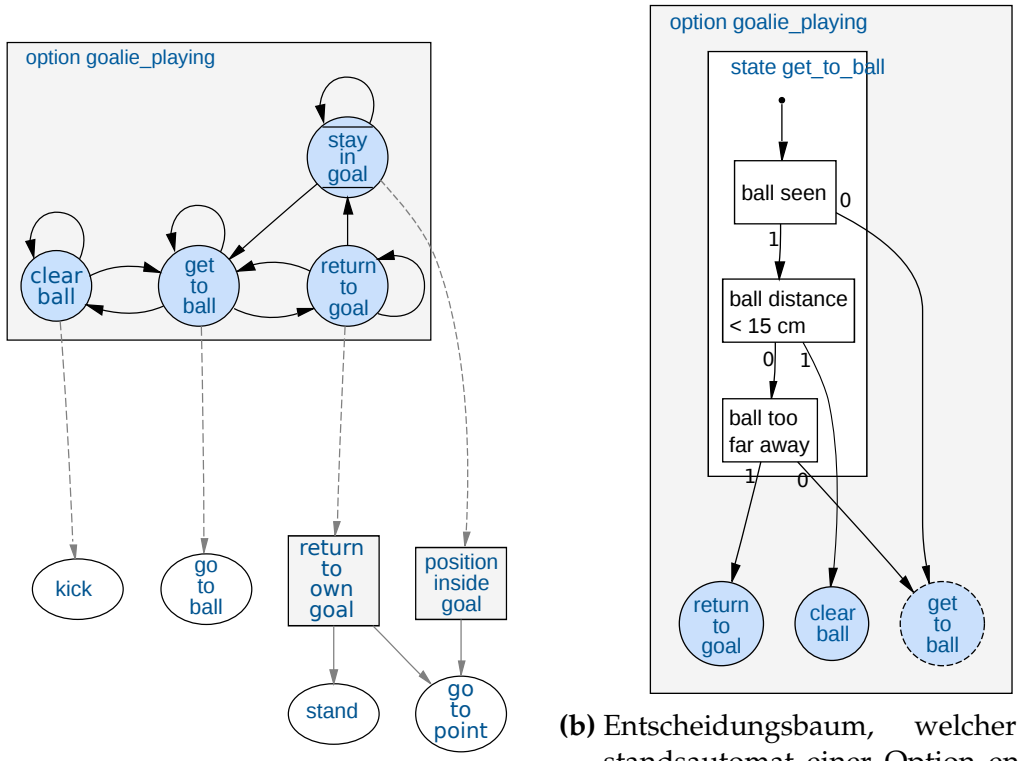
Verwandte Arbeiten

Die Formulierung von domänenspezifischen Bedingungen spielt auch in anderen Formalismen zur Verhaltensmodellierung eine Rolle. Dieses Kapitel fasst kurz die Methodik zur Formulierung von domänenspezifischen Bedingungen in den Formalismen *XABSL* und *3APL* zusammen.

4.1 XABSL

XABSL ist die Abkürzung für *Extensible Agent Behavior Specification Language* und wurde von Loetzsch et. al. zur Verhaltensmodellierung für Agenten entwickelt. Eine ausführliche Beschreibung von XABSL findet sich in [14]. Ähnlich wie ALICA, verwendet dieser Formalismus zur Verhaltensmodellierung hierarchische Zustandsautomaten. Das Verhalten wird durch Optionen, Zustände, Entscheidungsbäume und Basisverhalten beschrieben. Die Optionen werden zusammen mit Basisverhalten in einem Entscheidungsbaum angeordnet. Dieser aktiviert die Optionen oder Basisverhalten. Abbildung 4.1a zeigt die Modellierung einer Option. Eine Option ist ein Zusammenschluss von Verhalten, welche als Zustände modelliert sind. In der Abbildung ist *get to ball*[1] ein solches Verhalten. Verhalten können aus weiteren Optionen und Basisverhalten bestehen. Auch hier bestimmt ein Entscheidungsbaum die Übergänge zwischen den Zuständen. Abbildung 4.1b zeigt einen modellierten Entscheidungsbaum für den Zustand *get to ball*. Der Baum prüft Bedingungen und wählt auf diese Weise den Folgezustand aus. Beispielsweise prüft die Bedingung *ball distance < 15 cm*[2] die Entfernung des Balls. Beträgt die Entfernung mehr als 15 cm, wird die nächste Bedingung überprüft, andernfalls wird in den Zustand *clear ball*[3] gewechselt. Das Modell bildet auf

[1] "get to ball" engl. für "an den Ball kommen"
[2] "ball distance < 15 cm" engl. für "Ballentfernung < 15 cm"
[3] "clear ball" engl. für "Kläre Ball"



(a) Modellierte Option in XABSL

(b) Entscheidungsbaum, welcher Zustandsautomat einer Option entscheidet

Abbildung 4.1: Modellierung von Optionen und Entscheidungsbaum [3]

verschiedene Dateien ab, welche dieses textuell beschreiben. Innerhalb dieser Dateien formuliert der Anwender die Bedingungen aus. Dafür muss er eine Syntax einhalten, die XABSL definiert. Listing 4.1 zeigt beispielhaft, wie die ausformulierte Bedingung *ball distance* < *15 cm* aussehen könnte.

```
state first_state {
  decision {
    if (ball.distance < 15)
      goto clear_ball;
    else
     ...
  }
}
```

Listing 4.1: Beispielhafte Ausformulierung der Bedingung *ball distance* < *15 cm*

4.2 3APL

3APL ist eine an der Universität Utrecht entwickelte, rein textuelle Sprache zur Verhaltensmodellierung von Agenten. Eine detaillierte Beschreibung der Sprache bietet

[15]. Für die Modellierung einer Bedingung folgt 3APL dem BDI-Paradigma. Der Anwender modelliert für einen Agenten unter anderem seine *Überzeugungen*, *Ziele*, *Regeln* und *Fähigkeiten*. Die Formulierung von domänenspezifischen Bedingungen erfolgt zum Beispiel in der Definition der Fähigkeiten. Sei zum Beispiel die Anwendungsdomäne Fußball spielende Roboter. Eine beispielhafte Definition von Fähigkeiten für ein Verhalten *Gehe Zum Ball*, zeigt Listing 4.2.

```
CAPABILITIES {
{NOT roboterHatBall(r1)} FahreZumBall(r1) {roboterHatBall(r1)}
}
```

Listing 4.2: Definition der Fähigkeiten in 3APL

Die Beschreibung der Fähigkeit besteht aus einer Vorbedingung, einer Aktion und einer Nachbedingung. Als Formalismus für die Beschreibung der Bedingungen dient Prolog mit einer eingeschränkten Syntax. Die Vorbedingung muss gelten, damit ein Roboter die Aktion ausführt. Nach der Ausführung, gilt die Nachbedingung. Übertragen auf das Beispiel, wird die Aktion *FahreZumBall* ausgeführt, wenn die Vorbedingung *NOT*[4] *roboterHatBall(r1)* gilt, das heißt der Roboter *r1* nicht den Ball hat. Nachdem der Roboter *r1* die Aktion *FahreZumBall* durchgeführt hat, gilt die Nachbedingung *roboterHatBall*.

[4] "not" engl. für "nicht"

Kapitel 5

Implementierung

Die in der Aufgabenstellung genannten Anforderungen lassen sich für die Implementierung in drei Bereiche kategorisieren. Es ist nötig eine Pluginschnittstelle zu erstellen, welche Plugins verwaltet und in den Plan Designer integriert. Des Weiteren beinhaltet die Aufgabenstellung das Entwickeln von zwei Plugins. Ein Plugin für die Abwärtskompatibilität, welches die Modellierung einer Bedingung durch einen Text ermöglicht und ein Plugin, zur Modellierung mithilfe der Aussagenlogik.

Mit *Eclipse Equinox* verwendet der Plan Designer eine Implementierung der *OSGi-Spezifikation* und ist auf diese Weise mit Eclipse Plugins erweiterbar. Durch diese Eigenschaft sind die in der Aufgabenstellung genannten Anforderungen auf drei neue Bundles aufteilbar. Diese übernehmen jeweils folgende Aufgabe:

Core-Plugin: Dieses Bundle verwaltet Plugins zum Modellieren von Bedingungen. Es ermöglicht das Integrieren solcher Plugins zur Laufzeit des Plan Designers und stellt somit den Kern dar. Im Verlauf der Arbeit wird es mit der englischen Übersetzung *Core-Plugin* bezeichnet.

Default-Plugin: Dieses Plugin beinhaltet die Funktionalität zur Modellierung von Bedingungen durch einen Text und wird als *Default-Plugin*, zu deutsch Standard-Plugin, bezeichnet.

Propositional-Logic-Plugin: Dieses Bundle beinhaltet die Funktionalität, um eine Bedingung durch Aussagenlogik zu modellieren. Auch hier wird die englische Übersetzung *Propositional-Logic-Plugin* verwendet.

Im Rahmen dieser Arbeit sind Änderungen an dem Plugin, welches für die grafische Oberfläche (im Folgenden *UI-Plugin* genannt) verantwortlich ist, entstanden. Dies ermöglicht die Integration von grafischen Oberflächen, die Plugins zum Modellieren ei-

ner Bedingung bereitstellen. Eine Änderung am Plugin zum Generieren von Quelltext (im Folgenden *Code Generator* genannt) ermöglicht, dass Plugins zum Modellieren von Bedingungen festlegen, wie der generierte Quelltext einer Bedingung aufgebaut ist. Durch eine Änderung am Plugin, welches das Modell definiert, haben Plugins zum Modellieren von Bedingungen die Möglichkeit pluginspezifische Inhalte zu speichern. Das Plugin, welches das Modell definiert, wird im Verlauf als Modell-Plugin bezeichnet.

Die nachfolgenden Abschnitte zeigen detailliert die Funktionalität der verschiedenen Eclipse Plugins und gehen auf die Änderung ein. Dabei verdeutlichen sie wichtige Implementierungsdetails. Für die Bezeichnung eines Eclipse Plugins zur Modellierung von Bedingungen wird die englische Übersetzung des Wortes *Bedingung* benutzt. Im Verlauf wird es also *Condition-Plugin* bezeichnet.

5.1 Pluginschnittstelle

Durch Eclipse Equinox existiert zwar bereits eine Möglichkeit, den Plan Designer durch Plugins zu erweitern, jedoch unterstützt Eclipse Equinox dies nicht zur Laufzeit. Diese Aufgabe übernimmt das Core-Plugin, mithilfe der Extensions. Das Core-Plugin ist also die Schnittstelle, um Condition-Plugins in den Plan Designer zu integrieren. Im folgenden Verlauf werden Eigenschaften der Schnittstelle gezeigt und auf verschiedene wichtige Abläufe beim Laden und Initialisieren der Condition-Plugins eingegangen.

