Modellierung und Verifikation von Eisenbahn-Infrastrukturen mit semantischen Technologien

Dissertation

vorgelegt der
Technischen Fakultät

der Christian-Albrechts-Universität zu Kiel

Michael Lodemann
Institut für Informatik

27. Januar 2015

Erster Berichterstatter	Prof. Dr.-Ing. Norbert Luttenberger Institut für Informatik Christian-Albrechts-Universität zu Kiel
Zweiter Berichterstatter	Prof. Dr. Stefan Fischer Institut für Telematik Universität zu Lübeck
Disputationstermin	10. Dezember 2014
Prüfungskommissionsvorsitz	Prof. Dr. Manfred Schimmler
Weiteres Prüfungskommissionsmitglied neben den beiden Berichterstattern	Prof. Dr. Andreas Speck

Zusammenfassung

Diese Arbeit zeigt, wie semantische Technologien (Sprachen und Werkzeuge) genutzt werden können, um Eisenbahn-Infrastrukturen zu modellieren und zu verifizieren. Dazu wurde ein semantisches Domänen-Modell entworfen und implementiert, das sowohl die Elemente von Eisenbahn-Infrastrukturen und deren Eigenschaften als auch die Beziehungen dieser Elemente untereinander darstellen kann. Zur Modellierung wurde die Web Ontology Language (OWL) in Kombination mit der Semantic Web Rule Language (SWRL) verwendet. Im Rahmen der Arbeit wurden Planungsrichtlinien von Eisenbahn-Infrastrukturen analysiert und exemplarisch in SWRL formalisiert. So wurde es ermöglicht, konkrete Eisenbahn-Infrastruktur-Planungsdaten in Instanzen des semantischen Domänenmodells zu transformieren, um diese Instanzen hinsichtlich ihrer Konformität mit Planungsrichtlinien zu verifizieren. Dieser Prozess wird anhand mehrerer Beispiele veranschaulicht.

Ein weiterer Teil der Arbeit befasst sich mit einem Verfahren, mit dem Eisenbahn-Ingenieure bei der Suche nach Fehlern in Planungsdaten unterstützt werden können. Während OWL- und SWRL-Reasoner lediglich eine allgemeine Eingrenzung von Fehlern im zu verifizierenden Modell ermöglichen, wird durch Verwendung des in dieser Arbeit vorgestellten Konzepts der Semantic Constraints eine Eingrenzung auf konkrete fehlerhafte Axiome in Bezug auf die modellierten Planungsrichtlinien realisiert. Das Konzept basiert auf der Regelsprache SWRL und ermöglicht die Formulierung von Bedingungen, die dazu verwendet werden, Eisenbahn-Infrastruktur-Elemente in Form von OWL-Individuen zu überprüfen. Verstoßen einzelne Individuen gegen eine Bedingung, wird diese systematisch reduziert, um die Ursachen für den Verstoß gegen die Bedingung und die damit zusammenhängende Planungsrichtlinie zu ermitteln.

In der Diskussion wird die Vergleichbarkeit des vorgestellten Ansatzes mit aktuell verwendeten XML-basierten Verfahren untersucht. Die Arbeit schließt mit einem Ausblick auf mögliche Weiterentwicklungen der vorgestellten Verfahren ab.

Inhaltsverzeichnis

1 Einleitung

1.1 Einführung und Problemstellung

Die Eisenbahn stellt nicht nur ein wichtiges Transportmittel für den Personenverkehr dar, sie ist auch in der Logistik unverzichtbar. Deutschland gehört zu den Ländern der Welt, die über ein weit ausgebautes Schienennetz verfügen. Es erstreckt sich auf einer Gesamtlänge von über 37.000 km (Bezugsjahr 2010) [12] und hat somit eine hohe gesellschaftliche Relevanz im Personen- und Güterverkehr. Das Schienennetz wird ständig erweitert und modernisiert. Es ist nachvollziehbar, dass ein umfangreicher und komplexer Planungsprozess von Eisenbahn-Infrastrukturen insbesondere Sicherheitsaspekte berücksichtigen muss.

Eine Eisenbahn-Infrastruktur (EI) ist ein Gebilde aus einem Netz von Gleiselementen wie Schienen und Weichen, auf dem Zugbewegungen stattfinden. Zusätzlich zum Gleisnetz gehören auch gleisseitige Sicherheitskomponenten wie beispielsweise Signale, Achszähler etc. zu einer EI. Diese Sicherheitskomponenten sind entweder für die Überwachung oder die Beeinflussung von Zugbewegungen zuständig. Alle Weichen und Sicherheitskomponenten einer Zugstrecke werden zentral in einem Stellwerk beaufsichtigt und gesteuert. Die Aufgabe dieser zentralen Steuereinheit ist es, Zügen festgelegte Zugfahrten auf vordefinierten sogenannten Fahrstraßen über das Gleisnetz sicher zu ermöglichen. Fahrstraßen können als überlagerndes Netz von befahrbaren Strecken auf dem eigentlichen physikalischen Gleisnetz interpretiert werden. Eine Fahrstraße ist zusammengesetzt aus einer bestimmten Reihenfolge von sogenannten Blockabschnitten, die die kleinsten sicherungstechnischen Einheiten aus Stellwerkssicht darstellen. Die Sicherheitstechnik impliziert die Gewährleistung einer exklusiven Nutzung von Blockabschnitten für die jeweiligen Züge. Diese exklusive Nutzbarkeit wird innerhalb des Stellwerks durch bestimmte Zustände der beteiligten Weichen, Signale und je nach Szenario anderen Sicherheitskomponenten geregelt.

Im Laufe der Entwicklung von Eisenbahn-Infrastrukturen wurden von den entsprechenden Eisenbahnbehörden Planungsrichtlinien erarbeitet, in denen beschrieben wird, wie eine Infrastruktur von Gleisanlagen mit sicherungstechnischen Komponenten aus-

gestattet sein muss, um die oben genannten Ziele zu ermöglichen.

Im Kontext der vorliegenden Arbeit ist der Begriff 'Eisenbahn-Infrastruktur-Verifikation' als ein Prozess zu verstehen, der die Einhaltung sämtlicher zutreffender Planungsrichtlinien in Bezug auf gegebene Eisenbahn-Infrastrukturen gewährleistet. Es existieren vielfältige Arten von Planungsrichtlinien. Auch wenn sie ein allgemeingültiges Ziel verfolgen, beziehen sich diese unterschiedlichen Regelsätze auf länderspezifische Gegebenheiten und auf verschiedene Arten von Schienenverkehr wie beispielsweise langsame Regional- oder überregionale Hochgeschwindigkeitszugstrecken.

Es ist offensichtlich, dass das Bestreben nach einer EI-Planungsverifikation nur erfolgreich ist, wenn als Grundlage ein fundiertes und weitreichendes Modell sowohl für die physikalischen (z.B. Weichen, Gleise, Signale) als auch für die virtuellen EI-Elemente wie beispielsweise Fahrstraßen und Blockabschnitte vorhanden ist. Die Modellierung ist, wie der Titel der vorliegenden Arbeit andeutet, ein wesentlicher Bestandteil und die Basis für erfolgreiche und zielgerichtete Verifikationsprozesse. Unter Modellierung wird hier die formale Abbildung von realweltlichen Objekten einer Problemdomäne in konzeptuelle Objekte eines informatischen Systems verstanden [14]. Wenn das Modell einer formalen Semantik unterliegt, wird es möglich, Konzepte, Eigenschaften und Relationen zu erfassen und darüber zu räsonieren. Somit können spezielle Programme (sogenannte Reasoner) dazu genutzt werden, komplexe Abfragen an ein Modell zu stellen, um so implizite Zusammenhänge zu verdeutlichen und zu explizieren. Semantische Technologien sind dadurch ausgezeichnet, Informationssystemen die formale Repräsentation von Modellen, ihrer Konzepte, Eigenschaften und Relationen zu ermöglichen mit dem Zweck der Berechnung neuer, expliziter Fakten mithilfe von Schlussfolgerungsprozessen.

1.2 Lösungsansatz

Der generellen Annahme folgend, dass ein grundlegendes, vielschichtiges Modell für eine erfolgreiche EI-Planungsverifikation erforderlich ist, besteht die Notwendigkeit, dieses Modell einem modularen Ansatz entsprechend zu entwickeln. Dieser Ansatz reflektiert die unterschiedlichen Aspekte des EI-Planungsprozesses. Folgende Schichten sind hierzu zu beachten:

- Eine Abstraktion des Netzes von EI-Komponenten als ein Graph erschließt sich als sinnvoll und zielführend. Der Graph sollte unabhängig von der EI-Domäne formuliert sein.

- Eine Formalisierung der EI-Domäne ist notwendig, um Planungsverifikation zu betreiben. Hierzu scheint es ratsam, ein eigenständiges, in sich schlüssiges Domänenmodell zu erstellen, welches die EI-Konzepte, deren Eigenschaften und Relationen beinhaltet.

- Da unterschiedliche Sätze von Planungsrichtlinien die EI-Planung beeinflussen, ist es zielführend, diese Sätze auszugliedern und nicht direkt in das Domänenmodell zu integrieren.

- Allgemeingültig ist die pragmatische Strategie, eine Trennung zwischen terminologischen Informationen auf der Begriffsebene und Informationen über existente Fakten auf der Instanzebene zu verfolgen. Demzufolge ist die konkrete zu verifizierende EI als eigenständiges Modul zu betrachten, welches übergeordnet auf sämtliche vorgenannte Komponenten aufsetzt, somit das Domänenmodell nutzt und den spezifischen Planungsrichtlinien entsprechend verifiziert werden kann.

Grundsätzlich wird in der vorliegenden Arbeit die Anwendbarkeit von semantischen Technologien zur Verifikation von EI-Planungsdaten untersucht. Hierzu mussten die Konzepte und Zusammenhänge von EI-Elementen in ein formales semantisches Modell übertragen werden. Die Vorteile dieses Ansatzes werden erkennbar, wenn man sie mit anderen Modellierungstechniken vergleicht. Möglichkeiten zur formalisierten Beschreibung natürlichsprachlicher Planungsrichtlinien, ohne deren Aussagen und Informationsgehalte zu verfälschen, stellt eine weitere Anforderung dar. Ferner ist die Verknüpfung des semantischen Modells und der formalen Planungsrichtlinien eine Tätigkeit, die im Hinblick auf den Prozess der Verifikation untersucht werden muss. Außerdem ist es meist ein Problem, Fehler in den Planungsdaten präzise zu lokalisieren. Auch hierzu präsentiert die vorliegende Arbeit einen Vorschlag für ein Verfahren.

1.3 Stand der Technik

In diesem Abschnitt zum aktuellen Stand der Technik werden Technologien erwähnt, die in Kapitel 2 der vorliegenden Arbeit detaillierter erläutert werden. Grundsätzlich wird zwischen einem organisatorischen und einem wissenschaftlichen Bereich unterschieden.

1.3.1 Organisatorisch

An der Planung von EI sind eine Vielzahl Akteuren beteiligt. Neben den Auftraggebern, wie vornehmlich der Deutschen Bahn, sind Ingenieurbüros, Tiefbauunternehmen, Signalbauunternehmen sowie Firmen zur Erstellung der Leitsysteme beteiligt. Des Weiteren steht die Kontrollinstanz, das Eisenbahn Bundesamt (Abk.: EBA) in der Verantwortung die Ausführung gemäß den Planungsdaten zu genehmigen. Da diese heterogenen Partner Planungsdaten unterschiedlichster Granularität austauschen müssen und über keine einheitliche computergestützte Toollandschaft verfügen, erfolgt der Austausch der Planungsunterlagen häufig in Papierform. Die Abbildung 1.1 stellt auszugsweise den Datenaustausch der an der Planung beteiligten Partner dar. Hierbei kennzeichnen die violetten Pfeile eine nichttoolgestützte Planung und die dunklen Linien einen manuellen Datenaustausch. Lediglich in wenigen Phasen des Planungsprozesses kommen Softwareprogramme zur Anwendung, und auch dann erfolgt meist kein elektronischer Datenfluss. Die wenigen roten Linien im Schaubild weisen auf einen geringen digitalen Datenaustausch im Planungsprozess hin. Diese Abbildung verdeutlicht, dass nur eine limitierte elektronische Datenverarbeitung im Planungsprozess vorhanden ist und ebenso nur eine geringe Möglichkeit zur computergestützten Verifikation von EI-Planungsdaten besteht.

Generell existieren Ansätze für die Verwendung formaler Modelle zur Beschreibung und Überprüfung von EI. In der Industrie wird mittlerweile ein Ansatz erprobt, XML (vgl. Kapitel 2.2.1) als Datenaustauschformat zu nutzen, um Planungsdaten verschiedener Beteiligter zusammenzuführen und in einer Toollandschaft nutzbar zu machen. Innerhalb der Eisenbahn-Industrie existiert jedoch kein Konsens über ein einheitliches Format. In der Planungs- und Erprobungsphase befindet sich ein von der Deutschen Bahn favorisiertes XML-Format mit einem zugrundeliegenden Schema [54]. Jedoch kann mithilfe eines XML-Schemas keine weitreichende Verifikation von Datenabhängigkeiten sondern lediglich eine Validierung auf syntaktischer Ebene vollzogen werden.

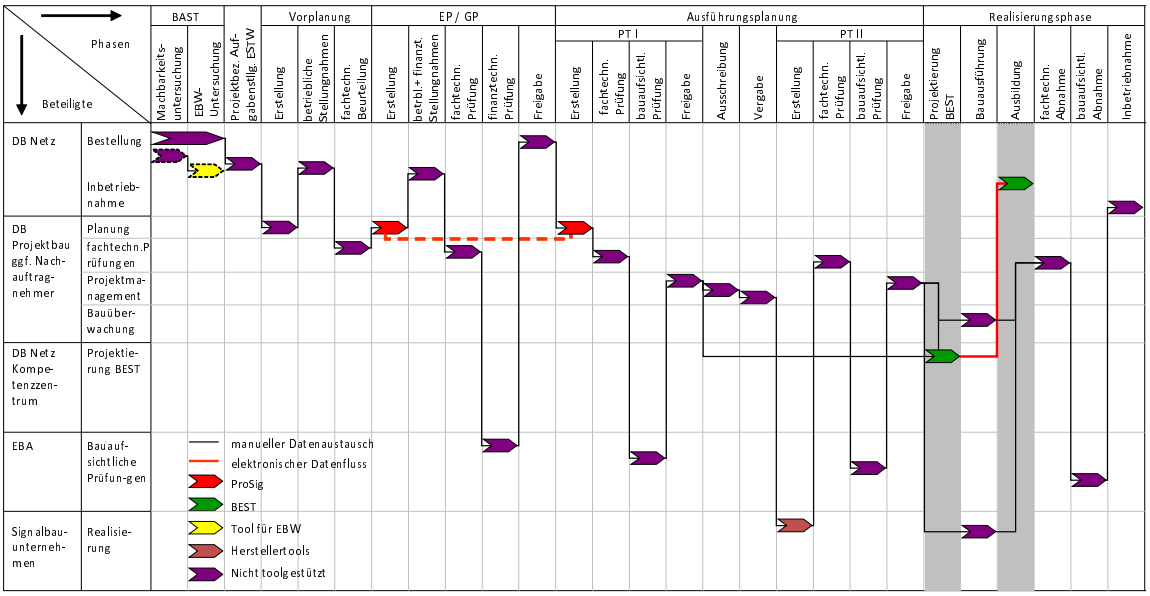

Abbildung 1.1: Planungsprozess von Eisenbahn-Infrastrukturen

1.3.2 Wissenschaftlich

1.3.2.1 Verifikation mit XML-Schema und Schematron

Bei der Planung von EI werden bereits elektronische Datenformate eingesetzt. Neben der Speicherung konventioneller Tabellen oder Dokumente von Textverarbeitungsprogrammen kommen in diesen Bereichen auch zunehmend XML-Formate zum Einsatz. Hieraus ergeben sich viele Vorteile. So ist es mit XML nicht nur möglich, Daten in Beziehung zu setzen und dementsprechend zu strukturieren, sondern auch eine formale automatisierte Verarbeitung dieser Daten wird realisiert. Der wesentliche Vorteil bei der Verwendung von XML-Dokumenten, welche EI-Planungsdaten repräsentieren, ist die Möglichkeit des Einsatzes eines XML-Schemas, nach dem diese Dokumente strukturiert werden müssen. In Kapitel 2.2.1 wird ausführlich beschrieben, welcher Nutzen aus der Verwendung von XML-Schemata gezogen werden kann. Die Struktur eines XML-Dokuments kann durch die Verwendung von XML-Schemata validiert werden.

Die Validierung ist jedoch nur eine syntaktische Prüfung der Daten. Eine Prüfung der Semantik, also des Sinns der Dateninhalte und ihrer Schlüssigkeit auch in wechselseitigen Beziehungen, ist mit XML-Schema nicht möglich. Für semantische Prüfungen ist also ein mächtigeres Ausdrucksformat als pures XML und XML-Schema nötig. Diese

semantische Prüfung ist jedoch notwendig, um einen ganzheitlichen, automatisierten Verifikationsprozess zu ermöglichen.

Es existiert eine open-source-Initiative rund um das railML-Schema [39]. Hierbei handelt es sich ebenfalls um ein XML-Schema, welches verschiedene Aspekte der Eisenbahntechnik abzudecken versucht. Neben der Infrastruktur-Beschreibung können in den unterschiedlichen Sub-Schemata auch Zugkonstellationen und Fahrpläne beschrieben werden. Ein fokussiertes Sub-Schema für die Beschreibung der Leit- und Sicherheitstechnik aus Stellwerkssicht fehlt jedoch. Es wird versucht, diese Defizite mit Erweiterungen des Infrastruktur-Sub-Schemas auszugleichen. In der Forschung und im universitären Umfeld wird railML häufig eingesetzt, spielt jedoch im industriellen Kontext insbesondere bezogen auf das Infrastruktur-Sub-Schema nur eine untergeordnete Rolle. Von Seiten der Deutschen Bahn erfährt railML kaum Unterstützung, da diese augenscheinlich ihr eigenes Format fördern möchte. Lediglich in der Schweiz existiert im industriellen Umfeld ein breite Akzeptanz von railML insbesondere im Bereich der Fahrplan- und Dispositionsplanung [35]. Eine Anwendung des railML-Schemas zur Verifikation von EI wurde von dem Autor dieser Arbeit untersucht und in der Veröffentlichung [31] beschrieben.

Obwohl es sich bei railML auch nur um ein XML-Schema handelt und die Verbreitung im industriellen Kontext in Deutschland nicht außerordentlich hoch ist, wird es in der vorliegenden Arbeit erwähnt, weil es Bestrebungen gibt, über die rein syntaktische Validierung auch eine Verifikation von Datenabhängigkeiten mithilfe von Schematron (vgl. Kapitel 2.2.3) zu ermöglichen. Hierzu existieren bereits Ansätze aus dem railML-Umfeld. Zu nennen sei an dieser Stelle eine Diplomarbeit [59], in der vornehmlich beschrieben wird, wie Schematron dazu genutzt werden kann, Dateninhalte einzuschränken. Hierbei lassen sich allerdings viele Schematron-Patterns auch in XML abbilden. In einigen Beispielen wird auf die Stärke Schematrons eingegangen, einfache Datenabhängigkeiten zu überprüfen. Dies wird jedoch nur exemplarisch erläutert. Als Beispiel wird hierzu die fehlerhafte Verwendung der *voltage* und *frequency* Attribute eines railML-Elements in Kombination mit Werten des *powerType*-Attributs genannt, die auf nicht-elektrische Spannungsversorgung eines eisenbahntechnischen Triebfahrzeugs hindeuten (beispielsweise 'diesel'). Die Verifikation komplexer Datenabhängigkeiten wird in dieser Arbeit nicht erläutert. Des Weiteren bezieht sich die Arbeit nur auszugsweise auf EI. Konkrete Planungsrichtlinien werden ebenso wenig erläutert und für Verifikationszwecke untersucht.