5.1.1 Architektur der Pluginschnittstelle

Die Pluginschnittstelle muss diverse Anforderungen erfüllen. Sie muss eine Schnittstelle für Condition-Plugins anbieten und diese zur Laufzeit laden und initialisieren. Das Laden eines Condition-Plugins beinhaltet das Suchen jener Plugins und das Erweitern der Quelltextgenerierung um pluginspezifische Inhalte. Der Initialisierungsvorgang beinhaltet das Überführen einer grafischen Oberfläche samt Funktionalität zum Modellieren von Bedingungen.

Für die Schnittstelle definiert das Core-Plugin einen neuen Extension-Point, zu der jedes Eclipse Plugin, welches als Condition-Plugin fungiert, die zugehörige Extension anbietet. Diese Extension wird im Verlauf als *Condition-Plugin-Extension*, beziehungsweise abgekürzt als *CP-E* bezeichnet. Wie die Abbildung 5.1 zeigt, erwartet der Extension-Point von den Extensions die folgenden Inhalte:

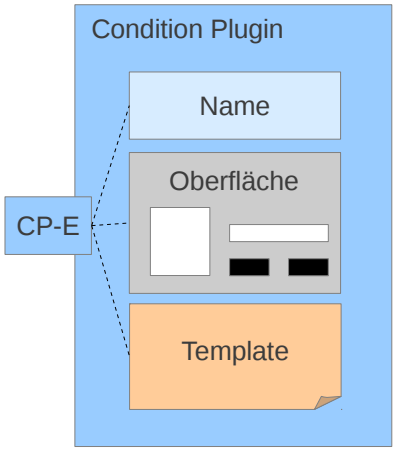


Abbildung 5.1: Architektur der Pluginschnittstelle

Name Ein eindeutiger Name, welcher intern zur Identifikation des Plugins genutzt wird, aber auch um dem Nutzer den Namen des Condition-Plugins anzuzeigen. Daher sollte der Name ein Kompromiss zwischen eindeutiger Kennung und Verständlichkeit sein.

Oberfläche Die grafische Oberfläche und damit verbundene Funktionalitäten, welche zur Modellierung einer Bedingung notwendig sind. Diese integriert das Core-Plugin in den Plan Designer.

Template für Codegenerierung Das Template, welches der Code Generator benötigt, um aus der modellierten Bedingung Quelltext zu generieren.

Ein Eclipse Plugin ist dann ein Condition-Plugin, wenn es die gezeigte Extension besitzt. Nur dann kann es das Core-Plugin erkennen und verarbeiten. Um ein Condition-Plugin zu finden und die Inhalte der Extensions zu verarbeiten und somit das Condition-Plugin zu initialisieren, besitzt das Core-Plugin zwei Komponenten, den *Condition Plugin Loader* und die *Template Engine*. Diese Komponenten übernehmen das Laden und Initialisieren der Condition-Plugins.

5.1.2 Laden der Condition-Plugins

Der Name der Komponente Condition Plugin Loader[1] beschreibt auch deren Aufgabe. Das Laden und die Initialisierung eines Condition-Plugins. Zusätzlich übernimmt sie die Suche von Condition-Plugins. In Abbildung 5.2 ist der erste Teil des Ablaufs zum Laden der Plugins schematisch dargestellt. Der Condition Plugin Loader erwartet alle fertig entwickelten Condition-Plugins in einem Verzeichnis, welches der Benutzer

[1] "loader": von "loading", engl. "laden"

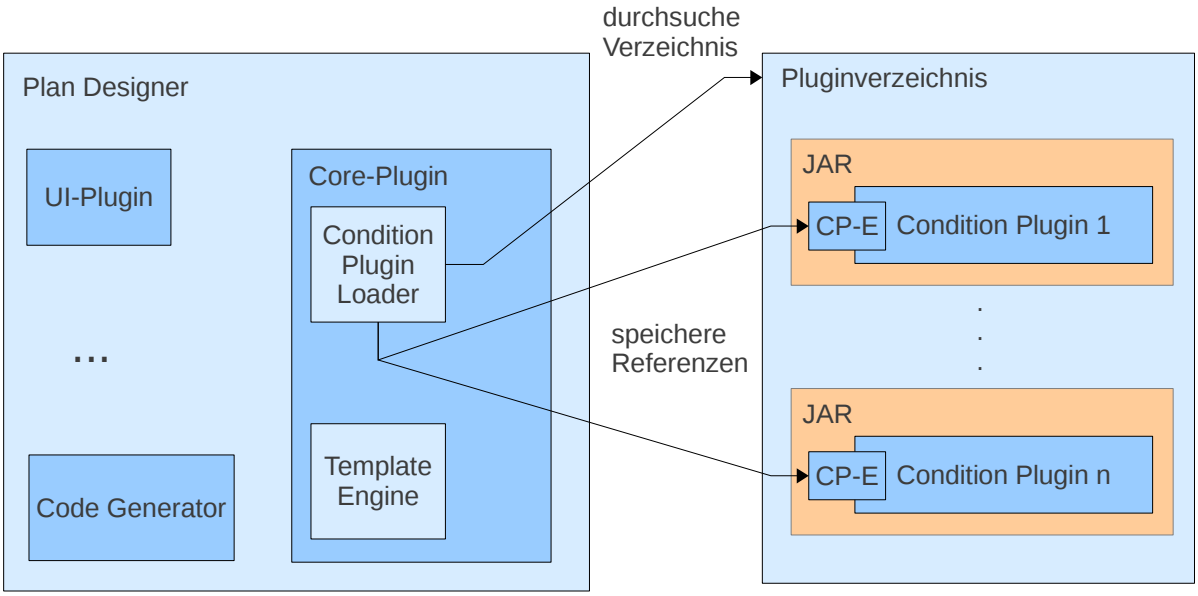


Abbildung 5.2: Laden von Condition-Plugins

festlegt. Da ein Eclipse-Plugin immer ein JAR-Archiv ist, untersucht der Condition Plugin Loader das Verzeichnis auf Dateien von diesem Typ. Erkennt der Condition Plugin Loader ein JAR-Archiv als Eclipse Plugin, welches die CP-E besitzt, nimmt er das Bundle als Condition-Plugin ins System auf. Dazu speichert der Condition Plugin Loader eine Referenz auf das Condition-Plugin, damit er zur Laufzeit darauf zugreifen kann. Adressiert wird das Condition-Plugin mit seinem Namen.

Durch das Finden von Condition-Plugins ist bereits die erste Aufgabe, um Condition-Plugins zur Laufzeit zu laden, abgedeckt. Als zweite Aufgabe bleibt noch die Anpassung der Codegenerierung. Diese Aufgabe erledigt der Condition Plugin Loader mithilfe der Template Engine. Eine Modifikation am Code Generator ermöglicht die Anpassung der Codegenerierung. Die Modifikation verwendet die Eigenschaft, dass der Mechanismus zur Quelltextgenerierung Aspekte unterstützt, welche die Quelltextgenerierung erweitern. Im Code Generator befindet sich ein Template, welches als Schnittstelle für die Templates dient, welche die Condition-Plugins definieren. Für alle in der Schnittstelle aufgelisteten Ausdrücke müssen die Plugin-Templates einen Aspekt anbieten. Wie Abbildung 5.3 zeigt, extrahiert der Condition Plugin Loader das Schnittstellen-Template aus dem Code Generator, sowie die Templates der Plugins. Durch die CP-E ist dem Condition Plugin Loader bekannt, wie er die Templates aus

den Plugins extrahieren kann. Anschließend übergibt der Condition Plugin Loader die Templates an die Template Engine. Deren Aufgabe ist es, zu überprüfen, ob jedes Plugin die benötigten Aspekte anbietet. Das Schnittstellen-Template in Abbildung 5.3 legt fest, dass jedes Plugin-Template die Ausdrücke *Ausdruck1, Ausdruck2* und *Ausdruck3* mit Aspekten erweitern muss. *Condition Plugin 1* erfüllt diese Bedingung und bietet ein gültiges Template an, *Condition Plugin 2* jedoch nicht, da es keinen Aspekt für *Ausdruck3* anbietet.
Sind alle Plugin-Templates gültig, baut die Template Engine ein einziges Template, welches alle Aspekte beinhaltet. Die Aspekte überprüfen, ob das Plugin, durch das sie definiert sind, für die Generierung verantwortlich ist. Daher ist es nötig, dass eine Bedingung weiß, durch welches Condition-Plugin sie modelliert wurde. Den beispielhaften Aufbau des fertig gebauten Templates zeigt der Pseudocode in Listing 5.1.

```
methode1-aspekt{
  if(bedingung.plugin == condition_plugin_1){
    Generiere Code für methode1
  }

  if(bedingung.plugin == condition_plugin_2){
    Generiere Code für methode1
  }
}

methode2-aspekt{
   if(bedingung.plugin == condition_plugin_1){
    Generiere Code für methode2
  }

  if(bedingung.plugin == condition_plugin_2){
    Generiere Code für methode2
  }
}
```

Listing 5.1: Pseudocode für Template mit Aspekten, welche die Verantwortlichkeit des Plugins überprüfen

Das fertig gebaute Template übergibt der Condition Plugin Loader dem Code Generator. Dieser webt zum Zeitpunkt der Quelltextgenerierung die entsprechenden Aspekte um die Ausdrücke der bisherigen Templates. Der ursprüngliche Ansatz, jedes Plugin-Template mit den Aspekten dem Code Generator zu übergeben, erwies sich als falsch. Dies führte dazu, dass die Plugin-Templates sich gegenseitig überschrieben. Daher wird ein Template gebaut, indem überprüft wird, welcher Aspekt verantwortlich ist.

Nach diesem Vorgang ist das Laden der Condition-Plugins abgeschlossen. Das Core-

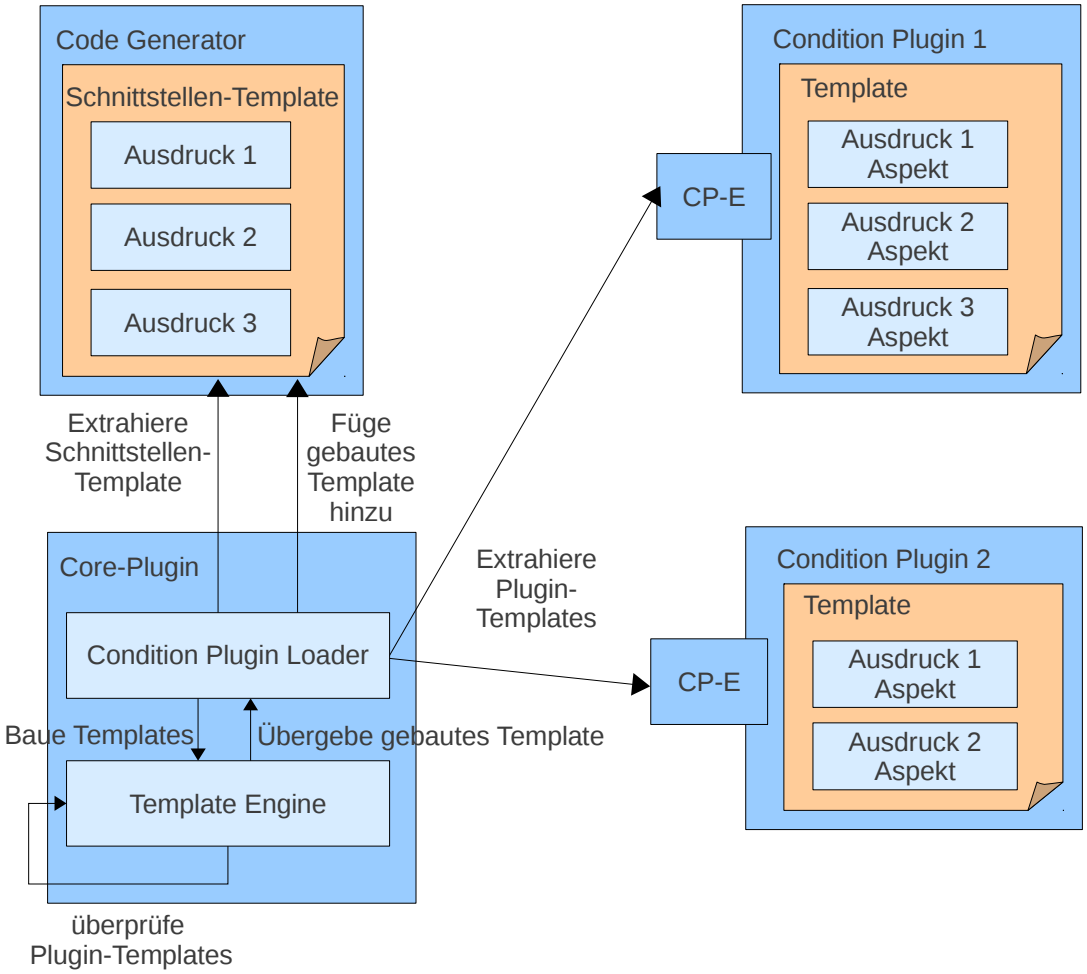


Abbildung 5.3: Darstellung des Bauvorgangs eines Templates aus den Aspekten eines Plugin-Templates

Plugin löst diesen Vorgang zum Start des Plan Designers aus. Zusätzlich hat der Anwender die Möglichkeit den Ladevorgang zur Laufzeit auszulösen. Durch den Ladevorgang ist es möglich, dem Pluginverzeichnis neue Condition-Plugins hinzuzufügen und diese zur Laufzeit in den Plan Designer zu integrieren. Ein Neustart des Plan Designers oder neu Kompilieren ist nicht notwendig. Der Benutzer muss lediglich den Ladevorgang auslösen.

5.1.3 Initialisieren eines Condition-Plugins

Solange der Plan Designer kein konkretes Plugin zur Modellierung von Bedingungen benötigt, initialisiert das Core-Plugin keins. Erst wenn eine Interaktion zwischen Anwender und Plan Designer die Funktionalität eines Condition-Plugins erfordert, initialisiert das Core-Plugin das benötigte Condition-Plugin. Eine solche Interaktion kann nur über die grafische Oberfläche des Plan Designers geschehen. Abschnitt 5.1.1 erwähnt, dass zur Initialisierung das Integrieren einer Benutzeroberfläche gehört. Diese Funktionalität sichert eine Erweiterung im UI-Plugin. Durch die Erweiterung reserviert das UI-Plugin Platz in der Oberfläche des Plan Designers. Zusätzlich ermöglicht sie dem Benutzer zwischen den verschiedenen Condition-Plugins das gewünschte auszuwählen. Das UI-Plugin ersetzt dann zur Laufzeit den reservierten Platz mit der Oberfläche des ausgewählten Condition-Plugins. Abbildung 5.4 zeigt schematisch den

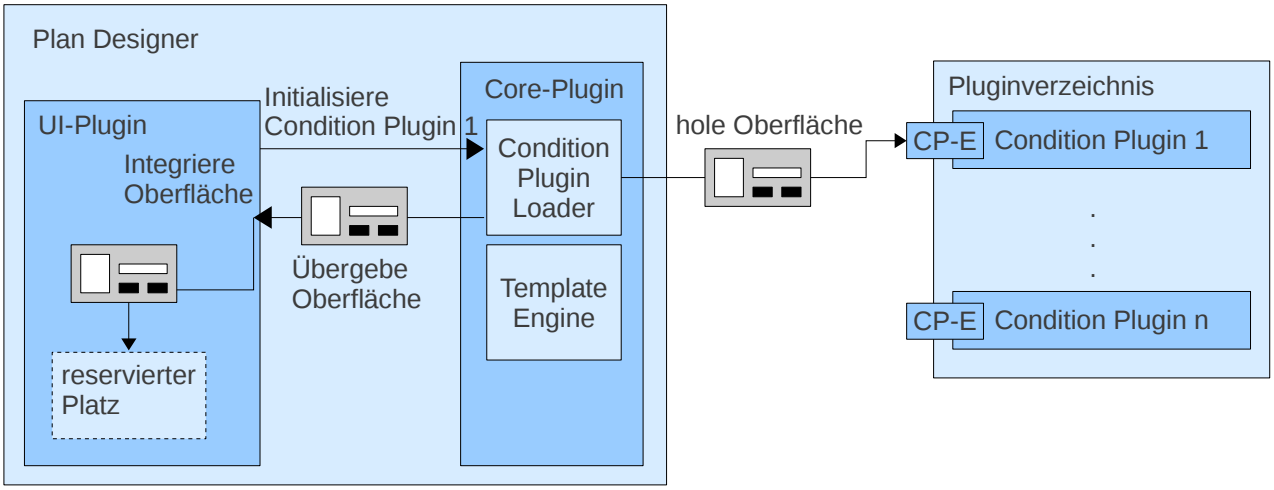


Abbildung 5.4: Initialisierung eines Condition-Plugins

Initialisierungsvorgang eines fiktiven Condition-Plugins, mit Namen *Condition Plugin 1*. Das UI-Plugin teilt dem Condition Plugin Loader mit, dass der Benutzer das Condition Plugin 1 benötigt. Dieser schaut in seinem Speicher nach dem Condition-Plugin und holt sich die Oberfläche. Da das Condition-Plugin die CP-Extension besitzt, ist dem Condition Plugin Loader bekannt, wie er die grafische Oberfläche extrahieren

kann. Diese Oberfläche übergibt der Condition Plugin Loader dem UI-Plugin, welches sie an den reservierten Platz einfügt. Da der Anwender das Pluginverzeichnis zur Laufzeit um neue Plugins erweitert, ist es somit möglich, die Oberfläche samt Funktionalität eines neuen Condition-Plugins, zur Laufzeit zu integrieren.

5.1.4 Speichern pluginspezifischer Inhalte

Das Modell-Plugin des Plan Designers modelliert die ALICA-Inhalte. Das heißt, dort sind zum Beispiel Pläne, Transitionen und Bedingungen sowie deren Beziehungen untereinander auf der Basis von EMF modelliert. Durch eine Erweiterung des Modells weiß eine Bedingung, welches Condition-Plugin sie modelliert. Dadurch haben die Aspekte innerhalb der Plugin-Templates die Möglichkeit zu überprüfen, welches Plugin die Bedingung modelliert.

Die Condition-Plugins müssen ihre Inhalte speichern. Dadurch stehen die Inhalte über die Laufzeit des Plan Designers hinweg zur Verfügung. Zusätzlich muss der Code Generator auf die pluginspezifische Inhalte zugreifen. Durch die Verwendung des Eclipse Modeling Projects ist EMF mit Open Architecture Ware verknüpft und somit hat der Code Generator Zugriff auf diese Inhalte, sofern sie im EMF-Modell auftauchen. Gleichzeitig kann EMF diese serialisieren und persistieren. Aus diesem Grund ist das EMF-Modell durch folgende Lösung erweitert: Da ein Condition-Plugin nur auf Instanzen einer Bedingung arbeitet, enthält die in EMF modellierte Klasse der Bedingung eine Objekt-Struktur, die selbst definierte Schlüssel auf pluginspezifische Inhalte abbildet. Die Schlüssel definiert der Anwender während der Entwicklung des Condition-Plugins. Dafür existiert die folgende Konvention:

<Pluginname>:<Schlüsselname> (5.1)

Ein Beispiel für einen pluginspezifischen Inhalt ist die Formel, die der Anwender mit dem aussagenlogischen Plugin erstellt. EMF serialisiert diese neue Objekt-Struktur automatisch.

5.1.5 Arbeiten mit der Pluginschnittstelle

Dieser Abschnitt zeigt, wie der Anwender die Pluginschnittstelle im Plan Designer nutzen kann. Im Detail wird gezeigt, wie die Pluginschnittstelle konfiguriert wird. Außerdem wird verdeutlicht, wie der Anwender im Plan Designer ein Condition-Plugin auswählt. Abschließend erfolgt eine Darstellung der Integration eines Condition-Plug-

ins zur Laufzeit des Plan Designers.

Konfigurieren der Pluginschnittstelle

Über die Einstellungen des Plan Designers lassen sich verschiedene Werte des Core-Plugins festlegen. Abbildung 5.5 zeigt einen Bildausschnitt der Einstellungen. Zu erkennen ist, dass im Feld *Path to plugin folder* der Anwender das Pluginverzeichnis festlegt. An dem dort angegebenen Ort sucht das Core-Plugin nach Condition-Plugins. Ändert der Anwender das Pluginverzeichnis über die Einstellungen, löst das Core-Plugin den Ladevorgang der Condition-Plugins erneut aus.
In der Drop-Down-Liste[2] *Default plugin* kann der Benutzer das Condition-Plugin auswählen, welches das Core-Plugin für die Bedingungen initialisiert, die bisher durch kein Plugin modelliert wurden. Auf diesen Wert greift das Core-Plugin immer dann zurück, wenn der Anwender eine Bedingung im Plan Designer auswählt, die noch durch kein Condition-Plugin modelliert wurde.