1.3.2.2 Verifikation mit SPARQL-Frameworks

Grundsätzlich könnten auch andere Ansätze als der in dieser Arbeit beschriebene verfolgt werden. So existieren Ansätze zur Datenverifikation mithilfe der Regelsprache SPARQL [48]. Die kommerzielle Entwicklungsumgebung *TopBraid Composer* [53] setzt der Eclipse-IDE [17] auf und ermöglicht die Bearbeitung von RDF- und OWL-Ontologien. Des Weiteren verwendet es eine Implementierung der W3C Member Submission *SPIN - Modeling Vocabulary* [27]. Im Wesentlichen handelt es sich hierbei um ein SPARQL-Framework zur Definition von Regeln und logischen Bedingungen an die Wissensbasis.

Eine weitere Variante des Verifizierens mit SPARQL-basierten Regeln und Bedingungen ist die auf der Arbeit [51] aufsetzende Implementierungsvariante des Pellet Reasoners [52]. Bei beiden Varianten der SPARQL-Verifikation wurde das SPARQL-Vokabular um einige Schlüsselwörter erweitert, um den Anforderungen an Datenverifikationen zu entsprechen.

Beide Ansätze stellen interessante Varianten dar, um die Überprüfung von Datenkorrektheit im Sinne der CWA zu ermöglichen. Da jedoch beide Varianten SPARQL nutzen, ist die gemeinsame Grundlage RDF. Zwar sind auch OWL-Ontologien in RDF/XML serialisierbar, die Ausdrucksstärke von OWL ist jedoch höher als bei RDF. So können beispielsweise keine Transitivitäten in RDF ausgedrückt und mit SPARQL verifiziert werden. Dies ist jedoch eine elementare Anforderung an ein Verifikationssystem für Eisenbahn-Infrastrukturen, in dem Verbindungseigenschaften zwischen einzelnen Elementen, aber auch über eine gesamte Zugstrecke überprüft werden müssen. Diese Verbindungen sind idealerweise über transitive Beziehungen ausgedrückt und können mit der Regelsprache SWRL verifiziert werden, da diese die OWL-Semantik inklusive der Transitivitätsbeziehungen verarbeiten kann.

1.3.2.3 Weitere Ansätze zur semantischen Verifikation im Eisenbahnsektor

Im Zuge der Recherchen zu dieser Arbeit wurden weitere Beispiele zur Verifikation von Eisenbahntechnologie mithilfe semantischer Technologien identifiziert. Ein prägnantes Beispiel hierfür ist das von der EU geförderte InteGRail-Projekt [2]. In diesem Projekt ist die Verwendung von Ontologien zur Modellierung von Fehlerszenarien vornehmlich in Triebfahrzeugen während des laufenden Betriebs analysiert worden [5]. Ziel des Projekts ist eine Steigerung der Wartungseffizienz der Triebfahrzeugstechnik durch semantifizierte Vorhersagemodelle. Die gleisseitige Eisenbahntechnik sowie die Planung der Leit- und Sicherheitstechnik ist nicht Gegenstand dieses Projekts.

Des Weiteren beschreiben Mohan und Arumagam in [38] den Aufbau einer Eisenbahn-Ontology in OWL und deren Verifikation mit SWRL. Diese Arbeit bezieht sich auf die Modellierung von verschiedenen Triebfahrzeugtypen, Zugkonstellationen und eine Fahrplanplanung. Auch in dieser Arbeit ist die Planung von Eisenbahn-Infrastrukturen nicht Forschungsgegenstand.

1.4 Eigener Beitrag

Da EI im Wesentlichen aus einem Graphen bestehen, liegt es nahe, die EI-Verifikation als Graphenproblem zu behandeln und dementsprechende Algorithmen und Tools einzusetzen. Jedoch greift die Sicht auf EI als Graphen zu kurz, da die 'Anhänge' eines EI-Graphen, wie z.B. Signale, Gleismagneten etc. in komplexen Wechselseitigen Beziehungen stehen und somit nicht durch graphentheoretische Ansätze überprüft werden können. Ein weiteres Ziel dieser Arbeit ist es, Planungsrichtlinien unter Beibehaltung von Verständlichkeit und Nachvollziehbarkeit zu formalisieren. Dies ist durch die Nutzung eines Domänenmodells (*RI* Core Ontology*) gewährleistet, welches eisenbahntechnische Konzepte und ihre Relationen mit gewohnten Bezeichnungen sowie der intuitiv nutzbaren Regelsprache SWRL (vgl. *RI* Rule Ontology* in Kapitel 4.4) verwendet.

Bezüglich des Ontologie-Debuggings existieren bereits mehrere Arbeiten. Grundsätzlich ermöglichen bereits OWL/SWRL-Reasoner die Aufdeckung von Inkonsistenzen in einer ontologischen Wissensbasis und zeigen deren Ursachen auf. Jedoch ist eine Überprüfung der Datenintegrität mithilfe dieses Verfahrens nicht möglich. In [44] erläutern die Entwickler des OWL-Reasoners *Pellet* [1], wie sie exemplarisch Debugging Hinweise, die sie aus Pellet abgeleitet haben, in den SWOOP Ontologie Editor integriert haben. Es fand jedoch keine Weiterentwicklung dieses Editors seit 2007 statt.

In [50] hat Heiner Stuckenschmidt den formalen Hintergrund des Ontologie-Debuggings beleuchtet. Begleitet durch praktische Versuche hat er den Schluss gezogen, dass in der Theorie vieles bezüglich dieser Thematik untersucht wurde, jedoch die praktische Umsetzung in Form von implementierter Software im Rückstand ist.

Im Gegensatz zu den oben beschriebenen Ansätzen, die sich auf die Überprüfung der TBox von Ontologien fokussieren, setzt der im Rahmen dieser Arbeit entwickelte praktische Ansatz des Ontologie-Debuggings mit Semantic Constraints (vgl. Kapitel 5) den Fokus auf die Überprüfung der Datenintegrität der ABox.

Im Rahmen dieser Arbeit wurden semantische Technologien dahingehend untersucht, ob und wie sie sich dafür eignen, den Planungsprozess neuer Eisenbahn-Infrastrukturen zu optimieren. Es zeigte sich, dass OWL und SWRL gerade in dem Gebiet der Verifikation sinnvoll eingesetzt werden können und ihre Ausdrucksstärke dazu genutzt werden kann, auch außerhalb des Umfeldes des Semantic Web praktische Verwendung zu finden. OWL und SWRL sind aufgrund ihres Fundaments - der Beschreibungslogik - und des damit verbundenen logischen Schließens nützliche Werkzeuge für die Bewältigung einer Reihe von komplexen und rechenintensiven Problemen - so auch dem Verifizieren von Eisenbahn-Infrastrukturen.

Die vorliegende Arbeit beschreibt, wie diese Technologien in einem exotischen Anwendungsgebiet sinnvoll verwendet werden können. Es wird dargestellt, wie Eisenbahn-Infrastrukturen bzw. deren Planungsdaten automatisch zu verifizieren sind. Außerdem werden Vorgehensweisen aufgezeigt, wie die Domäne der EI formal modelliert und dieses Modell für Verifikationszwecke genutzt werden kann. Dabei wird nicht nur die Abbildung der real-weltlichen Elemente beachtet, sondern ebenso die Bedingungen und Auflagen an deren konkrete Planung berücksichtigt (vgl. Veröffentlichungen [32] und [33]).

Ein weiterer Beitrag dieser Arbeit ist die Beschreibung eines Verfahrens des Debuggings von Ontologien. Hierbei wird nicht wie in anderen Ansätzen das Vorgehen geschildert, wie Fehler, die zu einer inkonsistenten Wissensbasis führen, aufgezeigt werden. Vielmehr wird ein Verfahren vorgestellt, mit dem Bedingungen an die Dateninhalte formuliert und konkrete Verstöße gegen diese Bedingungen identifiziert werden können (vgl. Veröffentlichung [34]).

1.5 Überblick über die vorliegende Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert: Nach dieser Einleitung wird in Kapitel 2 auf die Details der verschiedenen Technologien eingegangen. Zum einen werden die verschiedenen Elemente von Eisenbahn-Infrastrukturen erläutert und in Beziehung zueinander gesetzt. Zum anderen erfolgt eine Erläuterung der Technologien, mit denen Eisenbahn-Infrastrukturen modelliert und verifiziert werden. Hierbei erfolgt eine kurze Einführung in die Basistechnologie XML Schema sowie eine Technologie zur weiterführenden Überprüfung von XML-Dokumenten. Anschließend werden die Ontologiebeschreibungssprache OWL sowie die darauf aufsetzende Regelsprache SWRL erläutert. Diese beiden semantischen Technologien zählen zu den Kerntechnologien des Semantic Web.

In Kapitel 3 werden die beiden Grundkonzepte dieser Arbeit diskutiert. Sowohl der Ansatz der semantischen Verifikation als auch das Ontologie-Debugging mit Semantic Constraints sind in diesem Kapitel skizziert. Es wird erläutert, wie ein Weg von einer Überprüfung auf struktureller Ebene hin zu einer Überprüfung auf bedeutungsbezogener Ebene in Bezug auf Eisenbahn-Infrastrukturen beschrieben werden kann.

Die semantische Verifikation ist Gegenstand des Kapitels 4. In diesem Kapitel werden sowohl die Struktur des semantischen Modells als auch dessen Verifikation erläutert. Das semantische Modell verwendet die Technologien aus dem Grundlagenkapitel und beinhaltet die modellierten Eisenbahn-Infrastrukturelemente, die ebenfalls im Grundlagenkapitel 2 beschrieben wurden.

Der Ansatz des Ontologie-Debuggings mit Semantic Constraints wird in Kapitel 5 beschrieben. Hier wird ausführlich auf den Aufbau des Systems und die Algorithmen eingegangen.

Eine Bewertung der gezeigten Ansätze erfolgt in Kapitel 6. Die Arbeit wird mit einem Ausblick in Kapitel 7 abgeschlossen.

2 Technologische Grundlagen aus verschiedenen Domänen

Die vorliegende Arbeit beschäftigt sich mit Technologien unterschiedlicher Bereiche. Zum einen trägt ein Einblick in Eisenbahn-Infrastrukturen wesentlich zum Verständnis dieser Arbeit bei, zum anderen sind Modellierungstechnologien ein fundamentaler Bestandteil. In den folgenden Kapiteln werden diese unterschiedlichen Technologiebereiche und ihre Facetten erläutert.

2.1 Eisenbahn-Infrastrukturen

2.1.1 Erläuternde Abgrenzung

Unter Eisenbahn-Infrastrukturen sind sämtliche physische und virtuelle Komponenten eines Eisenbahnsystems zu verstehen, die dazu benötigt werden, eine (sichere) Bewegung von Zügen zu ermöglichen. Das impliziert, dass eben diese sich auf den Schienen bewegenden Triebfahrzeuge und Waggons explizit nicht zu einer Eisenbahn-Infrastruktur gezählt werden.

Zu den physischen Komponenten einer Eisenbahn-Infrastruktur werden beispielsweise die Schienen und Weichen gezählt, über die sich Züge bewegen können. Ebenso dazu gezählt werden sicherheitstechnische Komponenten wie Signale, Bahnübergänge sowie Komponenten zur Zugbeeinflussung und -überwachung.

Zu den virtuellen oder logischen Elementen einer Eisenbahn-Infrastruktur zählen beispielsweise Fahrstraßen. Dieses sind sicherheitstechnisch abgegrenzte Streckenabschnitte auf einem Schienennetz, die von Zügen benutzt werden, um von einem bestimmten Start- zu einem bestimmten Zielpunkt zu gelangen. Das grundsätzliche Sicherheitskonzept im Schienenverkehr gibt vor, dass solche Fahrstraßen immer jeweils für nur einen Zug reserviert sind. Fahrstraßen sind aus Stellwerkssicht gesicherte

Wegstrecken. Diese sind aus den sicherheitstechnisch kleinsten Einheiten, den sogenannt Blockabschnitten zusammengesetzt. Sie beschreiben einen Sicherungsabschnitt eine Zugstrecke und werden von Achszählern begrenzt, die detektieren, ob sich ein Zug in ihnen befindet.

In einem Stellwerk werden Fahrstraßen an zentraler Stelle für die jeweiligen Züge reserviert. Stellwerke ermöglichen die zentrale Erfassung und Steuerung sämtlicher Zugbewegungen einer Strecke. Auch wenn Stellwerke eine zentrale Komponente von Eisenbahnsystemen darstellen, werden sie nicht direkt zu Eisenbahn-Infrastrukturen gezählt. Sie sind diesen übergeordnet und in der Regel auch nicht an den entsprechenden Eisenbahnstrecken positioniert. Meist werden eine Vielzahl von Stecken über ein zentrales Stellwerk gesteuert.

In den nächsten Kapiteln wird detailliert auf die einzelnen Komponenten von Eisenbahn-Infrastrukturen eingegangen.

2.1.2 Übergeordnetes Element: Stellwerk

2.1.2.1 Funktionen eines Stellwerks

Stellwerke gewährleisten die sicherungstechnische Steuerung einer Eisenbahnstrecke. Sie sind die Schaltzentralen sämtlicher fest installierter Streckenkomponenten. Die Hauptaufgabe der in den Stellwerken arbeitenden Fahrdienstleitern liegt darin, die Zugbewegungen zu überwachen und zu steuern. Die Steuerung der Züge erfolgt in indirekter Weise über die Freigabe und Sperrung einzelner Fahrstraßen. Die Abbildung 2.1 zeigt die schematische Bedienoberfläche eines Stellwerks. Fahrstraßen sind aus einzelnen Blockabschnitten (graue, gelbe und rote Segmente in der Abbildung) zusammengesetzt. Diese Einheiten werden durch Gleisfreimeldeeinrichtungen wie z.B. Achszähler überwacht und können somit an das Stellwerk Rückmeldungen über besetzte Gleise versenden. Die ganze Sicherungstechnik ist darauf ausgelegt, die Exklusivnutzung der Fahrstraßen (rote Segmente in der Abbildung) zu gewährleisten. Ein- und Ausfahrten der Fahrstraßen werden in der Regel durch Signale begrenzt (Abbildung: Startsignal/Endsignal). Im Szenario mit der höchsten Sicherheitsstufe sind die Signalsysteme um Gleismagneten (siehe Kapitel 2.1.3.7) an den Schienen ergänzt (Nicht in der Abbildung enthalten). Diese bremsen Züge aktiv ab, wenn diese ein *Halt*-zeigendes Signal missachten. Somit ist gewährleistet, dass keine weiteren Züge unkontrolliert in eine bereits reservierte und in Anspruch genommene Fahrstraße einfahren und dort eine Gefährdungssituation erzeugen können. Nicht nur die direkten Ein- und Ausfahrten von Fahrstraßen sind potentiell gefährlich und des-

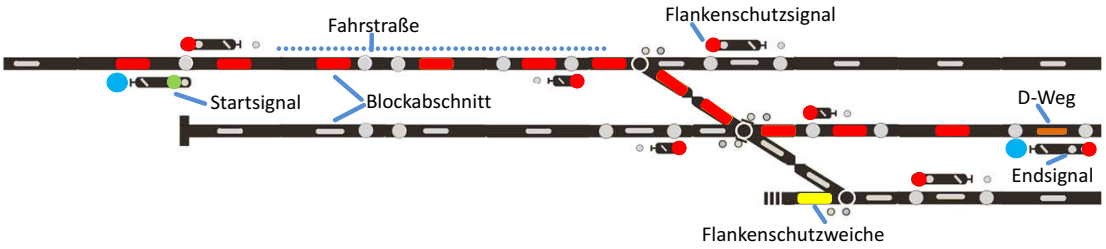

Abbildung 2.1: Schematische Darstellung der Bedienoberfläche eines Stellwerks

halb abzusichern. Auch die „Seiteneinfahrten“ in eine Fahrstraße, also Bereiche mit Weichen, die eine Aufteilung bzw. Zusammenführung von Zugstrecken ermöglichen, stellen eine Gefahr ungewollter Zugkollisionen dar. Diese sogenannten *Flanken* sind ebenso beim Einrichten von Fahrstraßen, wie die regulären Ein- und Ausfahrten, abzusichern. Diese Absicherung erfolgt durch Flankenschutzelemente. Diese bestehen aus Signalen, die *Halt* anzeigen, oder Weichen, die in die der Fahrstraße abweisende Richtung geschaltet sind.

Um in einem modernen Stellwerk eine Fahrstraße einzurichten, genügt es in der Regel, dass der Fahrdienstleiter den Anfang und das Ende der Fahrstraße auf dem Monitor oder der Schalttafel - je nach Stellwerktyp - auswählt (blaue Punkte in der Abbildung 2.1). Daraufhin wird die Fahrstraße automatisch eingestellt. Die darunterliegende Stellwerkstechnik ist jedoch um ein Vielfaches komplexer. Sie muss sicherstellen, dass alle an der Bildung der Fahrstraße beteiligten Komponenten den gewünschten Zustand einnehmen. Meist ist auch die Einhaltung einer zeitlichen Abfolge zu gewährleisten. So muss beispielsweise natürlich erst der Bahnübergang gesichert sein und die Schranken für Fußgänger und Autos müssen heruntergefahren und eingerastet sein, bevor das Zugsignal an der Einfahrt der Fahrstraße Weiterfahrtserlaubnis signalisiert und die Fahrstraße letztlich freigibt. Die Einhaltung dieser sogenannten Signalabhängigkeit ist zentrale Aufgabe eines Stellwerks.

2.1.2.2 Elektronisches Stellwerk

In der Vergangenheit wurden eine Vielzahl von Stellwerkstypen eingesetzt. Erst im Zuge der Entwicklung von Computern erfolgte der Einsatz elektronischer Stellwerke (ESTW). In Deutschland ging Anfang der 80-er Jahre das erste elektronischer Stellwerk in Betrieb. Ein Charakteristikum von ESTWs ist die räumliche Unabhängigkeit der Stellwerke von den zu steuernden Außenanlagen. Die Nutzungsmöglichkeiten von Computersystemen ermöglichen einen Standort, der weit entfernt von der zu steuernden Bahnstrecke gelegen sein kann. Dies war aufgrund der meist mechanischen

Funktionsweise voriger Stellwerkstypen in der Vergangenheit nicht möglich.

Im ESTW werden die Stellbefehle an einem Computer vom Fahrdienstleiter erzeugt. Die Bedienung erfolgt meist mit Maus und Tastatur. Zum Einstellen einer Fahrstraße klickt der Fahrdienstleiter auf das Start- und Endsignal einer Fahrstraße. Die Stellbefehle und Rückmeldungen der Außenanlage werden von mindestens zwei unabhängigen Rechnern erfasst, verarbeitet und an einen weiteren Rechner, den sogenannten Vergleicher, weitergeleitet. Erst wenn im Vergleicher dieselben Zustandsabbildungen festgestellt werden, kann ein Stellvorgang oder eine Bildschirmausgabe erfolgen. Dieser Aufwand wird betrieben, weil Nachweise für den sicheren Betrieb von Softwaresystemen sehr schwer zu erbringen sind. Da Stellwerke jedoch zu den sicherheitskritischen Einrichtungen zählen, sind diese Nachweise zwingend erforderlich. Bei älteren Stellwerkstypen wie z.B. den Relaisstellwerken waren diese Nachweise wesentlich einfacher zu erbringen, da Relais bei einer Fehlfunktion physikalisch bedingt in einen definierten Zustand fallen und somit bspw. bewirken, dass ein Signal bei einem Stromausfall immer *Halt* signalisiert. Aus diesem Grunde haben die Außenanlagen moderner ESTW trotzdem meist eine Relaisschaltung mit Kondensatoren, die ständig von den Stellrechnern mit Strom versorgt werden müssen. Fällt der Strom aus, entladen sich die Kondensatoren und das entsprechende Signal fällt ebenfalls in einen definierten, sicheren Zustand.

2.1.3 Physische Elemente einer Eisenbahn-Infrastruktur

2.1.3.1 Gleise

Züge werden ausschließlich auf einem Schienennetz bewegt. Dieses wird physikalisch gesehen im Groben in Gleise und Weichen unterteilt. Gleise werden weiter unterteilt in Gleisabschnitte. Spurbreite, Neigung und Steigung sind grundlegende physikalische Eigenschaften von Gleisen. Da diese Arbeit sich eher mit der Verifikation von Eisenbahn-Infrastrukturen aus Stellwerkssicht und damit mit den logischen Zusammenhängen auseinandersetzt, werden diese jedoch nicht fokussiert.