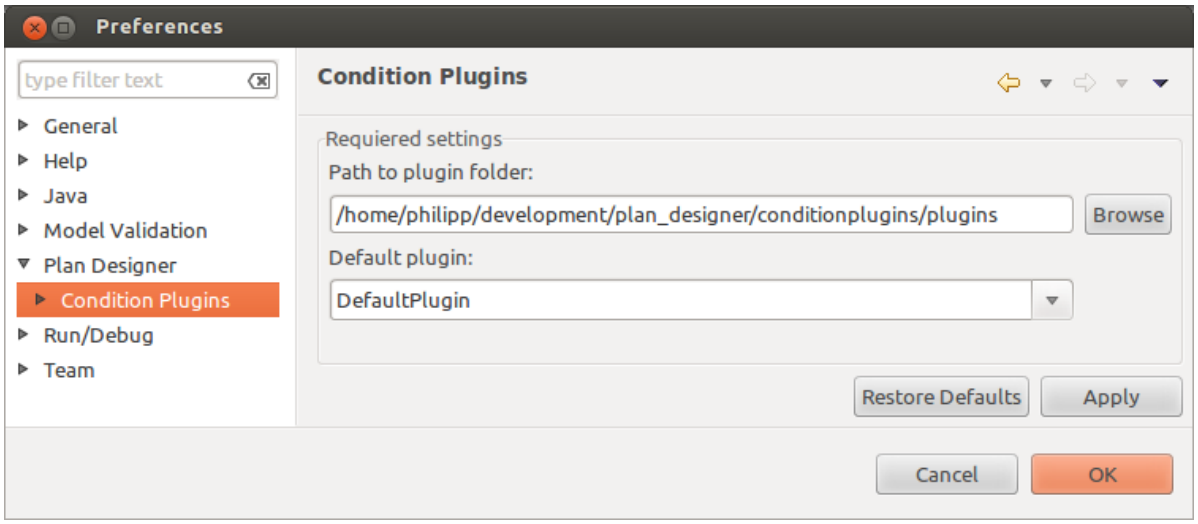


Abbildung 5.5: Oberfläche zum Konfigurieren der Pluginschnittstelle

Wahl eines Condition-Plugins

Wählt der Benutzer eine Bedingung im Editor des Plan Designers aus, zeigt dieser eine Oberfläche an, welche die Auswahl eines Condition-Plugins ermöglicht. Abbildung 5.6 zeigt diese Oberfläche. Über die Drop-Down-Liste *Plugin Settings* kann der Anwender eine Auswahl aus allen Condition-Plugins im Pluginverzeichnis treffen. Nach Auswahl eines Condition-Plugins, initialisiert das Core-Plugin dieses und die Oberfläche des Plugins wird im Abschnitt unterhalb des Menüs angezeigt.

[2] "Drop-Down" engl. für "herunterfallen"

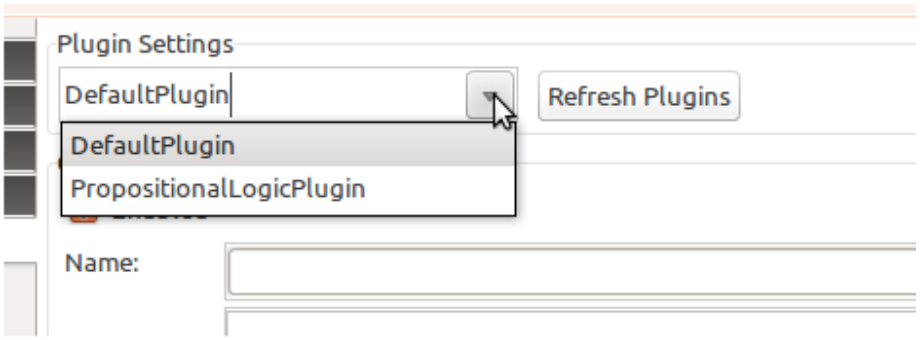


Abbildung 5.6: Oberfläche zum Auswählen eines Condition-Plugins

Integration eines neuen Condition-Plugins

Für die Integration eines neuen Condition-Plugins zur Laufzeit des Plan Designers, legt der Entwickler das Plugin in das festgelegte Pluginverzeichnis und löst anschließend den Ladevorgang der Condition-Plugins aus. Dazu bietet das Core-Plugin ein Symbol in der Werkzeugleiste des Plan Designers an. Abbildung 5.7 zeigt dieses Symbol. Ein Klick auf das markierte Symbol veranlasst das Core-Plugin den Ladevorgang zu starten. Anschließend steht das neue Condition-Plugin im Plan Designer zur Verfügung.

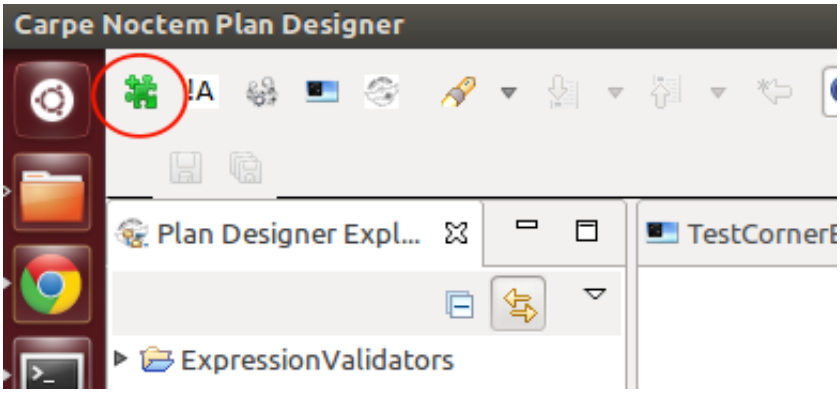


Abbildung 5.7: Symbol in der Werkzeugleiste, welches das Neueinlesen vom Pluginverzeichnis auslöst

5.2 Plugin zum Modellieren aussagenlogischer Formeln

Die Funktionalität zum Modellieren einer Bedingung mithilfe der Aussagenlogik befindet sich in einem eigenen Bundle, dem Propositional-Logic-Plugin. Dieses Bundle verwendet der Anwender zum Modellieren einer Bedingung und ist deshalb als Con-

dition-Plugin implementiert. Somit erkennt es die Pluginschnittstelle und ist dazu in der Lage es zu verwalten.
Das Plugin bietet alle nötigen Inhalte an, damit der Anwender eine Bedingung mithilfe der Aussagenlogik modellieren kann. Modellieren bedeutet hier, dass eine aussagenlogische Formel für die Bedingung verfasst wird. Durch das Plugin ist es dem Anwender möglich, aussagenlogische Formeln zu erstellen und diese später bei der Modellierung zu verwenden. Dazu stellt das Propositional-Logic-Plugin entsprechende Oberflächen zur Verfügung. Diese unterstützen den Nutzer durch verschiedene Besonderheiten, wie Autovervollständigung, oder Syntaxüberprüfung der verfassten Formel. Das Plugin bietet ein Template an, welches die Anforderungen eines Plugin-Templates erfüllt. Das heißt, es besitzt alle Aspekte, die das Schnittstellen-Template vorgibt und diese überprüfen vor der Quelltextgenerierung, ob das aussagenlogische Plugin für die Modellierung der Bedingung verantwortlich ist. Dadurch kann der Code Generator aus der Formel für die Bedingung Quelltext generieren.

Die Abbildung 5.8 fasst vereinfacht die Architektur des Propositional-Logic-Plugins zusammen. Da dieses Plugin ein Condition-Plugin ist, bietet es die Condition-Plugin-Extension an und somit Zugriff auf die Oberfläche zur Modellierung, das Template zur Quellcodegenerierung sowie den Namen des Plugins. Weiterhin besitzt das Plugin ein Modell, welches die interne Verwaltung von Formeln ermöglicht. Für die Verwaltung von Formeln durch den Nutzer bietet es einen Formel Editor an. Zu erkennen ist außerdem, dass das Plugin einen Kellerautomaten beinhaltet. Dieser wird intern für die Überprüfung der Syntax der erstellten Formel verwendet und sichert somit, dass die Formel der Syntax der Aussagenlogik entspricht. Dieser Abschnitt beschreibt die ab-

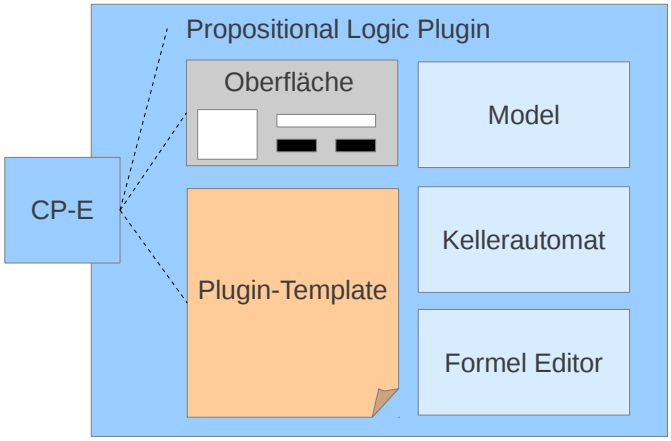


Abbildung 5.8: Vereinfachte Zusammenfassung des Propositional-Logic-Plugins

gebildeten Komponenten des Propositional-Logic-Plugins und verschiedene Aspekte der Integration in den Plan Designer sowie der Quelltextgenerierung. Weiterhin zeigt

der Abschnitt, wie der Anwender eine Formel für eine Bedingung erstellt und veranschaulicht den Umgang mit dem Formel Editor.

5.2.1 Datenmodell

Abbildung 5.9 zeigt das umgesetzte Datenmodell, welches die Organisation von aussagenlogischen Formeln ermöglicht. Das Datenmodell berücksichtigt, dass eine Formel entweder atomar oder verknüpft ist und bietet daher eine Klasse *Formel*, für verknüpfte Formeln und eine davon erbende Klasse *Aussage* für die atomaren Formeln an. Zu erkennen ist, dass eine Formel mit einem Namen bezeichnet wird. Des Weiteren erlaubt die Syntax der Aussagenlogik die Verknüpfung von Formeln zu einer neuen Formel. Im Modell ist dies durch die Assoziation *kinder* realisiert. Das heißt, eine Formel enthält alle Teilformeln, aus der sie besteht. Dadurch ist es möglich, eine Formel rekursiv bis in ihre atomaren Bestandteile zu zerlegen. Für die Organisation von Formeln existieren Vokabulare. Diese fassen Formeln zusammen und ermöglichen die Adressierung einer Formel über ihren Namen. Im Modell ist zu erkennen, dass das Propositional-Logic-Plugin zwei Vokabulare verwaltet. Ein Vokabular, welches ausschließlich Aussagen enthält (im Modell durch die Assoziation *aussagenVokabular* gekennzeichnet) und ein weiteres Vokabular, welches ausschließlich Formeln enthält (im Modell durch die Assoziation *formelVokabular* gekennzeichnet). Durch diese beiden Vokabulare hat das Plugin Zugriff auf alle Formeln.

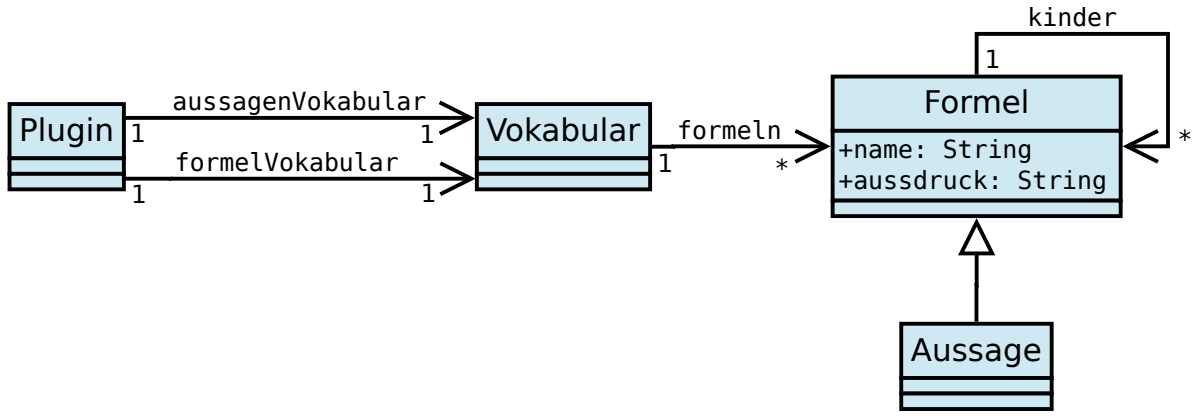


Abbildung 5.9: Datenmodell vom Propositional-Logic-Plugin

5.2.1.1 Persistieren der Vokabulare

Für die Sicherung der Verfügbarkeit von Aussagen und Formeln über die Laufzeit des Plan Designers hinweg, persistiert das Propositional-Logic-Plugin diese. Es löst diesen Vorgang immer aus, wenn der Anwender eine Aussage oder Formel erstellt

beziehungsweise verändert. Als Datenformat für die Persistierung wird das Schlüssel-Wert-Format verwendet. Da das Plugin zwei Vokabulare verwendet, gibt es auch zwei Dateien, welche diese Persistieren. Wie das Plugin Aussagen und Formeln persistiert, zeigen Listing 5.2 und Listing 5.3.

```
HatBall=Roboter hat Ball
ImGegnerischenStrafraum=Roboter befindet sich im gegnerischem Strafraum
FreieSicht=Es befindet sich kein Hindernis in Blickrichtung des Roboters
...
```

Listing 5.2: Auszug eines Aussagenvokabulars

```
KannSchiessen = HatBall & ImGegnerischenStrafraum & FreieSicht
...
```

Listing 5.3: Auszug eines Formelvokabulars

Das Symbol "&" bedeutet, dass die Operanden mit dem und-Operator verknüpft sind. Als Schlüssel dient der Name der Formel. In den Listings ist das zum Beispiel *HatBall* oder *KannSchießen*. Der Wert zu dem Schlüssel entspricht dem Ausdruck der Aussage oder Formel. Der Ausdruck einer atomaren Formel ist die Aussage selber. In Listing 5.2 ist zum Beispiel *Roboter hat Ball* ein solcher Wert. Ein Beispiel für den Wert in Listing 5.3 ist *HatBall* & *ImGegnerischenStrafraum* & *FreieSicht*. Beim Laden des Propositional-Logic-Plugins liest dieses die jeweiligen Dateien ein und erzeugt eine auf dem Datenmodell basierende Objekthierarchie.

5.2.2 Korrektheitsüberprüfung einer Formel

Bevor die dieser Abschnitt auf die Korrektheitsüberprüfung eingeht, soll die Notation der Operatoren erklärt werden. Da die Symbole $\wedge$, $\vee$, $\neg$ in der Regel nicht über die Tastatur erreichbar sind und sie auch nicht jeder Zeichensatz beinhaltet, verwendet das Propositional-Logic-Plugin die folgende Notation:

Symbol	Ersatz Symbol	Bedeutung
$\wedge$	&	und-Verknüpfung von Operanden
$\vee$	|	oder-Verknüpfung von Operanden
$\neg$	!	Negation von Operanden

Für den Anwender ist es nützlich, wenn er beim Arbeiten mit Formeln Rückmeldung über die Korrektheit der Formeln erhält. Daher überprüft das Propositional-Logic-Plugin sowohl beim Modellieren der Bedingung als auch beim Erstellen oder Verändern einer Formel, ob diese korrekt ist und informiert den Anwender über den Status.

Er bekommt Rückmeldung, wenn die Formel korrekt ist und wenn sie fehlerhaft ist. Eine Formel ist in diesem Anwendungsfall korrekt, wenn sie die folgenden Bedingungen erfüllt:

- Die Formel besteht aus bekannten Teilformeln
- Die Formel entspricht der Syntax der Aussagenlogik

Für die Überprüfung der ersten Bedingung zerlegt das Plugin eine Formel in ihre Teilformeln. Anschließend untersucht es die Vokabulare auf Vorhandensein der Teilformeln. Wenn alle Teilformeln vorhanden sind, ist die erste Bedingung erfüllt.

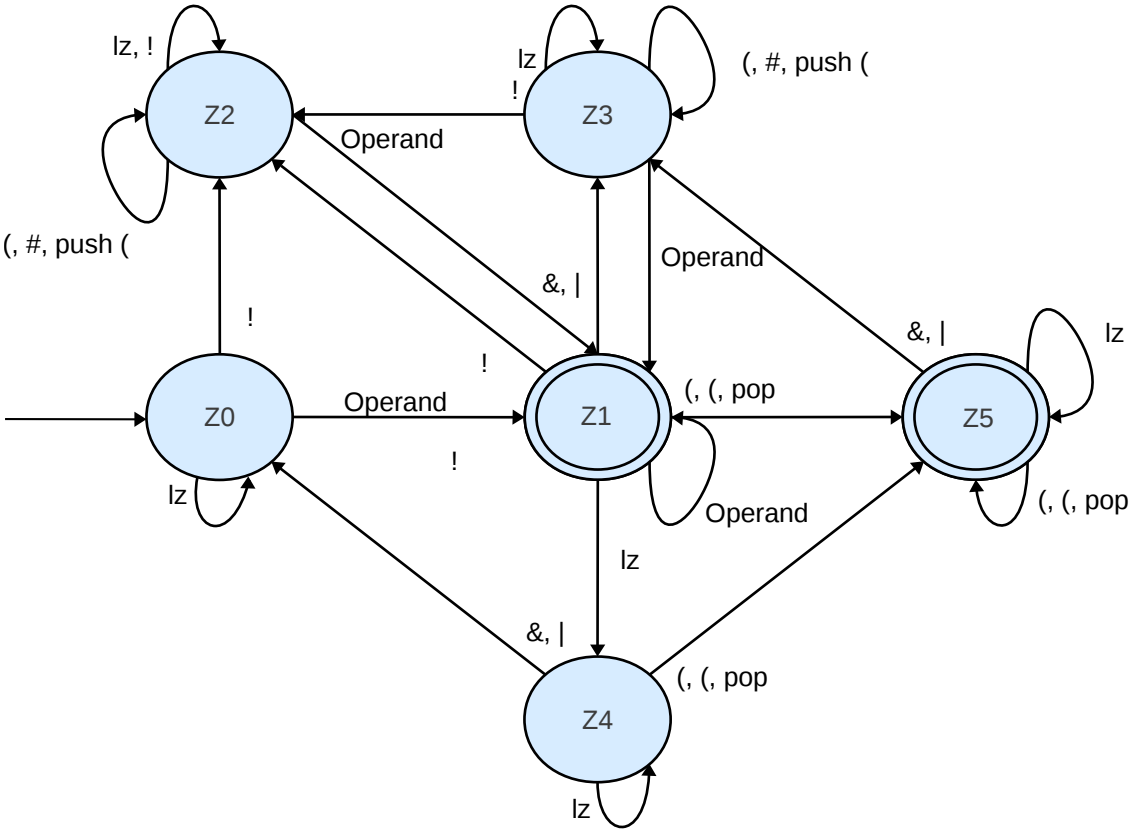


Abbildung 5.10: Kellerautomat zur Syntaxüberprüfung

Die Überprüfung der zweiten Bedingung ist deutlich komplexer. Dafür verwendet das aussagenlogische Plugin einen Kellerautomaten, denn mit diesem ist es möglich, einen Text auf Konformität zur Syntax der Aussagenlogik zu überprüfen. Der Automat wird durch das Tupel $PDA = (K, \Sigma, D, Z_0, \#, F, \Delta)$ beschrieben. Die Definition der Elemente

lautet:

$K := \{Z_0, Z_1, Z_2, Z_3, Z_4, Z_5\}$	Zustandsmenge
$\Sigma := \{OS, lz, !, \&, \mid\}$	Eingabealphabet, "lz" bedeutet Leerzeichen
$OS := \{a-z, A-Z, 0-9\}$	Operandensymbole, Symbole welche Name des Operanden beinhalten darf
$D := \{\#, (\}$	Kelleralphabet
$F := \{Z_1, Z_5, \}$	finale Zustände

Abbildung 5.10 zeigt das Zustandsdiagramm vom Entwickelten Kellerautomaten. Diesem Diagramm können die Übergangsrelationen Δ entnommen werden. Durch den Automaten kann das Plugin den Anwender über die folgenden Fehler informieren:

Falsche Klammerung: Fehler, die auf eine falsche Klammerung zurückzuführen sind. Ein mögliches Beispiel ist *A & (A & B*

Fehlender Operator: Fehler, welche durch einen fehlenden Operator verursacht werden. Beispiele sind *A B* oder *A (A | B)*

Fehlender Operand: Fehler, denen ein fehlender Operand zu Grunde liegt. Mögliche Beispiele sind *A &*, *& A* oder *A & (A |)*

5.2.3 Modellierung einer Bedingung

Für die Modellierung durch die Aussagenlogik steht dem Anwender die durch Abbildung 5.11 dargestellte Oberfläche zur Verfügung. Im abgebildeten Textfeld kann der Nutzer die Formel verfassen. Welche Formeln es bereits gibt und welche er verwenden kann, sieht er in der dargestellten Liste. Diese Liste wird ebenfalls für die Autovervollständigung verwendet. Das heißt, während der Benutzer die Formel für die Bedingung verfasst, aktualisiert sich die Liste. So bewirkt das Eintippen der Buchstabenfolge *"Hat"* beispielsweise, dass die Liste nur noch Formeln anzeigt, deren Name diese Buchstabenfolge enthält. Über die Pfeiltasten oder der Maus kann der Anwender dann die passende Formel auswählen. Das Propositional-Logic-Plugin fügt diese an die Stelle ein, an der sich die Einfügemarke im Textfeld befindet.