Wie schon beschrieben ist das Schienennetz einer Eisenbahnstrecke in Abschnitte unterteilt. Aus sicherheitstechnischer Stellwerkssicht wird jedoch der Begriff *Gleisabschnitt* vermieden. Im Stellwerk wird der Begriff *Blockabschnitt* oder *Freimeldeabschnitt* verwendet. In Kapitel 2.1.4.1 wird auf diese Infrastrukturkomponente näher eingegangen.

2.1.3.2 Signal

Abbildung 2.2: Eisenbahnsignale - Bildquelle: flickr.com / CC-BY: Eisenbahner

Im Zugverkehr fällt Signalen eine Vielzahl an Aufgaben zu. Neben dem Signalisieren von Halte- und Fahrtbefehlen, existieren beispielsweise auch Signale zum Anzeigen von Bahnübergängen, Höchstgeschwindigkeiten, Notverhalten etc. Signale können in zwei Gruppen unterteilt werden. Veränderliche- und unveränderliche Signale. Unveränderliche Signale sind meist Blechschilder mit einem festen sogenannten Signalbegriff. Sie sind somit nicht steuerbar. Veränderliche Signale können gesteuert werden und unterschiedliche Signalbegriffe anzeigen. Diese können entweder durch Veränderung der Lichtfarben oder -anordnungen (Lichtsignal) oder bei älteren Signalsystemen durch die Stellung und Ausrichtung des Signals selber (Formsignal) gesteuert. Grundlegend wird, bezogen auf das Ermöglichen einer Zugbewegung, zwischen Haupt- und Vorsignalen unterschieden. Ein Vorsignal steht immer, dem Namen entsprechend, vor einem Hauptsignal. Es zeigt in der Regel dasselbe Signalbild, welches auch vom Hauptsignal angezeigt wird und stellt somit eine Vorabinformation dar, worauf sich der Triebfahrzeugführer am Hauptsignal einzustellen und dementsprechend zu handeln hat. Hauptsignale sichern in der Regel einen Streckenabschnitt (einen sogenannten Streckenblock - siehe Kapitel 2.1.4.1) ab. Streckenblöcke sind stellwerkstechnische Einheiten, die immer nur von einem Zug besetzt werden dürfen.

Ein weiteres Unterscheidungsmerkmal von Signalen ist das zugrundeliegende Signalsystem. In Deutschland wurden und werden unterschiedliche Signalsysteme genutzt, die im Folgenden kurz erläutert werden.

- **H/V-Signalsystem** - Haupt-Vorsignal-System. Dieses ist das älteste Signal-

systems Deutschlands. Es besteht ursprünglich aus Formsignalen. In einer Weiterentwicklung sind auch Lichtsignale hinzugekommen, bzw. haben die alten Formsignale abgelöst. In der Abbildung 2.2 sind Formsignale des H/V-Systems dargestellt. Aus der Stellung der Formsignale kann ein Triebfahrzeugführer den Signalbegriff ablesen. So zeigt beispielsweise das rechte Signal dem herannahenden Zug den Signalbegriff HP-2 (Langsamfahrt), während das linke Signal den HP-0 (Halt) für entgegenkommende Züge anzeigt.

- **Hl-Signalsystem** - Das Hl-System wurde von der Deutschen Reichsbahn in der ehemaligen DDR ab 1959 eingesetzt. Auf einer einheitlichen Anzeigetafel können mit unterschiedlichen Lichtbildern alle wichtigen Signalbilder inklusive der vorgeschriebenen Geschwindigkeiten angezeigt werden. Formsignale kommen nicht mehr zum Einsatz. Ebenfalls charakteristisch für das Hl-Signalsystem ist die mögliche Kombination von Vor- und Hauptsignalen auf einer Anzeigetafel. In der DDR wurden auch ältere Stellwerkstypen in der Form nachgerüstet, dass sie das „moderne“ Hl-System unterstützen konnten.

- **Ks-Signalsystem** - Das Ks-Signalsystem (Ks = Kombinationssignal) ist das aktuelle Signalsystem Deutschlands. Bei Streckenneubau oder -modernisierungen werden in der Regel Sinale des Ks-Systems installiert. Dieses System wurde erstmals 1993 eingesetzt. Es entstand aus der Notwendigkeit, nach der Wiedervereinigung Deutschlands ein einheitliches Signalsystem zu etablieren, denn das Westdeutsche H/V-System war mit dem Ostdeutschen Hl-System nicht kompatibel. Das neu eingeführte Ks-System hingegen bedient sich beider Signalsysteme, ist zu ihnen kompatibel und ermöglicht die Anzeige von Signalbegriffen beider Systeme. Auch beim Ks-System ist es möglich, Vor- und Hauptsignal auf einem Signalschirm anzuzeigen.

Erprobt wurden noch weitere Signalsysteme. Jedoch haben diese sich nicht durchgesetzt bzw. werden nur in äußerst begrenzten Gebieten eingesetzt und finden aus diesem Grund in der vorliegenden Arbeit keine Erwähnung.

2.1.3.3 Weiche

Eine Eisenbahnweiche ist ein infrastrukturelles Element, das Verzweigungen innerhalb eines Gleisnetzes ermöglicht. Eine einfache Weiche verfügt über einen durchgehenden Gleisstrang (Stammgleis) mit einem Anfang (Spitze) und einem Ende sowie über einen abzweigenden Gleisstrang (Zweiggleis), der nur von oder über die Stammgleisspitze, nicht aber über das Stammgleisende befahren werden kann. Die bewegliche Weichenzunge im Herzstück der Weiche ist über ein Stellwerk steuerbar (teilweise auch manuell vor Ort). Die Weichenzunge beeinflusst die Fahrtrichtung, also ob ein beispielsweise über die Spitze herannahender Zug geradeaus über das Stammgleis oder abbiegend über das Zweiggleis geleitet wird.

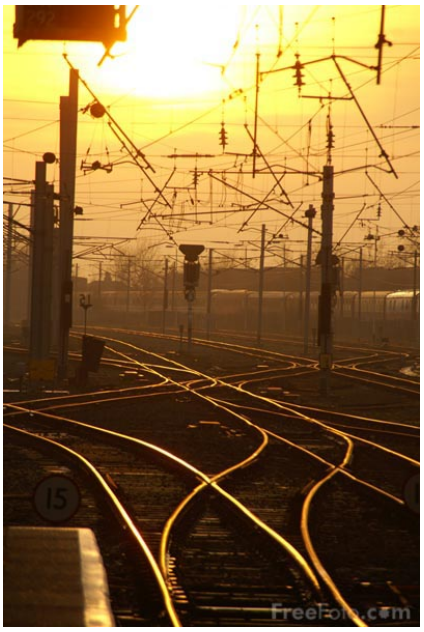

Abbildung 2.3: Eisenbahnweiche - Bildquelle: freefoto.com / CC-BY: Ian Britton

Es existieren mehrere Weichentypen. So wird grundsätzlich zwischen einfachen Weichen, Zweibogenweichen, Doppelweichen, Kreuzungsweichen und Doppelkreuzungsweichen unterschieden. Es sei darauf hingewiesen, dass innerhalb der grundlegenden Weichentypen noch weitere Unterscheidungen existieren, die im Rahmen dieser Arbeit jedoch nicht erläutert werden. Die Abbildung 2.3 zeigt im Vordergrund eine einfache Weiche, die jedoch ein nach außen gebogenes Stammgleis aufweist. Dieser Weichentyp wird als Außenbogenweiche oder Y-Weiche bezeichnet.

Aus Stellwerkssicht sind Weichen ein sicherheitskritisches Infrastrukturelement, da eine falsche Weichenlage evtl. dazu führen kann, dass ein Zug fälschlicherweise auf ein Gleis geleitet wird, auf dem sich bereits ein weiterer Zug in entgegenkommender Richtung nähert. Die Folge wäre ein Frontalzusammenstoß beider Züge, welcher als einer der schwersten Unfälle im Schienenverkehr mit gravierenden Folgen gewertet werden kann.

Für das Einstellen von Fahrstraßen im Stellwerk sind Weichen ebenfalls von großer Bedeutung. Bezogen auf Zugbewegungen bestehen Fahrstraßen aus einem Verbund von Gleisabschnitten und Weichen, die durch Signale und flankenschutzbietende Elemente abgesichert werden. Nicht nur für die Fahrstrecke innerhalb einer Fahrstraße sind Weichen relevant, sie werden wie auch Lichtsignale als Flankenschutzelemente

Abbildung 2.4: Bahnübergang (Großbritannien) - Bildquelle: flickr.com / CC-BY: Ingry the wingry

verwendet und in dieser Funktion in die der Fahrstraße entgegengesetzten Richtung geschaltet, um ein versehentliches Einfahren eines fremden Zuges in die reservierte Fahrstraße zu verhindern.

2.1.3.4 Prellbock

Prellböcke sind Gleisabschlusselemente. Sie werden in Kopfbahnhöfen, also nicht durchgängigen Bahnhöfen, bei Stumpfgleisen oder anderen Gleisenden, beispielsweise in Rangierbereichen, dazu verwendet, die Gleise so zu sichern, dass ein Zug nicht die Schienen verlassen und entgleisen kann.

Es gibt unterschiedliche Bauarten von Prellböcken. Neben den einfachen statischen Prellböcken, die nicht vom Zug verschiebbar sind und über keine Abbremsfunktion verfügen, existieren Bauformen, die beispielsweise über eine hydraulische Pufferung verfügen. In Kopfbahnhöhen kommen meist verschiebbare Prellböcke mit einer Bremse, bestehend aus einer Konstruktion aus Metalldämpfern, zum Einsatz. Prellböcke werden einem Triebfahrzeugführer in Deutschland in der Regel durch entsprechende Signale angezeigt

2.1.3.5 Bahnübergang

Bahnübergänge sind Zugstreckenelemente, die eine Kreuzung mit anderen Verkehrsstrecken ermöglichen und aus diesem Grund einer besonderen Absicherung bedürfen. Es existieren unterschiedliche Arten von Bahnübergängen, wie beispielsweise durch Schranken und/oder Lichtsignal gesicherte, oder solche, bei denen die Sicherung lediglich durch Schilder (Andreaskreuz) erfolgt. Zusätzlich zu den optischen Sicherungen sind an Bahnübergängen meist akustische Warnsignale installiert. In Deutschland ist es gesetzlich vorgeschrieben, Bahnübergänge zu sichern. Eine Beschilderung ist bei jedem Bahnübergang erforderlich. Ist nur die Beschilderung vorhanden und keine weiteren Sicherungseinrichtungen, so wird von einem „nicht-technisch gesicherten“ Bahnübergang gesprochen. Auch bei der Beschrankung existieren Unterschiede. Weit verbreitet sind Halbschranken, die lediglich den Verkehr in Fahrtrichtung über den Bahnübergang zurückhalten. Eine Vollbeschrankung ist nur in wenigen Orten Deutschlands vorhanden. Dort muss zusätzlich eine optische Überwachung durch einen Menschen, entweder vor Ort oder an einem Monitor erfolgen, da die Möglichkeit besteht, dass Fahrzeuge oder Personen auf den Gleisen innerhalb der geschlossenen Schranken eingesperrt werden könnten. Dies ist bei Halbschranken nicht der Fall, da die Schienen auch bei herunter gelassenen Halbschranken ohne Probleme verlassen werden können. Gesondert zur Beschrankung für den Autoverkehr ist in der Regel noch eine Beschrankung für den Fußgängerverkehr vorhanden. Welche Sicherung des Bahnübergangs installiert sein muss, wird durch die Stärke des Verkehrsaufkommens an der entsprechenden Stelle vorgegeben.

Zur Auslösung der Sicherungssysteme eines Bahnübergangs bei einem herannahenden Zug existieren mehrere Varianten. Zum einen können sie explizit durch einen Steuerbefehl, beispielsweise dem Einstellen einer Fahrstraße in einem Stellwerk, ausgelöst werden. Zum anderen können auch Schienenkontakte eine Zugpräsenz detektieren und die Auslösung der Sicherungssysteme des entsprechenden Bahnübergangs bewirken.

Da es in Bereichen von Bahnübergängen immer wieder zu schweren Unfällen kommt, besteht die Tendenz zu ihrer kontinuierlichen Reduktion. Beim Streckenneubau wird nur in seltenen Fällen ein neuer Bahnübergang genehmigt. In der Regel wird versucht, den Zugverkehr durch Über- oder Unterführungen vom Straßen- und Personenverkehr zu separieren. Auch bei der Modernisierung von Eisenbahnstrecken wird häufig dieser Separationsansatz gewählt. In urbanen Gegenden, deren Standortvorteil zum Teil auf der Anbindung an das Schienennetz beruht, sind solche Umbauten jedoch meist nicht wirtschaftlich.

Abbildung 2.5: Achszähler (Schweiz) - Bildquelle: flickr.com / CC-BY: Kecko

2.1.3.6 Zugüberwachungskomponente

Um Zugfahrten auf einem Gleis zu ermitteln, werden technische Komponenten eingesetzt. In Deutschland etabliert ist das Konzept der Achszähler (vgl. Abbildung 2.5). Diese Komponenten funktionieren nach dem Prinzip der elektromagnetischen Induktion. Dabei wird an speziellen Gleispunkten ein Magnetfeld erzeugt, dessen Feldstärke sich ändert, wenn ein Zugrad darüber rollt. Diese Feldstärkenänderung wird von einer Elektronik registriert und entsprechend an das Stellwerk weitergeleitet. Durch den Einsatz von kurz hintereinander platzierten Detektionselementen kann die Fahrtrichtung des Zuges ermittelt werden. Achszähler werden eingesetzt, um Freimeldeabschnitte zu begrenzen und zu überwachen. Vor und nach jedem Freimeldeabschnitt (FMA - vgl. Kapitel 2.1.4.1) sind solche Achszähler installiert. Wenn die Anzahl der in den FMA eingefahrenen Achsen der Anzahl der aus dem FMA herausgefahrenen Achsen entspricht, wird der FMA im Stellwerk als *frei* gemeldet und kann wieder für eine weitere Zugfahrt reserviert werden.

2.1.3.7 Zugbeinflussungskomponente

Um Unfälle zu verhindern, wurden Stellwerke entwickelt, die Zugfahrten nur auf einem technisch gesicherten Gleisbereich zulassen. Triebfahrzeugführen werden somit durch Licht- und/oder Formsignale darüber informiert, ob es sicher ist, in einen Gleisbereich einzufahren. Ein Großteil der Unfälle im Schienenverkehr ist auf menschliches Versagen zurückzuführen. Es wurde in Folge dessen nach Wegen gesucht, auch die-

Abbildung 2.6: Gleismagnet - Bildquelle: flickr.com / CC-BY: Dutch Densha

ses Sicherheitsrisiko zu minimieren und Abhängigkeiten zwischen Streckensignalisierung und Triebfahrzeugsverhalten einzuführen. Unfälle, die durch Unachtsamkeit des Triebfahrzeugführers ausgelöst werden, sind durch die vermehrte Installation gleisseitiger Zugbeeinflussungskomponenten minimiert worden. So kann, wenn ein Triebfahrzeugführer beispielsweise an einem *Halt*-anzeigenden Signal unrechtmäßig vorbeifährt, der Zug automatisch ab- bzw. ausgebremst werden, um einen Unfall zu vermeiden. Auch eine überhöhte Geschwindigkeit kann mit diesem Verfahren automatisch reduziert werden.Diese Abbremsung erfolgt durch Zugbeeinflussungskomponenten - in der Regel sogenannte Gleismagneten oder auch PZB (Punktförmige Zugbeeinflussung - vgl. Abbildung 2.6). Solche PZB-Systeme bestehen aus gleis- und zugseitigen Komponenten. Der gleisseitige Magnet beinhaltet einen passiven Schwingkreis, der auf eine bestimmte Frequenz (500, 1000 bzw 2000 Hz) eingestellt und mit den entsprechenden Signalen verbunden ist. Zeigt beispielsweise ein Hauptsignal *Fahrt* an, wird der Schwingkreis des Magneten deaktiviert, da keine Zugbeeinflussung erfolgen soll. Zeigt das Signal jedoch *Halt* an, bleibt der Schwingkreis aktiviert. Wenn nun ein Zug über den aktiven Magneten fährt, wird dies im zugseitigen Magnet-Schwingkreis registriert. Auf diese Registrierung kann entsprechend reagiert werden, im Falle des Überfahrens einer *Halt*-Signalisierung mit einer Zwangsbremsung des Zuges. Das eben beschriebene Beispiel ist simpel, im Detail sind PZB-Systeme wesentlich komplexer und vielschichtiger.

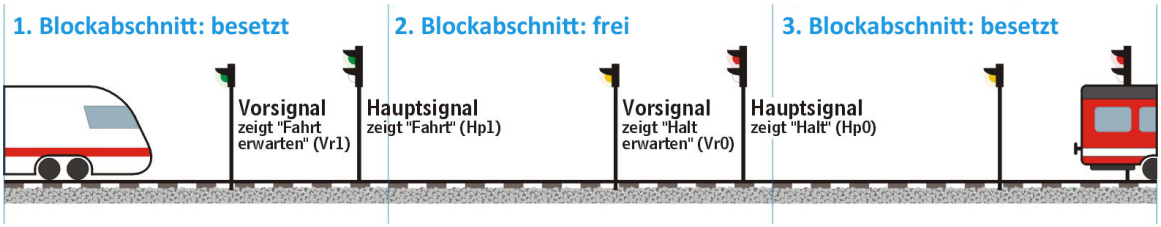

Abbildung 2.7: Schematische Darstellung von Blockabschnitten - Bildquelle: A. Darmochwal/Wikipedia Lizenz: CC-BY-SA 3.0

2.1.4 Konzepte der Leit- und Sicherungstechnik aus Stellwerkssicht

2.1.4.1 Blockabschnitt / Freimeldeabschnitt

Blockabschnitte / Freimeldeabschnitte (FMA) entsprechen aus Stellwerkssicht einem abgegrenzten Streckenabschnitt, in dem sich jeweils immer nur ein Zug befinden darf. Ein solcher Streckenabschnitt kann aus normalen Gleisen bestehen, aber auch Weichen, Bahnübergänge und andere infrastrukturellen Elemente beinhalten.

Die Abbildung 2.7 zeigt die Signalfolge zusammenhängender Blockabschnitte: Der Zug im von ihm besetzten Blockabschnitt 1 bekommt von den Signalen des freien Blockabschnitt 2 *Fahrt erwarten* bzw. *Fahrt* angezeigt - der Blockabschnitt 2 wurde somit im Stellwerk für diesen Zug reserviert. Im Blockabschnitt 3 befindet sich gerade ein Zug, somit ist dieser Block besetzt, was auch an den vorgelagerten Signalen durch die Signalbilder *Halt erwarten* bzw. *Halt* angezeigt wird. Wie im Beispiel beschrieben, werden FMA und Blockabschnitte in der Regel durch Signale abgegrenzt und gesichert. Des Weiteren existiert noch die Unterscheidung zwischen Gleisabschnitten der freien Strecke im Gegensatz zu denen eines Bahnhofs.

2.1.4.2 Fahrstraße

Fahrstraßen sind ein zentraler Begriff in der Stellwerkstechnik. Zugbewegungen finden grundsätzlich nur innerhalb einer definierten Fahrstraße statt, die zuvor im zuständigen Stellwerk für den entsprechenden Zug exklusiv für die Zeit der Fahrstraßendurchquerung reserviert wird. Die exklusive Nutzung wird dadurch gewährleistet, dass nach der Einfahrt des Zuges in die Fahrtstraße das Hauptsignal am Anfang der Fahrstraße auf *Halt* geschaltet wird und somit kein nachfolgender Zug in die Fahrstraße einfahren darf / kann. Wenn die Fahrstraße über mehrere Zufahrten erreichbar ist, werden auch

hier die Signale auf *Halt* geschaltet und die entsprechenden Weichen, wenn vorhanden, in eine der Fahrstraße abweisende Position umgestellt, damit auch hierher kein fremder Zug in die bereits reservierte Fahrstraße einfahren kann. Diese Weichen und Signale, die eine Fremdfahrt ermöglichen würden, werden als Flankenschutzelemente bezeichnet.

Das Einstellen einer Fahrstraße ist durchaus ein komplexer Vorgang, der aber durch den Einsatz moderner Stellwerkstechnologie weitgehend vereinfacht wird. Dazu prüft das Stellwerkssystem, ob alle beteiligten FMA frei sind und reserviert diese. Ist dies erfolgreich, werden daraufhin vom Stellwerk die Stellbefehle an die beteiligten Weichen gesendet, damit diese in die gewünschten Stellungen umlaufen. Ebenso werden die flankenschutzbietenden Weichen in eine sichere Stellung gebracht, die eine Fremdfahrt in die Fahrstraße verhindert. Zuletzt wird das Einfahrsignal der Fahrstraße auf *Fahrt* gestellt, um dem Zug die Einfahrt in die Fahrstraße zu ermöglichen. Diese komplexe Tätigkeitsreihenfolge wurde früher in mechanischen Stellwerken durch eine Abfolge des Umlegens bestimmter Hebel bewerkstelligt. Dazu war eine nicht unerhebliche Muskelkraft erforderlich. Die heutige Computertechnik in modernen ESTWs ermöglicht eine starke Abstraktion von den eigentlichen Vorgängen zur Einstellung einer Fahrstraße.

Eine Fahrstraße ist zusammengesetzt aus dem Fahr- und dem Durchrutschweg. Der Fahrweg ist auf die Strecke begrenzt, die der Zug im Rahmen der regulären Nutzung der Fahrstrecke in Anspruch nimmt. Der Durchrutschweg bezeichnet eine weitere Sicherheitseinrichtung. Er ist ein Streckenteil unmittelbar hinter dem Zielsignal einer Fahrstraße und ist ein Sicherheitsbereich, der ebenfalls reserviert und frei gehalten wird, falls ein Zug unrechtmäßigerweise nicht rechtzeitig vor dem Zielsignal zum Halten kommt.

2.1.5 Planung von Eisenbahn-Infrastrukturen

Die Planung neuer Eisenbahnstrecken unterliegt strengen Auflagen. Da es sich um sicherheitskritische Technik handelt, bei deren Fehlplanung oder unsachgemäßem Betrieb potentiell nicht unerheblicher materieller und personeller Schaden entstehen kann, sind sämtliche Details der Planung und des Betriebs in Richtlinien verankert. So existieren Richtlinien ebenso für die zu verwendenden Materialien wie für die Anforderungen an die Leit- und Sicherheitstechnik, die die Zugbewegungen überwacht und regelt.

Die in dieser Arbeit beschriebene Verifikation von EI-Planungsdaten bezieht sich auf Planungsdaten aus der Ausführungsplanung. Diese ist in einer fortgeschrittenen Pha-

se des Gesamtplanungsprozesses angesiedelt (vgl. Kapitel 1.3.1 und Abbildung 1.1). An der Ausführungsplanung sind die Firmen, die sicherheitstechnische Einrichtungen liefern, beteiligt.

In Deutschland ist die Deutsche Bahn AG der Herausgeber der entsprechenden Planungsrichtlinien. Für die vorliegende Arbeit relevant ist insbesondere die für die Ausführungsplanung entscheidende Richtlinie 'Ril 819: LST-Anlagen planen' [3]. Sie ist in folgende Modulgruppen für unterschiedliche Teilbereiche der EI-Planung untergliedert, wobei insbesondere die hervorgehobenen in dieser Arbeit untersucht wurden:

- 819.01 - **Entwürfe und Pläne**
- 819.02 - **Signale für Zug- und Rangierfahrten**
- 819.03 - Neben- und sonstige Signale
- 819.04 - Weichen, Kreuzungen und Gleissperren
- 819.05 - **Stellwerksfunktionen**
- 819.06 - Stellwerkseinrichtungen
- 819.07 - Betriebszentralen
- 819.08 - Beeinflussung und Schutzmaßnahmen
- 819.09 - Stromversorgung
- 819.10 - Blockanlagen
- 819.11 - Gleisfreimeldeanlagen
- 819.12 - Technische Sicherung der Bahnübergänge
- 819.13 - **Zugbeeinflussung und Führerraumsignalisierung**
- 819.14 - Rangiertechnik
- 819.15 - Strecken mit schwachem und mäßigem Verkehr (SMV-Strecken)
- 819.16 - Betriebliche Gefahrmeldeanlagen

- 819.17 - Bau- und RZ-Maßnahmen
- 819.18 - Betriebliche Telekommunikationsanlagen
- 819.19 - Funkfahrbetrieb
- 819.20 - Grundsätze S-Bahn Berlin/Hamburg
- 819.21 - Regeln für die Planung von Kabeltrassen

Die Ril 819 beinhaltet sehr viel Modulgruppen, die sich auf einer sehr niedrigen technischen Ebene bewegen. Da die vorliegende Arbeit sich eher mit einer höheren Ebene der EI-Planung befasst, ist die Auseinandersetzung mit den Inhalten dieser Modulgruppen nicht Gegenstand dieser Arbeit. Ferner ist es nicht Ziel der vorliegenden Arbeit, die Ril 819 und deren relevanten Modulgruppen in ihrer Gänze abzubilden. Lediglich eine Vorgehensweise der Formalisierung der EI-Domäne mit der Ril 819 als Leitfaden, insbesondere konkret formulierter Planungsanweisungen, sind als Beispiele in der vorliegenden Arbeit dargelegt. Diese werden in entsprechenden semantischen Regeln modelliert (vgl. Kapitel 4.4).

Es folgt ein Zitat aus der Ril 819.0202 - *Signale für Zug und Rangierfahrten*, welches die Bemessung von Durchrutschwegen beschreibt:

> *"Für Einfahrzeugstraßen ist ein Durchrutschweg hinter dem Zielsignal vorzusehen. Der Durchrutschweg beginnt am haltzeigenden Zielsignal und endet am maßgebenden Gefahrpunkt. [...] Der Durchrutschweg hinter Ausfahr- und Zwischensignalen ist in der Regel (in der Horizontalen) mindestens auf 200 m Regeldurchrutschweg zu bemessen.*
>
> *Eine Verkürzung des Regeldurchrutschweges gem. nachfolgender Tabelle ist zulässig [...]" (Ril 819.0202 - Signale für Zug- und Rangierfahrten, Abschn. 11, Abs. 1, 3 + 4)*

[...]	***zulässige Durchrutschweglänge***	***Einfahrgeschwindigkeit***
[...]	≥ 100	$100km/h \geq v \geq 40km/h$
	≥ 50	$v \geq 40km/h$

Die Verifikation der Ausführungsplanung von EI ist gegenwärtig ein Prozess, bei dem manuell sämtliche Planungsdaten gegen sämtliche Planungsvorschriften der Ril 819 geprüft werden müssen. Dies ist mit einem nicht unerheblichen zeitlichen und personellen Aufwand verbunden. Eine Planung gilt als erfolgreich und umsetzbar, wenn alle Planungsaspekte den entsprechenden Planungsvorschriften genügen und dies vom

EBA zertifiziert wurde. Die vorliegende Arbeit untersucht die Möglichkeiten einer automatisierten Verifikation dieser Planungsdaten mithilfe semantischer Technologien. Das oben angegebene Zitat zur Berechnung der Länge von Durchrutschwegen verdeutlicht, dass bei einer automatisierten Verifikation komplexe Zusammenhänge erfasst und formalisiert werden müssen. Somit muss eine Verifikationssoftware neben einem konzeptuellen Modell von EI ebenso über Fähigkeiten verfügen, konkrete Instanzen dieses Modells durch Schlussfolgerungen auf der Basis formaler Logik auszuwerten und mathematische Berechnungen durchzuführen, um eine umfassende Verifikation dieses Instanz-Modells zu realisieren.

Die Ausführungsplanung von EI erfolgt iterativ. Das bedeutet, dass meist ein erster Planungsentwurf anhand der Vorgaben der Entwurfsplanung angefertigt wird. Dieser enthält in der Regel nur wenige Elemente und stellt einen Durchstich dar. Es werden also beispielsweise alle Elemente für genau eine Fahrstraße geplant. Dann erfolgt eine Untersuchung auf Fehler und Inkonsistenzen sowie eine sukzessive Erweiterung des Elementumfangs und des Detailgrades. In der Regel wird bei diesen Untersuchungen ein Austausch von Planungsdaten mit beteiligten Unternehmen und Genehmigungsbehörden stattfinden. Ein Ergebnis dieses Austauschs ist meist ein Fehler- und Änderungskatalog, dessen Ausführungen in die Planungsdaten eingearbeitet werden müssen. Dieser Prozess ist zeitaufwändig und damit kostspielig. Die vorliegende Arbeit versucht, die Anzahl dieser Iterationsschritte durch eine Vorabverifikation der Planungsdaten zu minimieren.

In der Ausführungsplanung wird mithilfe eines graphischen Editors [54] das Modell der zu planenden EI erstellt. Im Editor können EI-Elemente arrangiert, attributiert und modifiziert werden. Auch abstrakte Elemente wie z.B. Fahrstraßen können mit dem Editor konzeptioniert und in Beziehung zu den EI-Elementen gesetzt werden. Des Weiteren wird die Möglichkeit geboten, Bedienoberflächen für die Bedienzentrale zu entwerfen und zu exportieren. Eine Simulationsfunktion ermöglicht das Testen der Dynamik, also die Analyse des potentiell laufenden Betriebs der konkreten EI inklusive der Schaltzeiten der einzelnen Elemente. Der Editor verfügt über eine rudimentäre Verifikationsfunktion zur Analyse der Sinnhaftigkeit der EI-Element-Attribute. Es kann beispielsweise überprüft werden, ob Fahrstraßenkombinationen möglich sind, oder ob Gleis- und Weichenelemente überhaupt verbunden sind und einen zusammenhängenden Graphen bilden. Weiterführende Möglichkeiten zur Überprüfung der modellierten EI werden nicht geboten.

2.2 XML Schema und Schematron

2.2.1 Einführung

Die *Extensible Markup Language* (kurz: XML) [42] ist weit verbreitet. Sie dient unter anderem als Notation zum implementierungs - und plattformunabhängigen Austausch strukturierter Daten. Das Anwendungsspektrum von XML ist sehr breit und bildet eine technologische Grundlage aus der eine Vielzahl von Sprachen entstanden ist. In Folge dessen wird XML auch als Metasprache bezeichnet.

In Listing 2.1 ist eine sehr einfache XML-Datei zur Beschreibung einer Zugstrecke mit zwei Gleisen und zwei Signalen dargestellt.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <zugstrecke
3    xmlns="http://www.trains.org/schemas/2013"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.trains.org/schemas/2013 ../schema/
         trains.xsd">
6    <gleise>
7      <gleis id="g1" laenge="10" />
8      <gleis id="g2" laenge="15" />
9    </gleise>
10   <signale>
11     <signal id="s1" typ="vorsignal" gleisref="g1" />
12     <signal id="s2" typ="hauptsignal" gleisref="g2" />
13   </signale>
14 </zugstrecke>
```

Listing 2.1: Simples XML Beispiel

Die Struktur eines XML-Dokuments ist syntaktisch betrachtet immer hierarchisch aufgebaut. Tags können andere Tags kapseln, jedoch nur vollständig und nicht teilweise. Dennoch können die hierarchischen Strukturen netzartige Eigenschaften aufweisen, bedingt durch Querverweise. So sind die im Dokument definierten Signale entsprechenden Gleisen zugeordnet. Die Modellierung dieser Aussage erfolgt durch eine Referenz: Die *gleisref*-Attribute in den Zeilen 11 und 12 referenzieren auf die Ids der jeweiligen Gleise aus den Zeilen 7 und 8, wodurch eine Graphen-Struktur innerhalb des Dokuments gebildet wird.

2.2.2 XML Schema

Zwar ist es ein großer Vorteil, Informationen formalisiert in XML speichern und austauschen zu können, jedoch ist der wesentliche Nutzen von XML ein anderer: XML-Dokumente können eine Referenz auf ein XML-Schema enthalten, welches die grammatikalische Struktur der referenzierenden XML-Dokumente vorgibt. Die Dokumente können dann automatisiert gegen dieses Schema validiert werden. Hierfür existieren Standardparser- und Validierungswerkzeuge wie beispielsweise Xerces von der Apache Foundation [16]. Ein solcher XML-Parser oder -Prozessor liest das XML-Dokument ein und stellt ein API zur Verfügung, mit der die Verarbeitung des Dokuments gesteuert werden kann. Ein XML-Dokument, welches von einem XML-Prozessor gegen sein entsprechendes Schema geprüft wird und diese Prüfung ohne Fehler besteht, gilt als valide. Eine Validitätsprüfung ist häufig die Voraussetzung für eine Weiterverarbeitung des Dokuments, da somit sichergestellt ist, dass das Dokument den Anforderungen für die sinnvolle Weiterverarbeitung entspricht. Dies ist insbesondere nach einem Austausch eines Dokuments zwischen mehreren Beteiligten / Anwendungen sinnvoll.

Die Prüfung eines Dokuments hat zwei Bestandteile: Prüfung auf Wohlgeformtheit und Prüfung auf Validität. Bei ersterer wird die korrekte Syntax des XML-Dokuments überprüft. Hierzu gehört die Prüfung, ob die Verschachtelungsvorgaben eingehalten worden sind, also ob keine Querverschachtelungen im Dokument vorhanden sind. Des Weiteren wird geprüft, ob die Tags der definierten Form entsprechen, also ob ein Tag mit einer spitzen, nach rechts geöffneten Klammer beginnt und mit einer spitzen, nach links geöffneten Klammer endet. Ebenso wird überprüft, ob die Attribute als Schlüssel-Wert-Paare formuliert sind. Ferner wird bei der Überprüfung gefordert, dass zu jedem Anfangs-Tag ein korrespondierendes Ende-Tag gefunden werden kann, oder die abgekürzte Form verwendet wird. Der zweite Bestandteil neben der syntaktischen Prüfung auf Wohlgeformtheit ist die Prüfung auf Validität. Hierzu wird die Grammatik des XML-Schemas, welches im zu prüfenden XML-Dokument referenziert wird hinzugezogen. In der Grammatik wird beispielsweise definiert, welche Tags in welcher Reihenfolge im Dokument auftreten dürfen, welche Attribute sie haben, welchem Typ oder Format die spezifischen Nutzdaten entsprechen müssen und Vieles mehr.

Eine Grammatik, die in XML Schema definiert wurde, ist wiederum ein XML-Dokument, das validiert werden kann. Neben XML Schema existieren andere Varianten zur Beschreibung von XML-Grammatiken. Hier sei die Dokumenttypendefinition (kurz: DTD) erwähnt, die eine eingeschränkte Mächtigkeit im Gegensatz zu XML Schema besitzt und in einer eigenen Syntax, also nicht in XML, formuliert wird. Aus Gründen der abnehmenden Verbreitung und des eingeschränkten Funktionsumfangs im Vergleich zu XML Schema, wird im Rahmen dieser Arbeit nicht näher auf DTD

eingegangen, sondern auf die Literatur verwiesen [8].

2.2.2.1 Einbindung eines XML-Schemas zur Validierung eines XML-Dokuments

In der Regel sind Schema und Dokument voneinander getrennt in separaten Dateien vorhanden. Somit muss ein Verweis auf das der Validierung zu Grunde liegenden Schemas im Dokument enthalten sein. Im vorigen Beispiel eines XML-Dokuments (vgl. 2.1) erfolgt die Einbindung des entsprechenden Schemas in Zeile 5. Sie besagt, dass eine Schemadatei des Namens „trains.xsd“ mit dem Namensraum „http://www.trains.org/schemas/2013“ existieren sollte, gegen welche dieses Dokument validiert werden kann. Zeile 3 des Dokuments besagt, dass der Basisnamensraum in diesem Dokument ebenfalls „http://www.trains.org/schemas/2013“ ist. Bei XML ermöglicht das Konzept der Namensräume die Verwendung und Unterscheidbarkeit von beispielsweise Tag-Namen, die eigentlich identisch sind. Ein voll-qualifizierter Tag-Name setzte sich aus einem Namensraum in Form einer URI und dem Tag-Namen zusammen. (Beispiel: „http://www.trains.org/schemas/2013/lecture“). Über abweichende Namensräume können diese unterschieden werden. Das Konzept dient lediglich der Unterscheidung. Es ist unerheblich, ob eine solche Ressource tatsächlich unter dem entsprechenden URI erreichbar ist.

In Listing 2.2 wird ein XML-Schema dargestellt, welches zur Validierung das XML-Dokuments aus dem Listing 2.1 genutzt werden kann.

```
1  <?xml version="1.0"?>
2  <xs:schema
3    xmlns:xs="http://www.w3.org/2001/XMLSchema"
4    xmlns:rail="http://www.trains.org/schemas/2013"
5    targetNamespace="http://www.trains.org/schemas/2013"
6    elementFormDefault="qualified">
7
8    <xs:element name="zugstrecke">
9      <xs:complexType>
10     <xs:sequence>
11       <xs:element name="gleise" type="rail:tracks"/>
12       <xs:element name="signale" type="rail:signals"/>
13     </xs:sequence>
14     </xs:complexType>
15   </xs:element>
16
17   <xs:complexType name="tracks">
18     <xs:sequence>
19       <xs:element name="track" type="rail:track" minOccurs="0"
20         maxOccurs="unbounded" />
```

```
21      </xs:sequence>
22    </xs:complexType>
23
24    <xs:complexType name="track">
25      <xs:attribute name="id" type="xs:ID" use="required"/>
26      <xs:attribute name="laenge" type="xs:integer" />
27    </xs:complexType>
28
29    <xs:complexType name="signals">
30      <xs:sequence>
31        <xs:element name="signal" type="rail:signal" minOccurs="0"
32          maxOccurs="unbounded" />
33      </xs:sequence>
34    </xs:complexType>
35
36    <xs:complexType name="signal">
37      <xs:attribute name="id" type="xs:ID" use="required"/>
38      <xs:attribute name="typ" type="xs:string" />
39      <xs:attribute name="gleisID"  type="xs:IDREF" use="required"/>
40    </xs:complexType>
41
42  </xs:schema>
```

Listing 2.2: XML-Schema für das Zugstrecken-Dokument

Die Vorteile im Gegensatz zu anderen Strukturbeschreibungssprachen wie beispielsweise DTD bestehen zum einen darin, dass XML-Schemata ebenfalls in der XML-Syntax geschrieben werden. Zum anderen verfügt XML Schema über eine höhere Ausdrucksstärke bedingt durch eines größeren Datentypenrepertoires und Konzepten wie beispielsweise Vererbung. Jedoch existieren auch bei XML Schema Grenzen im Hinblick auf die Ausdrucksstärke. XML Schema verfügt über keine arithmetischen Operationen, somit können keine Berechnungen mit Datenwerten durchgeführt werden. Des Weiteren können Datenabhängigkeiten nicht ausdrückt werden. So ist etwa die Formulierung eines Constraints in XML Schema nicht möglich, der unter der Voraussetzung, dass Element XY den Wert Z enthält, bei Element UV dann den Wert W erwartet.

Es existieren weitere Beschreibungssprachen, die diese Defizite zum Teil ausgleichen. Hervorzuheben ist an dieser Stelle die Beschreibungssprache Schematron [55]. Schematron ermöglicht die Formulierung von Constraints zur Beschreibung von Datenabhängigkeiten und gestattet die Verwendung von arithmetische Ausdrücken. Im folgenden Kapitel wird Schematron detailliert beschrieben.

2.2.3 Schematron

Schematron wurde 1999 in Taiwan von Rick Jelliffe entwickelt [25]. Seit 2006 ist die Sprache als ISO-Standard registriert. Als regelbasierte Sprache ermöglicht Schematron die Validierung von XML-Dokumenten hinsichtlich Struktur- und Inhaltsvorgaben. Es ist nicht als Ersatz von XML Schema entwickelt worden, jedoch können mit Schematron auch kontextbezogene Validierungen vorgenommen werden. Diese sogenannten *Co-occurence constraints* können dazu verwendet werden, Datenabhängigkeiten in XML-Dokumenten zu validieren. Datenabhängigkeiten ergeben sich daraus, dass Inhalte von Elementen/Attributen eines XML-Dokuments bezüglich der Inhalte anderer Elemente/Attribute in Relationen stehen. Diese Relationen können jedoch nur sehr eingeschränkt mit XML Schema abgebildet werden. Hier zeigt sich die Ausdrucksstärke von Schematron.

Ein Schematrondokument besteht aus Regeln, den sogenannten `patterns`. Diese sind gegliedert in `asserts` und `reports` und können komplexe Bedingungen beinhalten, die über die Ausdrucksstärke anderer Schemabeschreibungssprachen hinausgehen. Als Anwendungsgebiete eignen sich beispielsweise Szenarien, in denen Überprüfungen von Datenabhängigkeiten an verschiedenen Stellen eines XML-Dokuments durchgeführt werden sollen. Auch komplexe, arithmetische Operationen mit Datenwerten sind mithilfe von Schematron möglich.

Schematron Regeln werden in der XML-Syntax formuliert. Technisch basiert Schematron auf den etablierten XML-Technologien XML Schema, XSLT [13] und XPath [18]. Somit war zur Entwicklung von Schematron keine Neuimplementierung erforderlich.

Um die Funktionsweise von Schematron zu erklären, wird auf das Listing 2.3 verwiesen, welches sich auf das XML-Dokument aus 2.1 bezieht. In dem Schematrondokument werden, als konstruiertes Beispiel, zwei `asserts` definiert, die Bedingungen an valide Gleis-Elemente stellen.