Während der Anwender die Formel verfasst, informiert ihn das Plugin ständig über die Korrektheit der Formel. Das Plugin löst die Korrektheitsüberprüfung immer dann

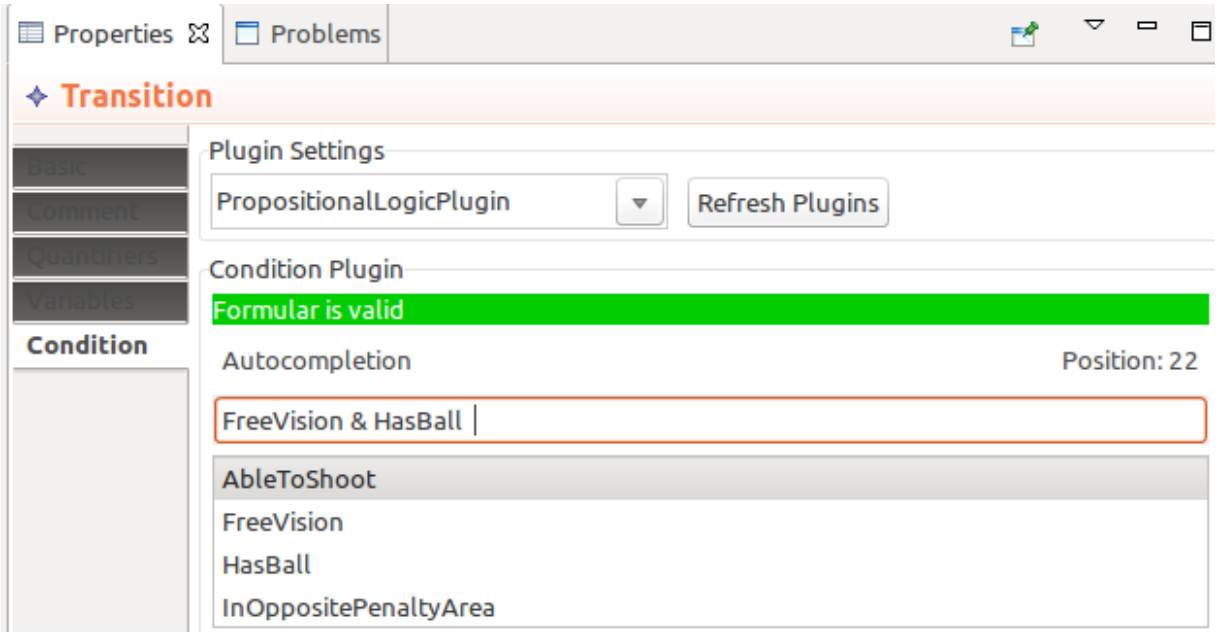


Abbildung 5.11: Oberfläche zum Verfassen einer aussagenlogischen Formel

aus, wenn sich der Text innerhalb des Textfeldes verändert. Die Anzeige des Ergebnisses erfolgt in der Statusleiste. In der Abbildung 5.11 ist die Statusleiste an ihrer grünen Hintergrundfarbe zu erkennen. Diese zeigt an, dass die Formel korrekt ist. Wenn die Formel fehlerhaft ist, wechselt die Hintergrundfarbe zu Rot und der Nutzer erhält eine aussagekräftige Fehlermeldung.

Da auch für die verfasste Formel gilt, dass sie über die Laufzeit des Plan Designers hinweg zur Verfügung stehen muss, speichert das Plugin diese. Des Weiteren muss der Code Generator während der Generierung von Quelltext auf die Formel und auf weitere, durch das Plugin erstellte, Inhalte zugreifen. Daher stellt die Serialisierung und Persistierung durch EMF eine geeignete Lösung dar. Aus diesem Grund nutzt das aussagenlogische Plugin die Änderung am Modell-Plugin und speichert in der eigens erstellten Objektstruktur die folgenden pluginspezifischen Inhalte:

Formel Die verfasste Formel, so wie sie der Anwender im Textfeld eingegeben hat.

Aufgelöste Formel Eine Formel ist aufgelöst wenn sie als Teilformeln nur noch Aussagen besitzt. Seien zum Beispiel A und B zwei Aussagen und $Formel1 = A \,\&\, B$, $Formel2 = !A$, sowie $Formel3 = Formel1 \,\&\, Formel2$ gegeben. Dann sind Formel1 und Formel2 bereits aufgelöste Formeln. Formel3 hingegen lässt sich noch weiter auflösen, nämlich zu $Formel3 = (A \,\&\, B) \,\&\, !A$.

Operanden der aufgelösten Formel Alle Operanden, also alle Aussagen, welche die aufgelöste Formel enthält. Sei zum Beispiel die aufgelöste Formel $A \mid !A \,\&\, B$

gegeben. Dann sind die Operanden dieser aufgelösten Formel A und B.

Die aufgelöste Formel und deren Operanden werden gespeichert, da somit die Verarbeitung der Formel während der Quelltextgenerierung deutlich einfacher ist. Der Abschnitt 5.2.5 erläutert die Verarbeitung näher. Immer wenn sich der Text im Textfeld ändert, schreibt das Plugin die oben aufgelisteten Daten in die Objektstruktur zum Speichern pluginspezifischer Inhalte. Wenn der Anwender den Speichervorgang des Plan Designers auslöst, speichert dieser die pluginspezifischen Inhalte endgültig mittels EMF.

5.2.4 Erstellen und Verändern einer Formel für Bedingungen

Im Laufe der Zeit kann es nötig sein, dass der Anwender eine neue Formel braucht oder eine existierende Formel verändern muss. Für diese Aufgaben steht der Formel Editor zur Verfügung. Die Abbildung 5.12 zeigt den Formel Editor. Die Oberfläche ähnelt der zum Modellieren einer Bedingung. An zusätzlichen Elementen enthält der Formel Editor ein Textfeld, mit dem der Anwender einen Namen für eine Formel definieren kann und links eine weitere Liste, mit welcher er die zu verändernde Formel auswählt. Zum Erstellen einer neuen Formel muss der Nutzer auf den Button *New* kli-

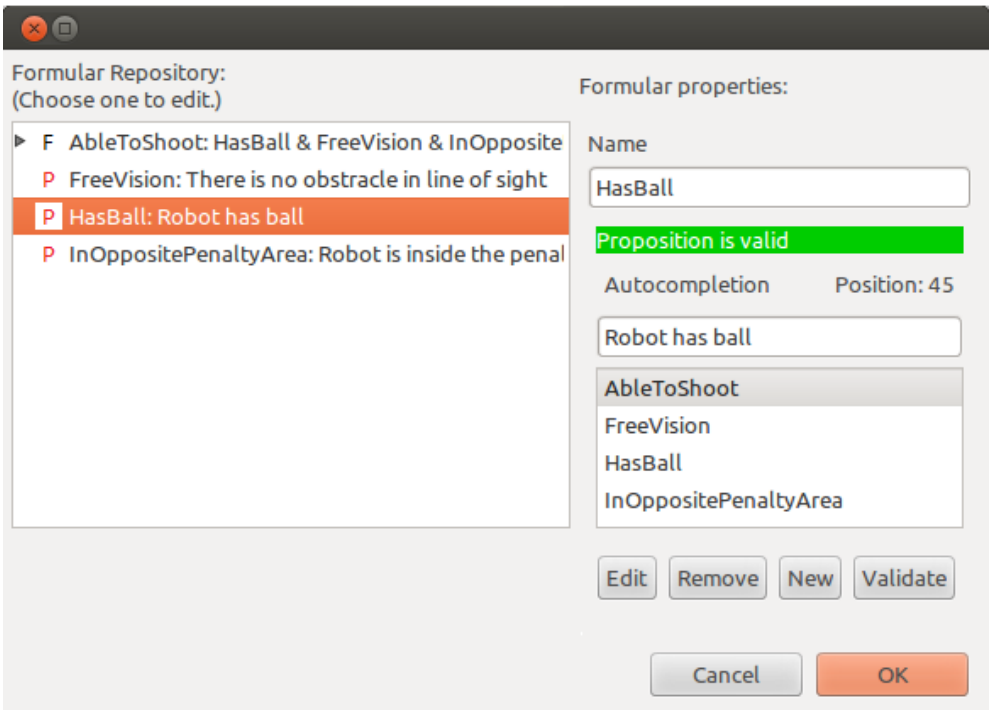


Abbildung 5.12: Formel Editor

cken. Anschließend kann er Name und Ausdruck der Formel definieren. Auch bei diesem Vorgang ist die Autovervollständigung verfügbar. Ebenso findet die Korrektheits-

überprüfung statt. Möchte der Anwender die Formel in das Vokabular übernehmen, muss er auf den Button *Add* klicken. Eine Übernahme ins Vokabular ist nur möglich, wenn die Formel valide ist. Andernfalls erhält der Nutzer eine Fehlermeldung. Nach der Übernahme in das Vokabular persistiert das aussagenlogische Plugin dieses neu.

Der Vorgang zum Ändern einer Formel ähnelt dem zum Erstellen einer Formel. In der Liste links wählt der Anwender die zu verändernde Formel aus. Anschließend kann er Name und die Formel verändern. Das Übernahmekriterium ist auch hier, dass die Formel valide ist. Zum Löschen der Formel steht dem Nutzer der Knopf *Delete* zur Verfügung. Nach erfolgreicher Übernahme der veränderten Formel in das Vokabular persistiert das aussagenlogische Plugin dieses neu.

5.2.5 Quelltextgenerierung

Für die Definition des Aufbaus des Quelltextes bietet das Plugin ein Template an, welches die Pluginschnittstelle weiter verarbeitet. Der generierte Quelltext unterscheidet sich nur wenig von dem bisher generierten Quelltext. Zum Vergleich seien die Quelltexte in Listing 5.4 und Listing 5.5 gegeben.

```
/*
*
* Transition:
*   - Name: Shoot Condition, ConditionString: Roboter is in
*   opposite penalty area, has the ball and free vision,
* Comment :
*
* Plans in State:
*   - Plan - (Name): 22Attack, (PlanID): 1235973370172
*
* Tasks:
*   - Attack (1222613952469) (Entrypoint: 1236002535776)
*   - Defend (1225115406909) (Entrypoint: 1236002586769)
*
* States:
*   - Defend (1236002594052)
*   - Attack (1236002535777)
*   - NewFailurePoint (1239726734451)
*   - NewFailurePoint (1239631086333)
*
* Vars:
*/
public static bool F1239631090101(RunningPlan rp) {
        /*PROTECTED REGION ID(1239631090101) ENABLED START*/
                //WorldModel wm = WorldModel.Get();
                --> "Transition: 1239631090101  not implemented";
                // return false;
```

```
        /*PROTECTED REGION END*/

}
```

Listing 5.4: Generierter Quelltext einer Bedingung, die mit der bisherigen Modellierungsmethode modelliert wurde.

```
/*
* Transition:
*   - Name: , Comment :
*   - Formula:
*       (HasBall & FreeVision & InOppositePenaltyArea)
*
* Plans in State:
*   - Plan - (Name): 22Attack, (PlanID): 1235973370172
*
* Tasks:
*   - Attack (1222613952469) (Entrypoint: 1236002535776)
*   - Defend (1225115406909) (Entrypoint: 1236002586769)
*
* States:
*   - Defend (1236002594052)
*   - Attack (1236002535777)
*   - NewFailurePoint (1239726734451)
*   - NewFailurePoint (1239631086333)
*/
public static bool F1239631090101(RunningPlan rp) {
        /*PROTECTED REGION ID(1239631090101) ENABLED START*/
                //WorldModel wm = WorldModel.Get();
                --> "Transition: 1239631090101  not implemented";
                // return false;
        /*PROTECTED REGION END*/
        bool HasBall;
        bool FreeVision;
        bool InOppositePenaltyArea;
}
```

Listing 5.5: Generierter Quelltext mit einer durch dem aussagenlogischen Plugin modellierten Bedingung

Listing 5.4 zeigt den generierten Quelltext durch die vorherige Modellierungsmethode, also jene, welche sich jetzt im Default-Plugin befindet. Listing 5.5 hingegen zeigt den durch das Propositional-Logic-Plugin generierten Quelltext. Für den Anwender ist wichtig, dass er die Formel sehen kann. Daher beinhaltet der Kommentar die Formel, genauer um die aufgelöste Formel. Weiterhin ist erkennbar, dass durch das Template die Quellcodegenerierung für jeden Operanden der aufgelösten Formel eine boolsche Variable im Methodenrumpf generiert. Wie genau diese definiert ist, wird später durch

den Entwickler von Hand nachprogrammiert.