```
1 <?xml version="1.0" ?>
2 <schema>
3   <title>Verification of railway infrastructures with Schematron</title>
4   <ns prefix="fw" uri="http://www.trains.org/schemas/2013"/>
5   <pattern name="Gleis Guidlines 0.1, Guidline 1" id="g1">
6     <rule context="fw:gleis">
7       <assert test="@laenge &lt; 10">Tracks have to be shorter than 10</
            assert>
8       <assert test="//signal[signal/@gleisref=current()/@id and signal/
            @type='vorsignal' and current()/@laenge = 10]">Every signal
            that is assigned to a track of length 10 has to be a '
            vorsignal'</assert>
9   </pattern>
```

```
10 </schema>
```

Listing 2.3: Schematron-Dokument mit Validierungsregeln

Da Schematron auf XSLT und XPath basiert, ist es möglich auf den vollen Umfang an XPath-Funktionen und -Operatoren zurückzugreifen. Dies eröffnet Möglichkeiten für numerische Vergleiche, die Verarbeitung von Zeichenketten sowie mathematische Operationen. In diesem Hinblick scheint Schematron ein geeigneter Kandidat für anspruchsvolle Verifikationen von EI zu sein. Eine wesentliche Bedingung für ein Verifikationssystem wird von Schematron jedoch nicht erfüllt: Da ein Gleisnetz als zusammenhängender Graph interpretiert wird, müssen transitive Relationen vom Verifikationssystem abgeprüft werden können. Als Beispiel sei hierfür der Zusammenhang zwischen einem Vor- und einem Hauptsignal eines Schienennetzes angeführt. Auf ein Vorsignal muss immer ein Hauptsignal folgen. Dieses muss sich jedoch nicht zwangsläufig auf dem selben Gleisabschnitt wie das Vorsignal befinden. Je nach Szenario kann das Hauptsignal auch auf einem der folgenden Gleisabschnitte positioniert sein. Bei der Überprüfung dieses Szenarios ist die Verwendung von transitiven Verbindungen von Gleisabschnitten erforderlich. Dies ist mit Schematron jedoch nicht validierbar.

2.3 OWL und SWRL

2.3.1 Einführung

Schon in der Antike versuchten die Menschen, ein in sich schlüssiges Begriffsgebäude zur Beschreibung realweltlicher Beobachtungen zu entwickeln. Diese Begriffssammlungen, in denen die verschiedenen Begriffe mit Attributen von einander unterschieden und gegenseitig in Beziehungen gesetzt wurden, hat die Bezeichnung *Ontologie* bis heute geprägt. Die *Ontologie*, aus dem Griechischen übersetzt - Die Lehre des Seienden - wurde zur formalen Beschreibung der Welt verwendet und ist ursprünglich in der allgemeinen Metaphysik angesiedelt, einer Disziplin der theoretischen Philosophie.

In der Informatik wird der Begriff *Ontologie* im Bereich der Wissensrepräsentation eingesetzt, um die Form einer Wissensbasis zu beschreiben. Ontologien in der Informatik stellen, ebenso wie in der Philosophie, Sammlungen von Begriffen dar, die über Attribute und Relationen beschrieben werden. In der Informatik werden Ontologien auf der Basis der Beschreibungslogik formal definiert. Dadurch können aussagekräftige Schlussfolgerungen aus den Beschreibungen der Ontologien gezogen werden, die wiederum in neuen Beschreibungen resultieren. Konkreter betrachtet können solche

Beschreibungen in einer Ontologie in zwei Typen unterteilt werden: Deklarationen und Fakten. Deklarationen sind Beschreibungen von Klassen, deren Attribute und die Beziehungen von Klassen untereinander sowie Einschränkungen bzw. Bedingungen für Klassenzugehörigkeiten. Diese Art der Beschreibung wird häufig auch als die TBox einer Wissensbasis bezeichnet. TBox - *terminological box* - bezeichnet somit die Terminologie der Wissensbasis. Zuzüglich zur TBox kann eine Ontologie auch die Beschreibung von Fakten enthalten. Dies sind die konkreten Instanzen der Klassen. Im Terminus von Ontologien wird hierbei in der Regel von Klassen-Individuen gesprochen. Diese Individuen beziehen sich auf eine oder mehrere Klassen (Polymorphismus [1]), haben eindeutige Wertezuweisungen zu ihren Attributen und können, je nach Deklaration zu anderen Individuen in Beziehung stehen. Diese Beschreibungsart wird als die ABox einer Wissensbasis bezeichnet. ABox - *assertional box* - bezieht sich somit auf das zugewiesene Wissen. ABox und TBox sind somit die beiden grundlegenden möglichen Bestandteile einer ontologischen Wissensbasis.

Ontologien in der Informatik sind ein wesentlicher Bestandteil der Semantic Web Initiative von Sir Tim Berners-Lee [9]. Diese hat das Ziel, den immensen Datenbestand des Internets besser zu strukturieren und es damit Computersystemen möglich zu machen, diese Daten vielseitig zu verarbeiten, in Beziehung zu setzen und somit den Benutzern fortschrittlichere Anwendungen zur Verfügung stellen zu können. Im Kontext des Semantic Web hat sich OWL als Ontologiebeschreibungssprache durchgesetzt. Die Sprache basiert auf der Beschreibungslogik und ermöglicht somit das logische Schließen von impliziten Fakten aus den expliziten. So ist es möglich, verdeckte Zusammenhänge in einem Modell zu entdecken und dementsprechend verschiedene Sichten auf die Daten des Modells aufzuzeigen. Dies ist eine wesentliche Anforderung an die Software des Semantic Webs. Der Prozess des logischen Schließens wird im Englischen als *Reasoning* bezeichnet. Die Programme, die dieses Reasoning ermöglichen werden dementsprechend als *Reasoner* bezeichnet. Es existieren eine Vielzahl unterschiedlicher Reasoner, sowohl im kommerziellen als auch im open source Bereich. Sie arbeiten mit verschiedenen Algorithmen, um die Schlussfolgerungen zu berechnen. Weit verbreitet ist hierbei die Familie der Tableau-Algorithmen [7], die Verfahren für eine performante, verhältnismäßig simple und vor allem entscheidbare Möglichkeit des Reasonings realisieren.

2.3.2 Open World vs. Closed World

Für Softwareentwickler existieren grundsätzlich zwei gegensätzliche Annahmen. Die eine Annahme, welche heutzutage bestimmend ist für Datenbanken und gängige Pro-

[1] Polymorphismus = Vielgestaltigkeit.

grammiersprachen, ist die einer geschlossenen Welt - die *Closed World Assumption* (CWA). Diese Annahme impliziert, dass nicht existente Angaben als falsch vorausgesetzt werden. Ist beispielsweise in einer Datenbanktabelle, die einen Terminkalender repräsentiert, für eine Uhrzeit kein Datensatz vorhanden, so wird im Allgemeinen davon ausgegangen, dass zu dieser Uhrzeit kein Termin existiert. Dies scheint auf den ersten Blick natürlich, da die Gewohnheit einer geschlossenen Welt vorherrscht. Die Aussage, dass unbekannte, nicht vorhandene Informationen grundsätzlich als falsch angenommen werden, wird im Englischen als *Negation as Failure* (NaF) bezeichnet. Im Gegensatz zur CWA existiert die Annahme einer offenen Welt - der *Open World Assumption* (OWA). Hierbei wird grundsätzlich nur das als wahr oder falsch angenommen, was explizit als solches definiert oder implizit als solches durch logische Rückschlüsse ermittelt wurde. Bei der OWA wird von einer unvollständigen Informationsbasis ausgegangen. Die OWA findet im Bereich der wissensbasierten Systeme Anwendung und unterliegt der Beschreibungslogik - ist also formal definiert und beinhaltet die Möglichkeit durch logisches Schließen einen Zugewinn expliziter Informationen aus impliziten Aussageverknüpfungen zu erlangen.

Die Beschreibungslogik ist mit dem Schlussfolgerungsmechanismus - dem sogenannten *reasoning* - ein mächtiges Werkzeug, mit dessen Hilfe sich wichtige Erkenntnisse aus eine Wissensbasis gewinnen lassen, die sich in Systemen, welche der CWA unterliegen, nicht erschließen würden. Ein vermeintlicher Nachteil der OWA ist jedoch das Fehlen von NaF. Dies hat zur Folge, dass Anwendungen, in denen eine Überprüfung der Vollständigkeit und Fehlerfreiheit der entsprechenden Daten gewährleistet sein muss, nur schwer umzusetzen sind.

Eine Übersicht über einen Vergleich von OWA und CWA lässt sich aus den Arbeiten von Horrocks und Patel-Schneider [47] sowie aus der Grundlagenliteratur *The Description Logic Handbook: Theory, Implementaion, and Applications* von Baader et al. [6] entnehmen. Die wichtigsten Aspekte sind in Tabelle 2.1 dargestellt.

2.3.3 OWL

Die Web Ontology Language OWL stellt eine Ontologiebeschreibungssprache dar, die dazu genutzt werden kann, unter der Verwendung verschiedener Syntaxen sowohl die A- als auch die TBox eindeutig formal zu beschreiben. OWL ermöglicht die Verbindung mehrerer Ontologien mithilfe eines Import-Mechanismus. So können z.B. A- und TBox auf unterschiedliche Ontologien aufgeteilt werden. Dies erlaubt unter anderem eine bessere Wart- und Lesbarkeit. Auch die Wiederverwendbarkeit grundlegender Ontologien wird durch diesen Mechanismus begünstigt. Es besteht ein breites Spektrum von Basis-Ontologien, die sich entweder auf eine konkrete Domäne beziehen

Open World Assumption - OWA	Closed World Assumption - CWA
Kein Negation as Failure In der OWA implizieren fehlende Fakten nicht automatisch, dass die entsprechende Ausssage als *falsch* angenommen wird. Ebenso wenig wird sie als *wahr* angenommen. In einer Wissensbasis, die der OWA unterliegt wird in dieser Situation lediglich von unvollständigem Wissen gesprochen, über das keine Aussage getroffen werden kann.	**Negation as Failure** In Systemen, die der CWA unterliegen, werden sämtliche Fakten, die nicht explizit als wahr deklariert sind als falsch angenommen. Hier gilt somit alles grundsätzlich solange als *falsch*, wenn der Wahrheitsgehalt nicht durch konkrete Aussagen als *wahr* deklariert worden ist.
Keine Unique Name Assumption Die OWA impliziert, dass Individuen grundsätzlich nicht als voneinander verschieden angenommen werden, es sei denn, es ist explizit definiert. Das hat zur Folge, dass Individuen unterschiedlichen Namens letztendlich auf das gleiche Individuum verweisen. Somit können Individuen nicht nur aufgrund ihres Namens eindeutig identifiziert werden.	**Unique Name Assumption (UNA)** In der CWA gilt das Prinzip der eindeutigen Identifizierbarkeit über Namen. Haben Individuen / Objekte unterschiedliche Namen, so ist davon auszugehen, dass es sich hierbei auch um unterschiedliche Individuen oder Objekte handelt.
Monotone Logik In der OWA können neu hinzugefügte Aussagen nicht bereits bestehende falsifizieren. Was als *wahr* angenommen ist, bleibt auch *wahr*. Wird eine konträre Aussage hinzugefügt, so wird die gesamte Wissensbasis in einen inkonsistenten Zustand versetzt. Schlussfolgerung sind in diesem Zustand nicht mehr formal möglich.	**Nicht-Monotone Logik** Die CWA geht davon aus, dass wenn keine Fakten vorhanden sind, alles als *falsch* angenommen werden muss. Das Hinzufügen von Fakten erweitert also die Wissensbasis. In der OWA hingegen wird die 'offenen Welt' über Restriktionen eingeschränkt. Wenn also in der CWA Fakten Erweiterungen darstellen, ist davon auszugehen, dass das Hinzufügen von Fakten nicht das Wissen, welches schon besteht, einschränken kann.

Tabelle 2.1: Vergleich der Open- und der Closed World Assumption

und deren Vokabular definieren, oder grundlegende Strukturen vorgeben, mit denen Domänen-Vokabulare definiert werden sollen. Eine verbreitete Domänen-Ontologie ist beispielsweise die FriendOfAFriend-Ontology *FOAF* [11], in der Beziehungen zwischen Personen modelliert werden.

OWL wurde vom W3C mit dem Ziel entwickelt, den Fortschritt des Internets hin zu einem 'intelligenteren', maschinen-interpretierbaren *Semantic Web* zu beschleunigen [57]. OWL-Ontologien sind in der Regel über URIs eindeutig referenzierbar und sind somit explizit dazu geeignet, im World Wide Web publiziert und geteilt zu werden.

OWL wurde als Erweiterung von RDF [19] und DAML/OIL [22] entwickelt. Neben der RDF/XML-Syntax können OWL-Ontologien in weiteren Syntaxen gespeichert werden. Die Wesentlichen sind *Manchester Syntax* [21], OWL/XML [46] und *Functional Syntax* [45]. In den Codebeispielen dieser Arbeit wird ausschließlich die *Functional Syntax* verwendet.

Zu den wesentlichen Sprachbestandteilen von OWL gehören die Beschreibungen von Klassen, Eigenschaften und Individuen. Hier sind auf den ersten Blick Parallelen zur objekt-orientierten Programmierung erkennbar, jedoch unterliegen die OWL-Modellierung und die objekt-orientierte Programmierung unterschiedlichen Paradigmen (OWL = OWA, OOP = CWA; Vgl. Kapitel 2.3.2).

Es wird zwischen verschiedenen Sprachumfängen von OWL unterschieden:

- **OWL Lite** wird häufig dazu verwendet, simple Hierarchien oder Beschränkungen abzubilden. Bewzogen auf Kardinalitäten, also die Vorgabe, wie viele Zuweisungen einer Eigenschaft das Individuum einer Klasse beschreibt, können bei OWL Lite nur Kardinalitätsbeschränkungen definiert werden, die entweder eine oder keine Zuweisung erlauben. OWL Lite findet in der Praxis kaum Verwendung.

- **OWL DL** ist in der Praxis am weitesten verbreiten. Das *DL* steht für *Description Logics*. Beschreibungslogiken beziehen sich in der Regel auf ein berechenbares Teilgebiet der Logik erster Ordnung, jedoch werden Beschreibungslogiken im Gegensatz zur Logik erster Ordnung durch ihre Entscheidbarkeit definiert.

- **OWL Full** beschreibt den größten Sprachumfang der OWL-Familie und ermöglicht beispielsweise die Verwendung von Klassen als Individuen. Des Weiteren können z.B. Eigenschaften von Eigenschaften definiert werden. Diese maximale Ausdrucksstärke hat zur Folge, dass nicht alle Bestandteile von OWL Full entscheidbar sind. Ein Reaosoning-Prozess kann somit nicht garantieren, ob und in welcher Zeit ein Ergebnis berechnet werden kann.

In dieser Arbeit ist das Reasoning ein zentraler Bestandteil der Aufgabenstellung, weswegen der Sprachumfang OWL DL verwendet wird. Wenn in den weiteren Erläuterungen dieser Arbeit von OWL geschrieben wird, ist davon auszugehen, dass sie sich somit auf OWL DL beziehen. Des Weiteren existieren zwei Hauptversionen des OWL Standards. In dieser Arbeit wird die neuere Version OWL 2 verwendet, wenn nicht anders beschrieben.

Klassen werden in der *functional syntax* von OWL mit den Schlüsselworten `Declaration(Class(<ClassName>))` deklariert. Neben diesen benannten Klassen, die über eine eindeutige Bezeichnung verfügen, ist es möglich, mit OWL anonyme Klassen zu definieren. Diese sind in der Regel durch Klassen-Ausdrücke (class expressions) deklariert. So kann beispielsweise eine anonyme Klasse aus der Vereinigung der Klassen `Mann` und `Frau` deklariert werden. Ihr werden somit alle Individuen der Klassen `Mann` oder `Frau` zugewiesen. Anonyme Klassen sind somit eher als eine Sammlung von Individuen zu verstehen, die bestimmten Voraussetzungen und Einschränkungen genügen. Klassenbeschreibungen können über unterschiedliche Aussagen - *Axiome* - definiert werden. So können Klassen-Beschreibungen nicht nur, wie im vorigen Beispiel erläutert, über Klassenvereinigungen beschreiben werden, sondern beispielsweise auch über Einschränkungen bezüglich Eigenschaftszuweisungen. Im Eisenbahnkontext ist so beispielsweise die anonyme Klasse eines Bahnhofsgleis denkbar, die aus der Oberklasse `Gleis` abgeleitet wird und über die Einschränkung verfügt, dass sie eine Eigenschaftszuweisung zu einer Typenbeschreibung `stationTrack` aufweisen muss. Die Beschreibung anonymer Klassen über Einschränkungen bezüglich Eigenschaftszuweisungen wird im Englischen als *property restriction* bezeichnet. Es existieren zwei Arten dieser Einschränkungen:

1. **Value constraint** - Die Einschränkung bezüglich des Wertebereichs einer Eigenschaft. So kann beispielsweise eine Eigenschaft `hasSibling`, die als Wertebereich `Man` und `Woman` hat, dahingehend eingeschränkt werden, dass sie nur Werte von `Woman` annehmen darf, wenn sie im Kontext einer anonymen Klasse zur Beschreibung einer Schwester-Beziehung verwendet wird.

2. **Cardinality constraint** - Einschränkungen bezüglich Kardinalitäten von Eigenschaften ermöglichen die Definition von anonymen Klassen, dessen Individuen über eine bestimmte Anzahl der betreffenden Eigenschaften verfügen. Im OWL-Standard werden hier als Beispiel anonyme Klassen für Sportmannschaften genannt. So müssen die Individuen eines Fußballteams genau elf `hasPlayer` Eigenschaftszuweisungen aufweisen und die Individuen eines Basketballteams genau fünf.

Klassenbeziehung werden über entsprechende Axiome deklariert. Mit ihnen können

beispielsweise Beziehung bezüglich Unterklassen (`owl:subClassOf`), Äquivalenzklassen (`owl:equivalentClass`) und disjunkte Klassen (`owl:disjointWith`) beschrieben werden.

Eigenschaften werden, im Gegensatz zur objekt-orientierten Programmierung, nicht direkt im Kontext von Klassen deklariert, sondern unabhängig von ihnen. Ferner werden Eigenschaften *domains* und *ranges* zugewiesen. *Domains* beschreiben immer die Klassen, von denen diese Eigenschaften verwendet werden können, also den Kontext im OOP-Sinn. *Ranges* hingegen beziehen sich auf die Datentypen, denen die Eigenschaftswerte unterliegen. Hierbei wird grundsätzlich zwischen zwei Arten von Eigenschaften unterschieden. Während *Datatype properties* sich immer auf einen Datenwert beziehen, sind *Object properties* in OWL DL immer auf ein Individuum bezogen. Eigenschaften können nicht nur verschiedenen Typs sein, sondern auch über weitere Modifikatoren beschrieben werden. Diese Modifikatoren werden in der folgenden Auflistung erläutert:

- **Functional** - Funktionale Eigenschaften können nur eine (eindeutige) Wertzuweisung pro Individuum haben. Es können also keine unterschiedlichen Werte $x1$ und $x2$ für ein konkretes Individuum existieren. Sowohl *datatype properties* als auch *object properties* können als funktional deklariert werden. Dies ist der einzige Modifikator, der auch für *datatype properties* verwendet werden kann. Alle folgenden Modifikatoren können nur bei *object properties* verwendet werden.

- **Inverse functional** - Eine invers funktionale Eigenschaft beschreibt, dass ein konkreter Wert nur einem spezifisches Individuum zugewiesen kann. Als Beispiel wird im OWL-Standard die invers funktionale Eigenschaft `biologicalMotherOf` aufgeführt. Hierbei kann ein `Mensch` (*range*) eindeutig über die Eigenschafts-*domain* `Mother` identifiziert werden.

- **Transitive** - Eine transitive Eigenschaft P ist so definiert, das für ein Paar (x,y) als Ausprägung von P und ein Paar (y,z) als Ausprägung von P darauf geschlossen werden kann, dass ebenso das Paar (x,z) als Ausprägung von P existiert. Im OWL-Standard wird hierbei die transitive Eigenschaft `subRegionOf` als Beispiel genannt. Diese Eigenschaft hat als *domain* und *range* die Klasse `Region` zugewiesen. Eine Sub-Region X kann somit eine Sub-Region Y haben, die wiederum eine Sub-Region Z hat. Z ist also auch Sub-Region von X.

- **Symmetric** - Bei symmetrischen Eigenschaften kann durch Schlussfolgerung ermittelt werden, dass *domain* und *range* vertauscht werden können. Beispiel hierzu wäre eine `friendOf` Eigenschaft, bei der A Freund von B ist und andersherum.

- **Asymmetric** - Im Gegensatz zur symmetrischen Eigenschaft kann bei der der Verbindung des Individuums A zum Individuum B über eine asymmetrische Eigenschaft darauf geschlossen werden, dass niemals die Verbindung B -> A existieren kann.

- **Reflexive** - Reflexive Eigenschaften beziehen sich immer von einem Individuum auf sich selbst. Im OWL-Standard wird hierzu das Beispiel `hasRelative` genannt, dass besagt, dass man immer auch mit sich selbst verwandt ist.

- **Irreflexive** - Bei irreflexiven Eigenschaften können sich im Gegensatz dazu die entsprechenden Individuen niemals auf sich selbst beziehen. Beispiel hierfür: Eine `istElternteilVon`-Eigenschaft.

Neben den Modifikatoren können Eigenschaften ebenso wie Klassen als disjunkt zueinander deklariert werden. Zwei Individuum können somit nicht über beide Eigenschaften gleichzeitig miteinander verbunden werden.

Ein weiteres Konstrukt bezüglich Eigenschaften ist erst in der Version 2 des OWL Standards hinzugefügt worden. Dieses ist das Konstrukt der Eigenschaftsverkettung (*property chain*). Hiermit können Individuen über eine Verkettung mehrerer Individuen miteinander verbunden werden. Als Beispiel wird im Standard die Eigenschaftskette `hasGrandparent` genannt, die durch eine Verkettung von genau zwei `hasParent` Eigenschaften beschrieben wird.

Individuen beschreiben das Faktenwissen, also die ABox, der Wissensbasis. Bei der Deklaration von Individuen wird zwischen zwei Typen von Fakt-Axiomen unterschieden:

1. Fakten über Klassenzugehörigkeiten und Eigenschaftswerten von Individuen

2. Fakten über die Identitäten von Individuen

In Fakt-Axiomen bezüglich Klassenzugehörigkeit können vorher deklarierte Individuen entsprechenden Klassen zugewiesen werden. Hierbei kann einem Individuum aufgrund des in OWL möglichen Polymorphismus mehrere Klassenzugehörigkeiten zugewiesen werden. Des Weiteren können einem Individuum Bezüge zu *properties* zugewiesen werden. Hierbei wird genaugenommen der entsprechenden Eigenschaft als *domain* das Individuum zugewiesen und als *range* ein entsprechender Eigenschaftswert. Wie vorher beschrieben, kann der Wert bei *datatype properties* ein Datenwert sein und bei *object properties* ein weiteres Individuum.

Da OWL nicht der Unique Name Assumption unterliegt, können über Gleichheit

und Verschiedenheit von unterschiedlich benannten Individuen ohne weitere Informationen keinen Aussagen getroffen werden. Um die genaue Identität eines Individuums festzulegen, können die drei OWL-Konstrukte `owl:sameAs, owl:differentFrom` sowie `owl:Alldifferent` in Axiomen verwendet werden. Während `owl:sameAs` und `owl:differentFrom` als Argumente jeweils zwei Individuen erwarten, kann bei `owl:Alldifferent` eine Liste von als voneinander verschieden anzusehenden Individuen angegeben werden.

OWL verfügt über die Möglichkeit der Nutzung verschiedener Datentypen. Neben XML Schema Datentypen kann auch der Typ `rdf:Literal` verwendet werden. Des Weiteren bietet OWL die Möglichkeit, einen Bereich von festgelegten Werten als Datentyp zu nutzen. Dieser Aufzählungstyp (*enumerated datatype* wird über das OWL-Konstrukt `owl:oneOf` definiert. Der Wertebereich kann sich sowohl über einfache Datentypen als auch über eine Sammlung von Individuen erstrecken.

2.3.4 SWRL

Die Semantic Web Rule Language SWRL ist einer Regelsprache, die aus einer Kombination von RuleML [10] und OWL besteht. Die Sprache wurde mit dem Ziel entwickelt, die OWL-Semantik um Implikations-basierte Regeln zu ergänzen. So wird die Ausdrucksstärke von OWL-basierten Wissensbasen um Formulierungen erweitert, die mit Standard OWL nicht darstellbar sind.