Im Propositional-Logic-Plugin legt das Plugin-Template fest, wie der generierte Quelltext aufgebaut ist. Es bietet alle durch das Schnittstellen-Template festgelegten Aspekte an, innerhalb denen geprüft wird, ob die Bedingung durch das Propositional-Logic-Plugin modelliert wurde. Beim Codegenerieren prüft der Code Generator diese Bedingung und löst im positiven Fall die Quelltextgenerierung aus.

5.2.6 Konfigurieren des Plugins

Der Anwender kann die Einstellungen vom Propositional-Logic-Plugin durch die in Abbildung 5.13 gezeigte Oberfläche konfigurieren. Die Konfiguration beinhaltet das Setzen des Pfades zum Aussagen-Vokabular und Formel-Vokabular. Übernimmt der Plan Designer die Einstellungen, liest das aussagenlogische Plugin die neuen Vokabulare ein. Vor dem Wechseln des Vokabulares sollte der Anwender überprüfen, dass die neuen Vokabulare auch all jene Formeln enthalten, die bisher in den Bedingungen verwendet wurden. Nur dadurch ist gewährleistet, dass die Quelltextgenerierung fehlerfrei funktioniert.

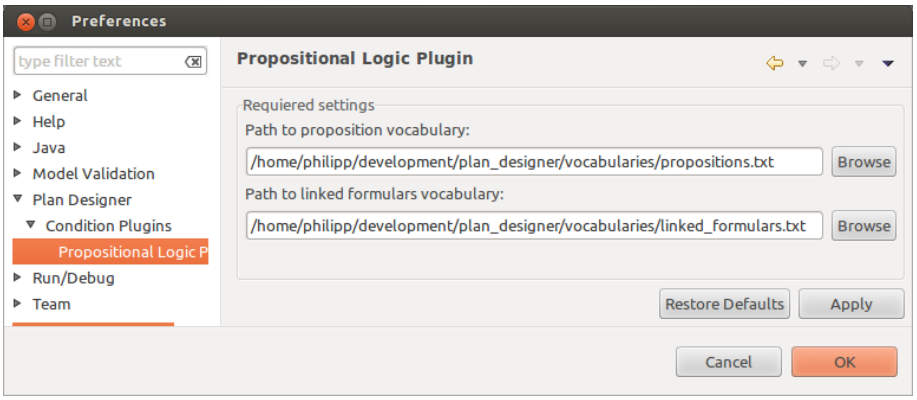


Abbildung 5.13: Fenster zum Konfigurieren des Propositional-Logic-Plugins

5.3 Herstellen der Abwärtskompatibilität

Die Problemstellung legt fest, dass die bisherige Modellierungsmöglichkeit erhalten bleiben soll. Dies sichert die Abwärtskompatibilität. Aus diesem Grund befindet sich die dazu nötige Funktionalität ebenfalls in einem Condition-Plugin, dem Default-Plugin.

Dies ist ebenfalls ein Eclipse Plugin, welches die Condition-Plugin-Extension anbietet. Die Oberfläche, welche das Plugin dadurch anbietet, entspricht der bisher verwendeten Oberfläche. Sämtlich Funktionalität, welche sich für die Modellierung im Plan Designer befand, befindet sich nun in diesem Plugin. Zusätzlich besitzt das Plugin ein Template, das Aspekte definiert, welche die Verantwortlichkeit des Plugins überprüfen. Dadurch verarbeitet es die Pluginschnittstelle erfolgreich. Abbildung 5.14 zeigt die

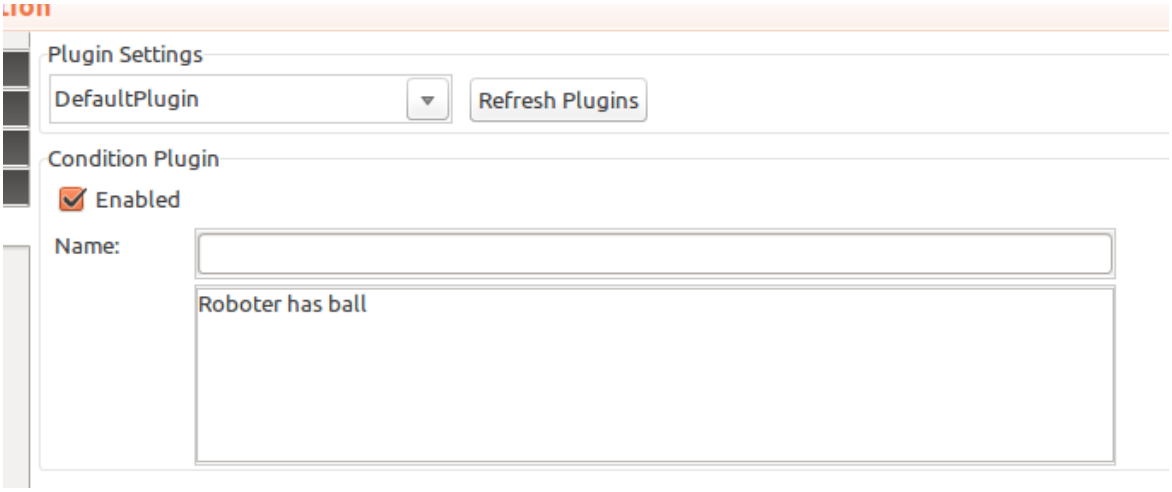


Abbildung 5.14: Oberfläche des Default-Plugins

Oberfläche des Default-Plugins. Dies entspricht der bisherigen Oberfläche.

Kapitel 6

Evaluation

Dieses Kapitel evaluiert die im Rahmen dieser Arbeit durchgeführten Änderungen am Plan Designer und das neu entstandene Plugin zur Modellierung von Bedingungen durch die Aussagenlogik. Dazu verdeutlicht es in den folgenden Abschnitten die Vor- und Nachteile der Pluginschnittstelle und des neuen Plugins. Weiterhin deckt das Kapitel am Beispiel des neuen aussagenlogischen Plugins die Vorteile für die Quelltextgenerierung auf.

6.1 Bewertung der Pluginschnittstelle

Bisher konnte der Anwender eine Bedingung nur auf eine Art und Weise modellieren. Der Vorgang beinhaltet das Verfassen einer Beschreibung für die Bedingung, welche die Quelltextgenerierung mit generiert. Dabei verzichtet der Plan Designer auf eine Überprüfung der Beschreibung auf etwaige Fehler. So können sich Fehler beispielsweise bis auf die C#-Ebene durchziehen. Eine Überprüfung ist auch gar nicht möglich, da die Beschreibung keinen konkreten Formalismus entspricht, sondern vom Anwender frei bestimmt wird. Durch die Pluginschnittstelle ist es möglich einen beliebigen Formalismus für die Modellierung auszuwählen und diesen als Condition-Plugin umzusetzen. Wird in Zukunft beispielsweise festgestellt, dass es vorteilhaft ist, wenn Anwender Bedingungen durch die Prädikatenlogik erster Ordnung modellieren können, dann können Entwickler dafür ein Condition-Plugin entwickeln, welches die Pluginschnittstelle in den Plan Designer integriert. Der Entwickler des Condition-Plugins kann dieses für den jeweiligen Formalismus, den es umsetzt, optimieren. So kann das Plugin den Anwender während des Modellierungsprozesses unterstützen. Außerdem kann das Plugin die Modellierung auf Fehler überprüfen.
Durch die Pluginstruktur ist es möglich, die Entwicklung der Condition-Plugins vom Plan Designer abzuspalten. Es ist nicht nötig direkt am Plan Designer zu entwickeln.

Der Plan Designer wird lediglich vom Entwickler benutzt, um die Condition-Plugins zu testen. Es reicht dabei den Plan Designer einmal zu starten und die Pluginschnittstelle für die Integration des Condition-Plugins zu nutzen. Die Abspaltung erhöht die Strukturierung des Quelltextes und ermöglicht die einfache Verteilung verschiedener Plugins auf verschiedene Entwickler. Beim Überführen des Condition-Plugins in die Versionsverwaltung muss der Entwickler keine zeitaufwendigen Versionierungsarbeiten, wie das Zusammenführen von Quellcode des Plan Designers und des Plugins durchführen.

Die neue Objekt-Struktur in der Bedingung zum Speichern pluginspezifische Inhalte ermöglicht das Hinzufügen von Inhalten zu der Bedingung ohne, dass der Entwickler im Modell die Klasse der Bedingung verändern muss. Dies grenzt den Plan Designer von den Condition-Plugins ab, erhöht die Übersicht und erspart Aufwand bei der Entwicklung. Auf diese Weise ist es nicht nötig für jedes neue Plugin Attribute zu der Bedingung hinzuzufügen oder sie zu entfernen, wenn die Nutzung eines Plugins nicht weiter erfolgt. Der Entwickler definiert lediglich Schlüssel, welche die Objekt-Struktur auf die pluginspezifischen Inhalte abbildet.

Ein wenig unhandlich gestaltet sich der Bauvorgang eines Condition-Plugins. Der Entwickler erstellt das Plugin in der Eclipse IDE. Das fertige Java-Archiv muss er dann gegebenenfalls von Hand in das Pluginverzeichnis des Plan Designers verschieben und anschließend die alte Version des Condition-Plugins aus dem Verzeichnis löschen.