```
1 Signal(?s), hasSignalType(?s, Main) -> MainSignal(?s)
```

Listing 2.4: Beispiel einer simplen SWRL-Regel in abstrakter Syntax

SWRL wird vom W3C seit 2004 als *Member Submission* geführt [24]. Syntaktisch orientiert sich die Sprache an OWL und ist somit in mehreren Serialisierungsformaten speicherbar, darunter eine abstrakte Syntax sowie RDF/XML. Im Rahmen dieser Arbeit wird in den Codebeispielen ausschließlich die abstakte Syntax verwendet.

SWRL-Regeln bestehen aus einem Antezedenz- und einem Konsequenz-Teil der Form *Antezendez -> Konsequenz*. Die Interpretation einer SWRL-Regel ist die Folgende: Wenn die Bedingungen der Antezedenz erfüllt sind, dann gelten auch die Bedingungen der Konsequenz als erfüllt. Antezedenz und Konsequenz können jeweils aus 0-n Regelatomen bestehen, die durch Konjunktionen miteinander verbunden sind. Ein leerer Antezedenz-Teil gilt als trivial wahr, während ein leerer Konsequenz-Teil als trivial falsch interpretiert wird. Regelatome können den Formen $C(x)$, $P(x,y)$, $sameAs(x,y)$ oder $differentFrom(x,y)$ entsprechen, wobei C eine OWL-Klassenbeschreibung, P

eine OWL-Eigenschaftsbeschreibung und x,y Variablenbeschreibungen, OWL-Individuen oder OWL-Datenwerten entsprechen. Bei einer Auswertung der Regeln mithilfe eines OWL/SWRL-Reasoners wird hierbei versucht, Belegungen für die Variablen der Regeln aus Daten der OWL-Wissensbasis zu finden. Ist dies erfolgreich, wird für jede mögliche Variablenbelegung der Konsequenz-Teil der entsprechenden Regeln als geschlussfolgerter Fakt der Wissensbasis betrachtet und in dieser verwendet.

Zuzüglich zu den bereits erläuterten Atomen ermöglicht SWRL die Verwendung sogenannter Builtins. Diese sind Funktionen, welche differenziertere Manipulationen und Vergleiche der x,y-Werte ermöglichen. Folgende Arten von Builtins werden unterschieden:

- **Builtins für Vergleiche** - Mithilfe dieser Builtins können insbesondere nummerische Werte auf Gleich- oder Ungleichheit verglichen werden. Des Weiteren können größer/kleiner-Vergleiche vorgenommen werden.

- **Mathamatische Builtins** - Diese bieten eine Reihe mathematischer Funktionen wie beispielsweise die Grundrechenarten, Rundungen oder Winkelfunktionen.

- **Builtins für boolsche Werte** - Hiermit können Wahrheitswerte verglichen werden.

- **Builtins für Zeichenketten** - Eine Reihe von Zeichenkettenmanipulationen wie beispielsweise Verkettungen oder Zeichenextraktionen, aber auch Zeichenkettenvergleiche werden mithilfe dieser Builtins ermöglicht.

- **Builtins für Datum und Zeit** - Die Verarbeitung von Zeit(räumen) und Datumsangaben wird durch mehrere Builtins dieser Kategorie unterstützt.

- **Builtins für URIs** - Um URIs zu Verarbeiten, werden Builtins dieser Kategorie genutzt.

- **Builtins für Listen** - Diese Kategorie bietet Builtins für die Manipulation von RDF-style Listen, wobei diese Arten von Listen lediglich in OWL Full verwendet werden können.

SWRL ist unentscheidbar (vgl. [23]). Das bedeutet, dass kein Algorithmus existiert, der in endlicher Zeit und in jedem Fall ermitteln kann, ob ein Axiom, abgeleitet aus einer SWRL-Regel, aus der Wissensbasis geschlussfolgert werden kann. Somit ist Standard-SWRL nicht in OWL DL Systemen einsetzbar. In der Praxis wurde jedoch eine Einschränkung für eine entscheidbare (*DL-safe*) Variante von SWRL identifiziert

und implementiert: SWRL-Regeln müssen sich auf die Verwendung von benannten Individuen beschränken. Es können also keine Individuen verarbeitet werden, die nicht benannt sind, aber aufgrund der logischen Schlussfolgerungen als existent angenommen werden. Mit dieser Einschränkung ist eine Verwendung von (DL-safe) SWRL sowie OWL DL und ebenso von entsprechenden Reasonern möglich.

2.3.5 Protégé Ontologie-Editor

Der Ontologie-Editor Protégé [20] ist ein Standardwerkzeug zur Bearbeitung von OWL-Ontologien sowie SWRL-Regeln und steht unter der Mozilla public license als open source Software zur freien Verfügung. Er wurde im Stanford Center for Biomedical Informatics Research der Stanford University School of Medicine entwickelt. Ursprünglich wurde Protégé dazu konzipiert, die Erstellung ontologischer Wissensbasen für medizinische Forschungszwecke zu unterstützen. Der Editor bietet neben einer graphischen Benutzeroberfläche die Möglichkeit des Einbindens von Reasonern. Somit können Konsistenzchecks sowie Inferenzbildungen innerhalb des Editors durchgeführt werden. Protégé basiert auf Java und verfügt über ein umfangreiches Plugin-Interface. Neben Reasonern, die als Plugin eingebunden werden, existiert eine Vielzahl unterschiedlicher Plugins beispielsweise zur Visualisierung von Ontologien, zur Codegenerierung oder als Erweiterungen zur kollaborativen Ontologiebeabeitung.

3 Von der syntaktischen zur semantischen Modellierung

Die Grundidee eines mehrschichtigen Verifikationssystems für Eisenbahn-Infrastrukturen wird in den folgenden Kapiteln erörtert. Neben der eigentlichen Verifikation stellt ein Debuggingmechanismus die Möglichkeit zur Verfügung, Verifikationsfehler auf ihren Ursprung zurückzuführen. Das Konzept dieses Mechanismus wird ebenfalls in diesem Abschnitt erläutert.

3.1 Verifikation - Konzeptbeschreibung

Bei XML-basierten Modellen scheint es ein pragmatischer Ansatz zu sein, regelbasierte Validierungssprachen wie Schematron einzusetzen, um Datenabhängigkeiten zu beschreiben und validieren zu können. Es hat sich gezeigt, dass jedoch auch Schematron über Grenzen bezüglich der Ausdrucksstärke verfügt und beispielsweise die Validierung von transitiven Strukturen nicht unterstützt.

Insbesondere Transitivitätsbetrachtungen und daraus abgeleitete Befahrbarkeitsmöglichkeiten sind jedoch ein Kernbereich bei der Verifikation von Schienennetzen. Hieraus entstand die Idee, ein System einzusetzen, welches explizit die Modellierung und Überprüfung von Transitivitätsbeziehungen unterstützt und ebenso die Repräsentation von weiteren Elementen und Zusammenhängen der Eisenbahndomäne auf einer konkreten Ebene ermöglicht. OWL bietet die Möglichkeit, Beziehung von Klassen durch *data properties* abzubilden. Diese *properties* können als transitiv gekennzeichnet werden. So kann beispielsweise eine *transitive data property* '`connect`' definiert werden, mit der die Verbindung von Gleisabschnitten beschrieben wird. Die Transitivität besteht darin, dass ein Gleisabschnitt über `connect` nicht nur mit seinem direkten Nachbarn verbunden ist, sondern ebenso mit allen folgenden Gleisabschnitten, die miteinander über die '`connect`' *property* verbunden sind. Im Listing 3.1 wird diese Funktionalität verdeutlicht. Zuerst werden die Klasse `Track` als Subklasse von `TwoDimObject` und die *transitive data property* `connect` deklariert. Hiernach werden

drei `Track`-Individuen, `t1, t2` und `t3` deklariert. `t1` wird über `connect` mit `t2` verbunden. Analog hierzu `t2` und `t3`. `t1` und `t2` sind somit nicht explizit als verbunden deklariert. Ein Reasoner kann dieses jedoch implizit schlussfolgern, da `connect` als *transitive data property* konzipiert ist. Diese Funktionalität eröffnet die Möglichkeit die Verbindungen eines gesamten Gleisbildes zu überprüfen.

```
1  Declaration ( Class ( :Track ) )
2  SubClassOf( :Track :TwoDimObject )
3
4  Declaration ( ObjectProperty ( :connect ) )
5  TransitiveObjectProperty ( :connect )
6  ObjectPropertyDomain( :connect :TwoDimObject )
7  ObjectPropertyRange( :connect :TwoDimObject )
8
9  Declaration (NamedIndividual( :t1 ) )
10 ClassAssertion ( :Track :t1 )
11 Declaration (NamedIndividual( :t2 ) )
12 ClassAssertion ( :Track :t2 )
13 Declaration (NamedIndividual( :t3 ) )
14 ClassAssertion ( :Track :t3 )
15
16 ObjectPropertyAssertion ( :connect :t1 :t2 )
17 ObjectPropertyAssertion ( :connect :t2 :t3 )
```

Listing 3.1: Veranschaulichung einer *transitive data property* am Beispiel von Gleisabschnitten

OWL zeigt sich als geeigneter Kandidat, diese Modellierungsmöglichkeiten zu bieten. Die auf OWL und RuleML basierende Regelsprache SWRL präsentierte sich als zweckmäßiger Kandidat zur Repräsentation von Planungsrichtlinien und beinhaltet die notwendige Fähigkeit zur Formulierung und Auswertung arithmetischer Ausdrücke, die beispielsweise zur Berechnung von Distanzen benötigt werden.

Das grundlegende Konzept zur Verifikation von EI-Planungsdaten erschließt sich aus der Abbildung 3.1: Ein konkretes EI-Modell, welches Planungsdaten einer Eisenbahnstrecke beinhaltet, soll auf seine Konformität in Bezug auf definierte Planungsrichtlinien getestet, also im Sprachgebrauch dieser Arbeit verifiziert, werden. Das EI-Modell liegt als XML-Dokument vor und validiert gegen ein entsprechendes XML-Schema. Um über die rein syntaktische Schemenvalidierung hinauszugehen, ist es notwendig, das EI-XML-Modell in ein anderes Format zu überführen. Dies geschieht mithilfe eines entsprechenden Transformationswerkzeugs. Da das Zielformat, OWL, ebenfalls neben anderen Syntaxen eine XML Syntax besitzt, ist es möglich, einen XML-Transformator wie beispielsweise Saxon [26] zu verwenden. Eine solche Transformation wird durch ein entsprechendes Skript gesteuert. Hierzu wird XSLT, ebenfalls eine XML-Technologie, verwendet. Mithilfe der Transformation wird das syntaktische EI-

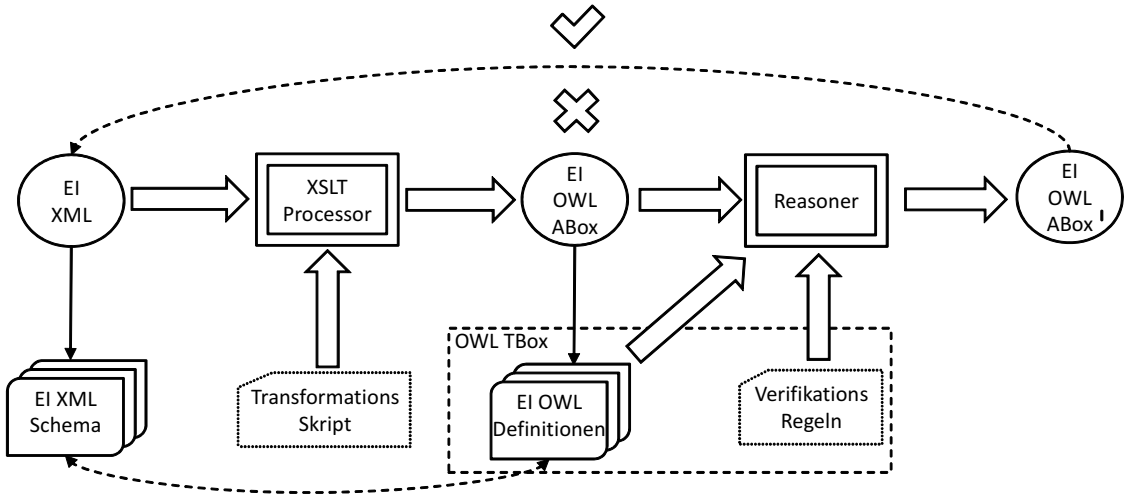

Abbildung 3.1: Grundlegendes Konzept des Verifikationssystems

XML-Modell in ein semantisches EI-OWL-Modell überführt, dessen Semantik auf der Basis einer Beschreibungslogik formalisiert ist. Das bedeutet letztendlich, dass jede Klasse über einen nennenswert größeren Eigenschaftsumfang verfügt und dass auch der Vernetzungsgrad der Klassen untereinander signifikant größer ist.

Das EI-OWL-Modell entspricht einer ABox (*assertional knowledge*), enthält also Klassen- und Eigenschaftszuweisungen für Individuen. Die Klassen- und Eigenschaftsdefinitionen hingegen werden als TBox (*terminological knowledge*) bezeichnet. Diese wird von der ABox importiert, damit die entsprechenden Definitionen verwendet werden können. Die TBox ist so konstruiert, dass sie konform zu dem XML-Schema des Ursprungsmodells ist, also sämtliche ursprünglichen Modellaspekte enthält und darüber hinausgeht. Zusätzlich erfolgten umfangreiche Diskussionen mit Experten der Eisenbahndomäne in denen sichergestellt wird, dass sämtliche Aspekte des ontologischen Modells konform sind mit den, diese Domäne betreffenden, realweltlichen Vorstellungen.

Als weiterer Bestandteil des ontologischen TBox-Modells wird neben den Klassen- und Eigenschaftsdefinitionen auch die SWRL-Regelbasis betrachtet. Die Definitionen aus der domänenspezifischen TBox werden dazu genutzt, mit SWRL die Planungsrichtlinien zu formulieren, gegen die die konkreten EI-Planungsdaten in Form der transformierten EI-OWL-ABox verifiziert werden sollen. Der eigentliche Verifikationsablauf wird durch den Einsatz des Reasoners angestoßen. Dieser wertet die Aussagen in der A- und TBox aus und führt ebenso eine Anwendung der SWRL-Regelbasis aus. Als Resultat wird ein erweitertes EI-OWL-ABox (EI OWL ABox') Modell zurückgeliefert. Dieses Modell enthält Aussagen über die Regelkonformität

der Individuen und liefert somit einen Bericht über Fehler oder Lücken in dem ursprünglichen EI-Planungsmodell.

3.2 Verifikation - Kontroverse

Auch wenn OWL / SWRL und deren Basis einer Beschreibungslogik auf den ersten Blick pragmatische Technologien für die Überprüfung von (semantischen) Modellen darstellen, ist jedoch der Ansatz der Verifikation insbesondere von EI-Planungsdaten mithilfe diesen Technologien ein Novum. Die Kontroversen, die diese Vorgehensweise begleiten, werden im Folgenden erörtert.

3.2.1 Unterschiedliche Formate

Die zu verifizierenden EI-Daten liegen ursprünglich in einem XML-Format vor, welches einem XML-Schema folgt. Um diese Daten auf semantischer Ebene verifizieren zu können, müssen sie dafür in das OWL-Format transformiert werden. Da auch OWL eine XML-Repräsentation besitzt, ist dies mit XSLT möglich (vgl. Kapitel 4.5). Hierbei muss sichergestellt werden, dass keine relevanten Informationen bei der Transformation verlorengehen oder verfälscht werden. Um dies zu gewährleisten, müssen lediglich die Transformationsskripte diesbezüglich untersucht werden. Es ist nicht notwendig, nach jeder erfolgten Transformation, jede einzelne EI-Planungsdatei im XML Format mit der entsprechenden EI-Planungsdatei im OWL-Format zu vergleichen, da mit XSLT die Transformationen nicht auf Instanz-Ebene (M0), sondern auf Modell-Ebene (M1) definiert werden (siehe Ebenen der Meta Object Facility in [41]). Die Definition einer Transformation auf Modell-Ebene hat den Vorteil, dass diese als eine Transformation für alle Instanzen eines (XML)-Schemas verwendet werden kann.

In enger Zusammenarbeit mit Domänenexperten der EI-Planung wurde systematisch die Vollständigkeit und Unverfälschtheit der relevanten transformierten EI-Planungsdaten untersucht (vgl. Kapitel 6).

3.2.2 Unterschiedliche Paradigmen

Wie im vorigen Kapitel beschrieben, wird für Verifikationszwecke mithilfe semantischer Technologien von einem in ein anderes XML-Format transformiert. Bei dem

Ausgangsformat handelt es sich lediglich um ein XML-Dokument, welches einem XML-Schema unterliegt. Bei dem transformierten Format handelt es sich jedoch um eine XML-Serialisierung einer OWL-Ontologie. Auch wenn beide Formate XML als Grundlage haben (können), folgen sie jedoch zwei grundsätzlich verschiedenen Modellierungsparadigmen. Die Planungsdaten im Ausgangsformat unterliegen der CWA, d.h. nicht explizit definierte Elemente werden als fehlende Elemente aufgefasst, währenddessen OWL der OWA unterliegt (vgl. Kapitel 2.3.2), d.h. zu fehlenden Planungsinformationen kann keine Aussage getroffen werden. Dadurch ergeben sich bei der Transformation eine Reihe von Problematiken, die es zu beachten gilt:

Da in der OWA im Gegensatz zur CWA nicht die NaF gilt, wird hier von einer unvollständigen Wissensbasis ausgegangen. Somit ist auch die Menge der Individuen nicht auf die explizit deklarierten beschränkt. Dies ist jedoch zwingend erforderlich, um eine Verifikation durchführen zu können. Hierzu erfolgt die Verwendung von sogenannten *value partitions*. Diese ermöglichen die Einschränkung der Definition einer Klasse. Somit werden nur die in den *value partitions* eingeschlossenen Individuen als Teil dieser Klasse akzeptiert und auf diese beschränkt.

Eine weitere Problematik, die sich aus der OWA ergibt, ist die der fehlenden UNA. Dies bedeutet, dass Individuen nicht aufgrund unterschiedlicher Bezeichner als voneinander verschieden angenommen werden können. Im Umkehrschluss können somit unterschiedlich benannte Individuen ein einziges 'Objekt' beschreiben. Dieses Verhalten ist für Verifikationszwecke in der Regel unerwünscht. Um nun eindeutige Individuen bei einer Transformation zu erhalten, müssen in der entsprechenden Individuen-Deklaration diese über ein *owl:AllDifferent*-Konstrukt als voneinander verschieden angegeben werden. Das Handling dieser Problematik und die Verwendung der *value partitions* wird in Kapitel 4.5 beschrieben.

Unabhängig von der Transformation, ist bei der Modellierung der Klassen-Ontologie zu beachten, dass Individuen über obligatorische *property*-Zuweisungen verfügen, um als zu einer Klasse zugehörig charakterisiert zu werden. Dies erfolgt über die Verwendung von sogenannten *closure axioms*. Eine Verwendung dieses Konstrukts wird im Kapitel 4.3 beschrieben.

3.2.3 Gerechtfertigter Aufwand

Es ist anzumerken, dass die Verwendung semantischer Technologien mit Aufwand verbunden ist: Die EI-Domäne muss mithilfe dieser Technologien schlüssig modelliert werden und eine Transformation in die entsprechenden Formate muss definiert und implementiert werden. Dieser Aufwand ist jedoch insofern gerechtfertigt, als dass er

einen erheblichen Mehrwert bei Verifikationstätigkeiten bietet. Mit reinem XML Schema kann das Modell nur strukturell validiert werden. Die Dateninhalte insbesondere deren Abhängigkeiten können nicht oder nur unzureichend verifiziert werden. Für das Angestrebte, eine automatisierte Verifikation, ist XML Schema nicht ausreichend. Auch weitere XML-Technologien wie z.B. Schematron bieten nicht die notwendige Ausdrucksstärke, um beispielsweise transitive Beziehungen zu modellieren. Hierbei zeigt sich die Stärke von OWL.

Ein weiterer Vorteil von OWL ist, dass es sich mit SWRL kombinieren lässt. Dies ermöglicht die Formalisierung von Richtlinien, denen der EI-Planungsprozess unterliegt. Diese Richtlinien liegen verbalsprachlich vor. Eine Formalisierung dieser Richtlinien mit SWRL erlaubt jedoch deren Nutzung in einem automatischen Verifikationssystem unter Beibehaltung einer hohen Verständlich- und Nachvollziehbarkeit.

3.3 Debugging - Konzeptbeschreibung

Es ist allgemein bekannt, dass der Begriff Debugging [37] üblicherweise dazu verwendet wird, den Prozess der Fehlerbeseitigung in einem Programm zu beschreiben. Dieser Prozess beinhaltet verschiedene Tätigkeiten wie beispielsweise die schrittweise Programmausführung, die Inspektion von Variablen und Speicherinhalten oder auch die Erzeugung von zusätzlichen Programmausgaben. Auch die Modifikation von Programmcode mit dem Ziel der Fehlerbeseitigung sowie die erneute Kompilierung und Revision des Programmablaufs stellen Tätigkeiten dar, die in der Regel zum Debugging-Prozess zugehörig sind.

Trotz dieser eindeutigen Begriffsdefinition wurde im Rahmen dieser Arbeit der Begriff des Debuggings für einen abweichenden Prozess vereinnahmt. Debugging, im Sinne der vorliegenden Arbeit, beschreibt das Auffinden von fehlerhaften oder nicht explizit deklarierten Eigenschafts-Zuweisungen in der ABox einer Ontologie.

Der in dieser Arbeit beschriebene Debugging-Ansatz ist konzeptionell vom Autor erarbeitet worden. Die darauf basierende Idee der Semantic Constraints wurde im Rahmen einer entsprechend betreuten Diplomarbeit [36] als Plugin für den Protégé Ontologie-Editor implementiert. In diesem Kapitel wird das Konzept vorgestellt und erörtert. Technische Details sowie die Implementierung des Plugins werden in Kapitel 5 beschrieben.

Eine grundlegende Konzeptbeschreibung ist der Abbildung 3.2 zu entnehmen. Es wird davon ausgegangen, dass eine EI-OWL-ABox vorhanden ist, die konform ist mit ent-

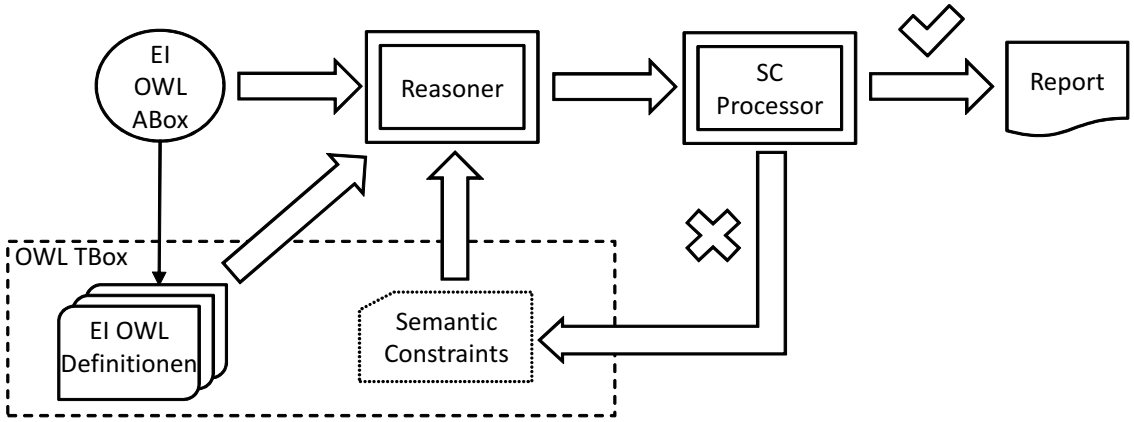

Abbildung 3.2: Grundlegendes Konzept des Debuggings

sprechenden EI-OWL-Definitionen. Zusammen mit den Semantic Constraints bilden diese Definitionen die TBox der Wissensbasis. Der Abbildung nicht zu entnehmen ist die Tatsache, dass die EI-OWL-ABox wie im Ansatz zur Verifikation (vgl. Kapitel 3.1) aus einer XML-Datenbasis durch XSL-Transformation generiert worden ist. Diese Tatsache wurde aufgrund der Übersichtlichkeit nicht in die Abbildung integriert.

Die A- und TBox werden von einem Reasoner verarbeitet. Dabei werden die SWRL-Regeln der SCs ausgewertet. Eine Software, hier als SC-Processor bezeichnet, kontrolliert, ob alle zu überprüfenden Individuen durch die SWRL-Regeln der SCs entsprechend klassifiziert werden konnten. Ist dies nicht der Fall, kann darauf geschlossen werden, dass Fehler in den Axiomen der Individuen vorliegen. Um diese Fehler einzugrenzen, wird vom SC-Processor ein Reduktionsprozess der entsprechenden SWRL-Regeln angstoßen. Dabei werden schrittweise einzelne Regelatome entfernt und durch eine neue Verarbeitung mithilfe des Reasoner geprüft, ob die fehlerhaften Individuen den Antezedenz-Teil der reduzierten Regel erfüllen. Tritt diese Situation ein, werden diese Individuen im Konsequenz-Teil der Regel entsprechend markiert und die reduzierten Atome mit ihnen in Beziehung gesetzt. Diese Atome stellen die potentiellen Fehlerursachen dar. Sind alle Individuen durch die (zum Teil abgeschwächten) SWRL-Regeln der SCs klassifiziert worden, endet der Reduktionsprozess. Vom SC-Processor wird ein Report generiert und dem Benutzer präsentiert. Der SC-Processor ist, wie im vorigen Teil dieses Kapitels beschrieben, als Plugin für den Ontologie-Editor Protégé realisiert.

3.4 Debugging - Kontroverse

Der Ansatz des Ontologie-Debuggings mithilfe von Semantic Constraints ist begleitet mit Problematiken, die in den folgenden Unterkapiteln identifiziert und deren Lösungsansätze im Kapitel 5 detailliert beschrieben werden.

3.4.1 Große Anzahl an Subconstraints

Die Skalierung des Systems ist eine Problematik, die behandelt werden muss: Die SWRL-Regel eines SCs besteht aus n Atomen. Im Laufe eines Reduktionsprozesses können daraus theoretisch 2^n abgeleitete Regeln (und Sub-SC = SSCs) entstehen. Dies ist bezogen auf die Skalierbarkeit ein unerwünschtes Verhalten. Eine Abschwächung dieser Problematik ergibt sich aus folgenden Vorgehensweisen: Zum einen muss bei der Definition eines SC immer $1-x$ Regel-Atome als *necessary* angegeben werden. Sie sind notwendig für die Sinnhaftigkeit der Regel und werden im Reduktionsprozess nicht entfernt. Zum anderen wurden Heuristiken identifiziert, bei denen eine Entfernung bestimmter Arten von Atomen bevorzugt wird. Dadurch kann eine signifikante Verringerung der Anzahl von SSCs erreicht werden. Diese Verfahren werden im Unterkapitel 5.2.3.2 beschrieben.

3.4.2 Häufige Wiederholungen von Constraintteilen

Für jeden Aspekt der Verifikation von EI sollte mindestens ein SC erstellt werden, um ein Debugging zu ermöglichen. Hieraus ergibt sich die Problematik des generell hohen Umfangs von SCs in einer Wissensbasis. Dies erschwert die Les- und Wartbarkeit des Systems und beinhaltet somit ein hohes Frustrationspotential bei dessen Verwendung. Hinzu addiert sich die Tatsache, dass bei der Formulierung von SCs, die sich auf eine einzige Klasse von Individuen beziehen, es häufig zu Wiederholungen in Teilen der den SCs unterliegenden SWRL-Regeln kommen kann. Dies erschwert weiter die Les- und Wartbarkeit und bildet eine generelle Fehlerquelle. Ein Lösungsansatz, der in dieser Arbeit vorgestellt wird, ist ein Hierarchisierungsmechanismus. Dieser erlaubt die Referenzierung von SCs in anderen SCs. Erst wenn diese sogenannten *Preconditions*, also die referenzierten SCs, erfüllt sind, wird der Haupt-SC ausgewertet. Hiermit können SC 'wiederverwendet' werden, und die Komplexität des Gesamtsystems verringert sich. Dieser Ansatz wird detailliert in Kapitel 5.2.4 diskutiert.

3.4.3 Aussagefähigkeit

Generell ist die Aussagefähigkeit eines solchen Debugging-Systems zu hinterfragen. Wenn ein Individuum gegen einen SC verstößt und als Debugging-Resultat entsprechende Atome der SWRL-Regel des SCs als potentielle Fehlerursachen präsentiert werden, liegt es in der Verantwortung des Benutzers, diese möglichen Fehlerursachen zu untersuchen. Das Debugging-System kann lediglich einen Hinweis liefern und so die Zeitaufwand bei einer Fehlersuche minimieren. Eine konkrete Aussage über die tatsächliche Fehlerursache kann nur eingeschränkt von einem Debugging-System formuliert werden. Sie hängt von vielen Faktoren ab. Die Wesentlichen sind die Struktur und der Aufbau der Wissensbasis und somit letztendlich das Modellierungsgeschick des Erstellers, aber auch die Formulierung und die Komposition der SCs. Es lassen sich durchaus Parallelen zum Debugging konventioneller Programme ziehen. Auch hier hängt ein schnelles und erfolgreiches Debugging stark von der Struktur des Programmcodes ab. Ebenso kann ein angezeigter Fehler auch nur Rückschlüsse auf die Fehlerursache geben. Die Behebung eines Fehler kann jedoch je nach Szenario eine außerordentlich komplizierte Tätigkeit darstellen.

4 Semantische Modellierung und Verifikation von Eisenbahn-Infrastrukturen

In den nachfolgenden Kapiteln wird der Aufbau des semantischen Modells von Eisenbahn-Infrastrukturen mithilfe von OWL-Ontologien und SWRL-Regeln sowie deren Verwendung zur Verifikation konkreter Eisenbahn-Infrastrukturen erläutert.

4.1 Semantische Modellierung zu Verifikationszwecken

Zur Verifikation von EI-Planungsdaten wird eine Kombination von OWL und SWRL eingesetzt werden. Hierzu wird die vorhandene und bereits beschriebene Werkzeugpalette von Editoren zur Modellierung und Reasonern zur eigentlichen Verifikation verwendet. Aufgrund des logischen Fundaments, der Beschreibungslogik, können mit diesen Technologien komplexe Zusammenhänge auf logische Korrektheit überprüft werden.

Die OWL unterliegende OWA erschwert eine Datenverifikation, da grundsätzlich von einem unvollständigen Datenmodell ausgegangen wird. Mit einigen Kunstgriffen lässt sich die OWA-Problematik jedoch umgehen. So kann beispielsweise die genaue Menge an Individuen einer OWL-Klasse durch die Verwendung von *ObjectOneOf*-Ausdrücke festgelegt werden. So kann die OWA, die standardmäßig von weiteren, noch nicht bekannten Individuen ausgeht, in diesem Teilbereich umgangen werden.

Der Ausdruck zur Beschreibung der genauen Menge von Individuen der Klasse `Signal` gemäß Abb. 4.1 ist beispielsweise:

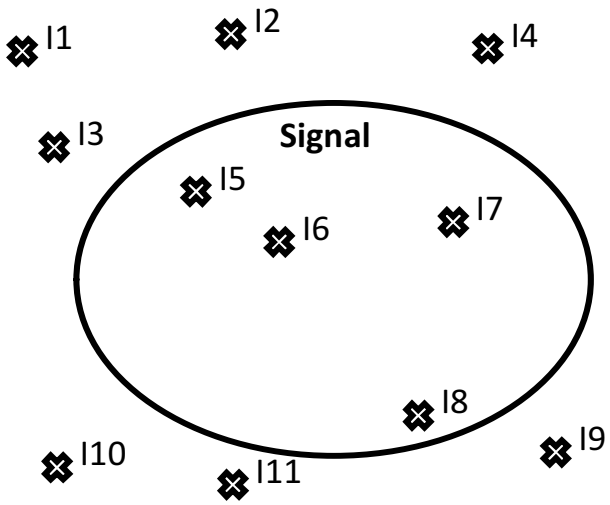

Abbildung 4.1: Menge der Individuen der Klasse `Signal`