6.2 Bewertung des aussagenlogischen Plugins

Aktuell stehen dem Benutzer zwei verschiedene Modellierungsmöglichkeiten zur Verfügung. Die bisherige Modellierung über einen beschreibenden Text und die Modellierung durch die Aussagenlogik. Durch die Verwendung des aussagenlogischen Plugins ergeben sich für den Anwender verschiedene Vorteile gegenüber der bisherigen Modellierungsmöglichkeit. Durch die Autovervollständigung während der Modellierung unterstützt das Plugin den Anwender in seiner Effizienz. Der Benutzer kann zum einen die Formel schneller verfassen und wird zum anderen nicht dazu verleitet, auf nicht existierende Formeln zurückzugreifen. Außerdem zeigt das aussagenlogische Plugin durch die Statusleiste dem Nutzer sofort an, wenn etwas mit seiner Modellierung nicht in Ordnung ist. So fallen Fehler gleich auf und der Anwender kann diese sofort beheben. Durch die Darstellung der aussagenlogischen Formeln in einer Liste sieht der Anwender gleich, was das Weltmodell unterstützt. Unterstützt das Weltmo-

dell zum Beispiel die Erkenntnis, dass sich der Roboter in der gegnerischen Hälfte befindet, gibt es dafür auch eine Aussage der Form '*RoboterIstInGegnerischerHälfte*.

6.3 Vorteile für die Quelltextgenerierung

Dadurch, dass durch die Pluginschnittstelle die Möglichkeit besteht, feste Formalismen als Condition-Plugin umzusetzen, können zukünftige Arbeiten die Quelltextgenerierung optimieren. Bisher war das Problem, dass der beschreibende Text nicht mit dem Weltmodell verknüpft werden konnte, da er keinen Konventionen oder einem konkreten Formalismus entspricht. Die Condition-Plugins lösen dieses Problem. Die Erklärung soll am Beispiel des aussagenlogischen Plugins erfolgen. Die Quelltextgenerierung erstellt momentan nur Methodenrümpfe, nicht aber die Logik, welche für die Auswertung der Bedingung nötig ist. Durch die Verwendung eines Formalismus, wie der Aussagenlogik, können zukünftige Arbeiten das aussagenlogische Plugin mit dem Weltmodell verknüpfen und so die Quelltextgenerierung optimieren. Das Weltmodell bietet Funktionen an, welche Entwickler zum Auswerten der Bedingung verwenden können. So gibt es Funktionen, um die Position des Balls oder des Roboters zu ermitteln. Bei einer Verknüpfung des aussagenlogischen Plugins mit dem Weltmodell, kann eine Aussage als Schlüssel zu einer passenden Funktion im Weltmodell fungieren. Sei zum Beispiel die Aussage *RoboterHatBall* gegeben. Unter der Annahme, dass das Weltmodell mit dem aussagenlogische Plugin verknüpft wurde, kann sich dadurch Quelltext generieren lassen, der für die Aussage *RoboterHatBall* bereits die passende Funktion aus dem Weltmodell aufruft.
Auf diese Art und Weise erhält man für eine Bedingung nicht nur den Methodenrumpf, sondern gleich auch die Logik, welche die Bedingung auswertet. Die Verknüpfung kann zum Beispiel mit Annotationen geschehen. Das heißt, die Funktionen im Weltmodell werden annotiert und die Condition-Plugins verwenden diese Annotationen, um auf die passende Funktion zuzugreifen.

Kapitel 7

Zusammenfassung

Agenten in Multi-Agenten-Systemen sind dazu in der Lage eigenständig zu handeln und auf ihre Umwelt zu reagieren. Dieses Handeln, also das Verhalten der Agenten, muss der Entwickler zuvor spezifizieren. Für die Verhaltensmodellierung existieren verschiedene textuelle und grafische Rahmenwerke. ALICA ist ein solches Rahmenwerk, welches an der Universität Kassel entwickelt wurde und die grafische Verhaltensmodellierung ermöglicht. ALICA wurde als domänenunabhängiges Rahmenwerk entwickelt, jedoch ist es für die Beschreibung von konkreten Verhalten nötig, domänenabhängige Bedingungen zu formulieren. Ohne domänenabhängige Bedingungen kann der Entwickler auch kein Verhalten für eine konkrete Domäne modellieren. In der Domäne der Fußballroboter ist eine solche domänenabhängige Bedingung zum Beispiel *"Wenn der Roboter in der gegnerischen Hälfte steht und den Ball hat, kann er schießen"*. Mit dem Plan Designer steht ein Werkzeug zur Verfügung, welches die Verhaltensmodellierung mit ALICA ermöglicht.

Die Ergebnisse dieser Arbeit bieten eine Möglichkeit zur Formulierung von domänenabhängigen Bedingungen in ALICA durch unterschiedliche Formalismen. Dabei wird die Bedingung, dass ALICA ein domänenunabhängiges Rahmenwerk ist, nicht verletzt. Für die Umsetzung erhält Plan Designer eine Pluginschnittstelle. Zwei zur Schnittstelle konforme Plugins ermöglichen das Modellieren von Bedingungen auf unterschiedliche Art und Weise. Ein Plugin sichert die Abwärtskompatibilität, ermöglicht also die Modellierung einer Bedingung durch die herkömmliche Methodik. Ein weiteres Plugin erlaubt die Formulierung von Bedingungen durch die Aussagenlogik.

7.1 Schlussfolgerung

Durch die durchgeführten Änderungen am Plan Designer ist es dem Entwickler möglich für Verhalten, welche er mit ALICA modelliert, verschiedene Formalismen für die Modellierung von domänenabhängigen Bedingungen zu verwenden. Dabei wurde ALICA selbst nicht verändert und bleibt somit weiterhin domänenunabhängig. Durch die neue Pluginschnittstelle können Entwickler weitere Formalismen in Zukunft hinzufügen. Dafür muss ein Plugin erstellt werden, welches konform zur Pluginschnittstelle ist. Die Plugins werden von der Pluginschnittstelle zur Laufzeit des Plan Designers in diesen integriert. Dadurch wird Zeit und Aufwand bei der Entwicklung eingespart. Die während dieser Arbeit erstellten Plugins ermöglichen das Formulieren einer Bedingung auf die herkömmliche Art und Weise und durch die Aussagenlogik. Das aussagenlogische Plugin unterstützt den Anwender durch eine Autovervollständigung und Fehlerbenachrichtigung bei der Erstellung einer aussagenlogische Formel.

7.2 Ausblick

Dieser Abschnitt stellt mögliche zukünftige Arbeiten vor. Diese können auf dieser Arbeit aufbauen oder Ergebnisse dieser Arbeit weiterentwickeln.

7.2.1 Hinzufügen neuer Formalismen

In Zukunft ist es möglicherweise wünschenswert, dass Anwender des Plan Designers neue Formalismen für die Beschreibung einer Bedingung nutzen können. Zukünftige Arbeiten könnten die Integration eines neuen Formalismus in den Plan Designer beinhalten. Dafür muss ein neues Plugin erstellt werden. Dieses muss konform zu der Pluginschnittstelle sein. Das Plugin erweitert die grafische Oberfläche des Plan Designers und besitzt die vollständige Funktionalität um die Bedingung zu modellieren und daraus Quellcode zu generieren. Weitere mögliche Formalismen sind zum Beispiel die Prädikatenlogik erster Ordnung, verschiedene räumliche Logiken [16], Modal-Logiken [17] oder auf Answer Set Programming basierende Formalismen [18].

7.2.2 Anbindung an das Weltmodell

Der Abschnitt 6.3 erläutert, dass die Quelltextgenerierung nur das Grundgerüst für die Bedingung generiert, nicht aber die Logik, welche die Bedingung auswertet. Dadurch ist es nötig, dass der Entwickler den Quelltext manuell nacharbeitet. In zukünftigen Arbeiten könnte das aussagenlogische Plugin mit dem Weltmodell verknüpft werden.

Dadurch ist es möglich Quelltext zu generieren, welcher die Logik für die Auswertung der Bedingung beinhaltet und somit das nachträgliche Editieren von Hand überflüssig macht.

7.2.3 Validität von Plänen überprüfen

Bei der Überprüfung der Validität von Plänen ist es auch nötig, die verschiedenen Bedingungen zu validieren. So ist zum Beispiel die Feststellung möglich, dass sich Bedingungen über die verschiedenen Hierarchieebenen eines Plans nicht widersprechen. Wenn Bedingungen nun aber mit verschiedenen Formalismen modelliert sind, muss es eine Möglichkeit geben, diese für die Validitätsüberprüfung zu verknüpfen. Zukünftige Arbeiten könnten sich mit der Verknüpfung von Formalismen beschäftigen, um somit die Validierung der Bedingungen zu ermöglichen.

Literaturverzeichnis

[1] Opfer, S. (2012) *Towards a Description Logic Support for ALICA*. Master's thesis, University of Kassel.

[2] Scharf, A. (2008), Grafische verhaltensmodellierung für kooperative autonome roboter.

[3] Risler, M. et al., Xabsl. `http://www.xabsl.de/` (Abgerufen am 30.03.2014).

[4] Wooldridge, M. (2005), Intelligent Agents: The Key Concepts. `http://www2.hawaii.edu/~nreed/ics606/papers/wooldridge02agentkey23220003.pdf` (Abgerufen am 20.02.2014).

[5] Wooldridge, M. and Jennings, N. (1995) Intelligent angents: theory and practice. *The Knowledge Engineering Review*, **10**, 115–152.

[6] Skubch, H. (2013) *Modelling and Controlling of Behaviour for Autonomous Mobile Robots*. Westdeutscher Verlag GmbH.

[7] OSGi-Alliance (2003) *OSGi service platform, release 3, March 2003*. I O S Press.

[8] Wütherich, G. (2009) *Die OSGi Service Platform : eine Einführung mit Eclipse Equinox*. dpunkt-Verl., 1. aufl., korrigierter nachdr. edn.

[9] Sippel, H., Jastram, M., and Bendisposto, J. (2008) *Eclipse rich client platform : Entwicklung von erweiterbaren Anwendungen mit RCP*. Entwickler.Press.

[10] Steinberg, D. (2008) *EMF - Eclipse modeling framework*. The eclipse series, Addison Wesley, 2. ed., revised and updated edn.

[11] Efftinge, S. et al., openarchitectureware. `http://www.openarchitectureware.org/index.php` (Abgerufen am 30.03.2014).

[12] Schöning, U. (2005) *Logik für Informatiker*. Spektrum.

[13] Erk, K. and Priese, L. (2002) *Theoretische Informatik - Eine umfassende Einführung*. Springer.

[14] Loetzsch, M., Risler, M., and Jungel, M. (2006) Xabsl - a pragmatic approach to behavior engineering. *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, Oct, pp. 5124–5129.

[15] Hindriks, K., De Boer, F., Van der Hoek, W., and Meyer, J.-J. (1999) Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, **2**, 357–401.

[16] Aiello, M., Pratt-Hartmann, I., and van Benthem, J. (eds.) (2007) *Handbook of Spatial Logics*. Springer.

[17] Blackburn, P., Benthem, J. F. A. K. v., and Wolter, F. (2006) *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc.

[18] Eiter, T., Ianni, G., and Krennwallner, T. (2009) Answer set programming: A primer. Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., and Schmidt, R. (eds.), *Reasoning Web. Semantic Technologies for Information Systems*, vol. 5689 of *Lecture Notes in Computer Science*, pp. 40–110, Springer Berlin Heidelberg.