```
1 EquivalentClasses(:Signal ObjectOneOf(:I5 :I6 :I7 :I8))
```

Listing 4.1: OneOf-Ausdruck zur Beschränkung der Menge von Individuen einer Klasse

Durch die konsequente Anwendung der *ObjectOneOf*-Ausdrücke ist es möglich, die für die Datenverifikation nachteilige OWA auszugleichen, da so von einer genauen Menge von Individuen einer jeweiligen Klasse im Sinne der CWA ausgegangen werden kann.

In OWL greift die UNA nicht und somit können Individuen nicht als voneinander verschieden angesehen werden, selbst wenn sie unterschiedliche Namen haben. Jedoch stellt OWL unterschiedliche Möglichkeiten zur Verfügung, um zu beschreiben, dass konkrete Individuen voneinander verschieden sind. Hierzu existiert zum einen die Möglichkeit der Angabe eines *DifferentIndividuals* Statements. Die Verwendung dieses Statements impliziert jedoch, dass jedes Individuum explizit als verschieden zu anderen Individuen deklariert werden muss. Dies hat im Falle einer Erweiterung der Ontologie einen hohen Wartungsaufwand zur Folge, und, wie in [30] beschrieben, ist auch von einem Performance-Verlust im Reasoning-Prozess auszugehen. Eine diesbezüglich elegantere Lösung ist die Verwendung des in OWL 2 neu eingeführten *Key*-Mechanismus [43]. Ein key Axiom der Form $HasKey(CE(OPE_1 \ldots OPE_m)(DPE_1 \ldots DPE_n))$ gibt an, dass jedes benannte Individuum des Klassenausdrucks CE eindeutig identifiziert wird. Diese Identifikation erfolgt über die object property Ausdrücke OPE_i und/oder die data property Ausdrücke DPE_j. Sind dieses Ausdrücke in mindestens einem Wert verschieden, kann geschlossen werden, dass die entsprechenden

Individuen verschieden sind.

Im Hinblick auf die Datenverifikation ist ein weiterer Aspekt der semantischen Modellierung von Vorteil: SWRL ist grundsätzlich unentscheidbar und kann demzufolge nur unzureichend praktisch eingesetzt werden. Damit dieses Defizit ausgeglichen wird, arbeiten SWRL-fähige Reasoner mit einer entscheidbaren Untermenge der SWRL-Sprachspezifikation. Diese Untermenge wird in der OWL Functional Syntax mit dem Schlüsselwort `DLSafeRule` beschrieben. Sie ist in der Prädikatenlogik erster Stufe (der das verwendete OWL DL ebenso unterliegt) entscheidbar. Die Einschränkung besteht darin, dass die Variablen-Belegungen von SWRL-Atomen auf benannte Individuen beschränkt sind. Es können also keine anonymen/unbekannten Individuen in den Belegungen auftreten. Dies ist für eine Datenverifikation jedoch von Vorteil, da somit eine Klassifikation im Sinne der CWA ermöglicht wird. Etwaig vorhandene, aber nicht explizit deklarierte Individuen werden von SWRL-Regeln nicht berücksichtigt. SWRL kann so unter anderem dazu verwendet werden, Verifikationsregeln zu formulieren, die eine Einteilung in verifizierbare und nicht verifizierbare, also nicht vollständig spezifizierte, Elemente einer EI ermöglichen.

SWRL bietet hinausgehend über die eben genannten Möglichkeiten zur Verifikation noch weitere Vorteile: In den Planungsrichtlinien der Deutschen Bahn wird unter anderem auf die korrekte Anordnung von EI-Elemente sowie deren Abstände und konkreten Bezeichnungen eingegangen. Beispiele hierfür sind zum einen Vorsignale, die in bestimmten Fällen vor Hauptsignale positioniert werden müssen, oder die konkrete Länge von Durchrutschwegen in Abhängigkeit von möglichen Zuggeschwindigkeiten. Zum anderen müssen laut den Planungsrichtlinien die Bezeichner von Vorsignalen immer mit *'VA-'* beginnen und die Bezeichner von Weichen immer mit dem Buchstaben *'W'*. Die zur Prüfung von korrekten Abständen notwendigen mathematischen Berechnungen oder Vergleiche von Zeichenketten um korrekte Bezeichnungen zu ermitteln, ist jedoch mit OWL nicht möglich. Hierfür bietet SWRL die Möglichkeit der Verwendung von Builtins. Mit ihnen können Berechnungen, Vergleiche sowie die Verarbeitung von Zeichenketten durchgeführt werden. Die Prüfung solcher Planungsdetails wird somit mit SWRL-Regeln vollzogen. Sämtliche SWRL-Regeln sind in einer eigenen Ontologie gespeichert. Eine ausführliche Erläuterung dieser Ontologie ist in Kapitel 4.4 zu finden.

Die Aufteilung der Wissensbasis in mehrere Ontologien ermöglicht eine hohe Transparenz und vereinfacht die Erweiterbarkeit. Im Rahmen dieser Arbeit wurde ein modulares ontologisches Schichtenmodell entwickelt und implementiert. Die Hierarchie der Ontologien ist in Abbildung 4.2 dargestellt.

Der Name der Ontologie-Sammlung - *RI** (gesprochen: *railway infrastructure star*) - bezeichnet die Erweiterung der Darstellungsmöglichkeit von EI mithilfe eines in sich

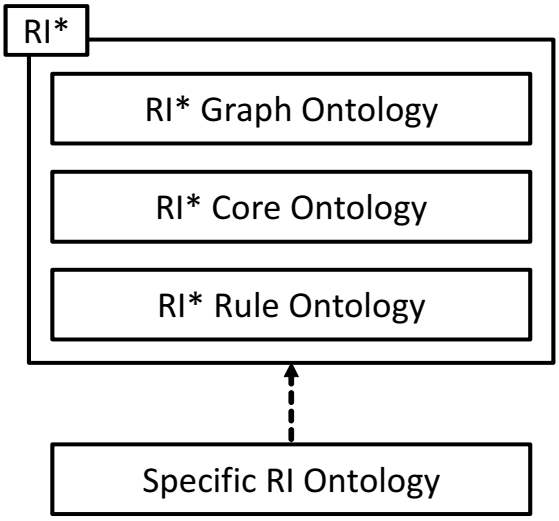

Abbildung 4.2: Ontologie Schichtenmodell

schlüssigen, formalen semantischen Modells. Zur RI*-Ontologie-Sammlung gehören die folgenden drei Ontologien:

- ***RI* Graph Ontology*** - Die Basis-Ontologie beschreibt die Klassen und Eigenschaften eines Graphen. Dieser ist unabhängig von Eisenbahn-Infrastrukturen, jedoch bildet diese Ontologie die Grundlage für die Verifikation fundamentaler Verbindungs- und Befahrbarkeitsvorschriften von EI.

- ***RI* Core Ontology*** - Innerhalb dieser Ontologie wird die Eisenbahndomäne ausführlich modelliert. Hier werden alle Klassen, Eigenschaften und Interrelationen von physikalischen und logischen Elementen von EI definiert.

- ***RI* Rule Ontology*** - Diese Ontologie beinhaltet zum einen SWRL-Regeln, in denen verschiedene Aspekte der Verifikationsvorschriften abgebildet werden. Zum anderen sind hier sogenannte Verifikationsklassen integriert, die in einem Teil der SWRL-Regeln verwendet werden.

Neben den drei Ontologien der RI*-Ontologie-Sammlung ist in Abbildung 4.2 im unteren Bereich die *Specific RI Ontology* dargestellt. Sie ist ein Repräsentant für eine konkrete EI, die einem Verifikationsprozess unterzogen werden soll. Sie stellt eine ABox dar und verwendet hierzu die Klassen und Eigenschaften aus den RI*-Ontologien. Der Verifikationsprozess selbst wird durch Anwendung eines OWL/SWRL-Reasoners und der entsprechenden Auswertung durch eine Softwarekomponente beschrieben. Der Reasoner verwendet dabei die TBox, bestehend aus den drei *RI**-Ontologien und die aus der *Specific RI Ontology* bestehenden ABox mit konkreten Planungsda-

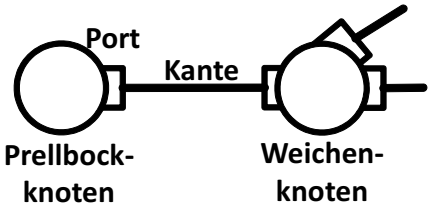

Abbildung 4.3: Zusammenhang zwischen `Edge`, `Port` und `Vertex`

ten und unterzieht diese gesamte Wissensbasis dem reasoning-Prozess. Dabei werden auf der Basis der Beschreibungslogik neue explizite Axiome aus den impliziten Herleitung gebildet und die explizite Wissensbasis somit erweitert.

In den folgenden Kapiteln werden die einzelnen Ontologien, die Strategien bei ihrer Modellierung sowie die sich daraus ergebende Verifikationsmethodik ausführlich beschrieben.

4.2 RI* Graph Ontology

Grundsätzlich können EI als Netz aus Knoten und Kanten in Form eines Graphen betrachtet werden. Somit ergibt sich auf der untersten Abstraktionsebene die Möglichkeit einen Graphen zu modellieren, der von EI unabhängig ist. Diese Vorgehensweise bietet den Vorteil, dass auf dieser tiefsten Ebene schon grundlegende Verbindungseigenschaften EI-unabhängig analysiert werden können. Die EI-spezifischen Konzepte, definiert in der *RI* Core Ontology*, werden in einer höheren Abstraktionsschicht auf diesem Basisgraphen modelliert.

Die *RI* Graph Ontology* beinhaltet somit, neben anderen Konstrukten, Klassen und Eigenschaften für Knoten (`Vertex`), Kanten (`Edge`) und Anschlüsse (`Port`). Der Eisenbahndomäne entsprechend repräsentieren Kanten Blockabschnitte, während Knoten für RI-Elemente wie Weichen und Prellböcke stehen, welche die Blockabschnitte begrenzen. Knoten und Kanten werden nicht direkt miteinander verbunden, sondern über `Ports` (vgl. Abbildung 4.3).

Anschlüsse sind als Bestandteil von Knoten modelliert. Die Modellierung des Anschlusskonzepts bildet eine Grundlage zur Unterscheidung verschiedener Anschlusstypen. So können beispielsweise die Abzweigungen einer Weiche als solche klassifiziert werden. Dies geschieht jedoch auf einer höheren Ebene, in der *RI* Core Ontology*. Es sei folgendes Beispiel genannt: Ein Prellbock ist in der Regel ein Gleisabschluss

eines Kopfbahnhofs und somit ein Knotenelement mit nur einem verbundenen Gleisabschnitt. Eine Weiche hingegen ist ein Knotenelement, das mindestens eine Abzweigung von einem Hauptgleis ermöglicht und somit über Verbindungen zu mindestens drei Gleisabschnitten verfügt. Es existieren auch Weichentypen mit mehr als drei Verbindungen zu Gleisabschnitten. Unter den Voraussetzungen, dass jeder Anschluss jeweils mit genau einer Kante verbunden wird, und dass unterschiedliche Knotentypen über eine unterschiedliche Anzahl an Anschlüssen verfügen, kann der vollständige Zusammenhang der Elemente des Graphen und somit die grundlegende topologische Korrektheit gewährleistet werden.

Formal beschrieben sei V eine Menge von Kanten, E eine Menge von Knoten und P eine Menge von Anschlüssen. Für diese Mengen gilt:

Für alle v aus der Menge der Individuen der Klasse V `Vertex` existiert eine Menge von Individuen der Klasse P `Port`. Die Anzahl dieser Individuen beträgt n, wobei n größer oder gleich 1 ist.

$$\forall\, v \in V\ \exists\, p \subseteq P\ \mid\ |p| = n\ \mid\ n \geq 1 \tag{4.1}$$

Für alle p aus der Menge der Individuen der Klasse P `Port` existiert genau ein e aus der Menge der Individuen der Klasse E `Edge`.

$$\forall\, p \in P\ \exists!\ e \in E \tag{4.2}$$

Für alle e aus der Menge der Individuen der Klasse E `Edge` existieren die beiden disjunkten Individuen p_i und p_j der Klasse `Port`.

$$\forall\, e \in E\ \exists\, p_i, p_j\ \in\ P\ \mid\ p_i\ \neq\ p_j \tag{4.3}$$

In OWL definieren wir den Zusammenhang zwischen Anschlüssen und Knoten über die functional object property `portDocksEdge` und die dazu inverse Eigenschaft `edgeIsDockedByPort`:

```
1  Declaration(ObjectProperty(:portDocksEdge))
2  InverseObjectProperties(:portDocksEdge :edgeIsDockedByPort)
3  FunctionalObjectProperty(:portDocksEdge)
4  ObjectPropertyDomain(:portDocksEdge :Port)
5  ObjectPropertyRange(:portDocksEdge :Edge)
6
7  Declaration(ObjectProperty(:edgeIsDockedByPort))
8  InverseObjectProperties(:edgeIsDockedByPort :portDocksEdge)
9  FunctionalObjectProperty(:edgeIsDockedByPort)
10 ObjectPropertyDomain(:edgeIsDockedByPort :Edge)
```

```
11 ObjectPropertyRange(:edgeIsDockedByPort :Port)
```

Listing 4.2: `portDocksEdge` und `edgeIsDockedByPort`: Zusammenhang zwischen Anschlüssen und Knoten

Der Zusammenhang zwischen Kanten und Anschlüssen ist über die Eigenschaften `vertexHasPort` und `portIsAtVertex` definiert:

```
1  Declaration(ObjectProperty(:portIsAtVertex))
2  InverseObjectProperties(:portIsAtVertex :vertexHasPort)
3  FunctionalObjectProperty(:portIsAtVertex)
4  ObjectPropertyDomain(:portIsAtVertex :Port)
5  ObjectPropertyRange(:portIsAtVertex :Vertex)
6
7  Declaration(ObjectProperty(:vertexHasPort))
8  InverseObjectProperties(:vertexHasPort :portIsAtVertex)
9  FunctionalObjectProperty(:vertexHasPort)
10 ObjectPropertyDomain(:vertexHasPort :Vertex)
11 ObjectPropertyRange(:vertexHasPort :Port)
```

Listing 4.3: `vertexHasPort` und `portIsAtVertex`: Zusammenhang zwischen Anschlüssen und Kanten

Nun können diese Eigenschaften so verknüpft werden, dass vom Reasoner Verbindungsfehler innerhalb des Graphen entdeckt werden können. Hierzu wird das in OWL 2 eingeführte Konzept der Eigenschaftsketten (property chains) genutzt. Wie in [57] beschrieben, ist es in OWL 1 nicht möglich, Eigenschaften aus einer Komposition von anderen Eigenschaften zu definieren. Mit dem OWL 2-Konstrukt *ObjectPropertyChain* in einem *SubObjectProperty*-Axiom ist dies dagegen möglich. Eine Eigenschaftskette, die die oben deklarierten EI-Eigenschaften verknüpft, um die topologische Korrektheit des Graphen zu gewährleisten, ist folgendermaßen definiert:

```
1 SubObjectPropertyOf(ObjectPropertyChain(:vertexHasPort :portDocksEdge
    :edgeIsDockedByPort :portIsAtVertex) :vertexConnectVertex)
```

Listing 4.4: Eigenschaftskette: `vertexConnectVertex`

Neben den grundlegenden Graphenelementen wie Knoten und Kanten, die Gleise, Weichen etc. repräsentieren, müssen ebenfalls sicherheitstechnische Elemente wie Signale, Achszähler usw. abgebildet werden können. Konzeptionell werden diese Elemente als Anhänge an Knoten und Kanten betrachtet und in der *RI* Graph Ontology* mit der Klasse `Attachment` modelliert. Um zu ermöglichen sowohl Knoten als auch Kanten Anhänge zuzuordnen, wurden die entsprechenden Klassen `Edge` und `Vertex` in einer gemeinsamen Oberklasse `BaseModel` zusammengefasst. Dieser kann nun über die *object property* `baseModelHasAttachment` eine beliebige Anzahl von

Anhängen zugeordnet werden. Umgekehrt kann Anhängen über die *object property* `attachmentIsOfBaseModel` ein Individuum der Klasse `BaseModel` zugeordnet werden.

`Attachment` ist eine Oberklasse für die zwei disjunkten Unterklassen `DirectedAttachment` und `UndirectedAttachment`. Hierbei wurde der Tatsache entsprochen, dass zum einen sicherheitstechnische Elemente existieren, die eine bestimmte Wirkrichtung aufweisen. Hierzu zählen beispielsweise alle Signale. Diese können grundsätzlich nur aus einer Fahrtrichtung wahrgenommen werden. Zum anderen existiert die Gruppe der wirkrichtungslosen Elemente wie beispielsweise Achszähler. Jene detektieren jede Achse eines sie überfahrenden Zuges, unabhängig von der Richtung aus der er sich genähert hat. Um nun die Wirkrichtung eines gerichteten Anhangs zu modellieren, wurde die Klasse `AttachmentDirection` entworfen. Sie ist eine Äquivalenzklasse zur Menge der beiden disjunkten Individuen `Ascending` (Wirkrichtung aufsteigend) und `Descending` (Wirkrichtung absteigend). Aufsteigend und absteigend beziehen sich dabei auf einen definierten Ursprungspunkt des Modells einer spezifischen EI.

4.3 RI* Core Ontology

4.3.1 Einleitung

Die *RI* Core Ontology* beinhaltet eine Klassenhierarchie EI-spezifischer Konzepte, ihre Eigenschaften sowie deren wechselseitige Beziehungen. Es werden annähernd 50 Eisenbahnelemente mit insgesamt ca. 80 data und ca. 80 object properties modelliert. Die *RI* Core Ontology* beinhaltet, wie auch die *RI* Graph Ontology* und die *RI* Rule Ontology*, nur terminologische Fakten. Sie beschreiben also die TBox des wissensbasierten Systems zur Verifikation von EI. Somit werden keine Klassen- oder Eigenschaftszuweisungen von konkreten, zu verifizierenden Individuen beschrieben.

Hierzu sei an dieser Stelle hinzugefügt, dass die zu verifizierenden Individuen konkreter Planungsdaten in der *Specific RI Ontology* (vgl. Kapitel 4.5) gespeichert werden. Aus diesen Planungsdaten im XML-Format werden die entsprechenden Individuen in der *Specific RI Ontology* durch eine XSL-Transformation erzeugt. Hierbei werden nur die Basis-Individuen wie beispielsweise `Track` (siehe 4.3.2.1) oder `Signal` (siehe 4.3.3.1) erzeugt. Eine weitere Spezialisierung zu beispielsweise `OpenTrack` / `StationTrack` bzw. `MainSignal` / `DistantSignal` erfolgt durch die Anwendung eines Reasoners. Durch diese Spezialisierung können letztendlich die Regelausdrücke in der *RI* Rule Ontology* verkürzt und somit die Komplexität vermindert werden. Im Rahmen der Beschreibung der *RI* Core Ontology* wird lediglich auf die Basisklassen (z.B. `Track`) eingegangen,

da die Ableitung der spezialisierten Klassen (z.B. `OpenTrack`) aus den entsprechenden *properties* zu Aufzählungsklassen (z.B. `hasTrackType` = open) trivial ist.

Im Folgenden werden einzelne Designentscheidungen bei der Erstellung der *RI* Core Ontology* detailliert erläutert sowie ein Großteil der Konzepte und ihrer Eigenschaften beschrieben.

Die grundlegende Basisklasse der *RI* Core Ontology* und damit aller modellierten Elemente einer EI, die spezifiziert und verifiziert werden sollen, ist die Klasse `BaseIdName`. Sie ermöglicht durch Verwendung eines key Axioms (vgl. Kapitel 4.1) die eindeutige Identifizierbarkeit der einzelnen Individuen. Hierzu wird die Basisklasse über eine funktionale *data property* mit einer Id bestehend aus Literalen verknüpft. Diese *data property* `hasId` wird in einem `HasKey` Axiom verwendet und der Klasse `BaseIdName` zugeordnet (siehe Listing 4.5).

```
1 Declaration(Class(:BaseIdName))
2 EquivalentClasses(:BaseIdName
3   ObjectIntersectionOf(DataSomeValuesFrom(:hasId xsd:string) owl:Thing))
4
5 FunctionalDataProperty(:hasId)
6 DataPropertyDomain(:hasId :BaseIdName)
7 DataPropertyRange(:hasId xsd:string)
8
9 HasKey(:BaseIdName () (:hasId))
```

Listing 4.5: Deklaration der Basisklasse `BaseIdName`; Verwendung des `HasKey` Axioms zur eindeutigen Identifizierbarkeit ihrer Individuen

Konzeptuell können alle EI-Elemente grundsätzlich in zwei Gruppen unterteilt werden. Eine Gruppe beschreibt diejenigen Elemente, die den Graphen des Schienennetzes ausprägen. Sie sind konzeptionell eine Abstraktionsschicht über den in der *RI* Graph Ontology* deklarierten Klassen `Edge` und `Vertex` angeordnet und weisen eine räumliche (im Modell: zweidimensionale) Ausdehnung auf. Auf Elementen dieser Gruppe finden Zugbewegungen statt. Zu ihnen zählen unter anderem Gleisabschnitte und Prellböcke sowie sämtliche Weichentypen.

Im Gegensatz zu dieser ersten Gruppe der Elemente mit räumlicher Ausdehnung ist die Gruppe der punktförmigen Elemente definiert. In dieser Gruppe sind sämtliche Elemente enthalten, die an oder auf der Gleisstrecke lokalisiert sind. In der Regel sind dies Elemente zur Zugdetektion oder -beeinflussung. Beispiele hierfür sind Signale, Balisen, Achszähler usw. Diese Elemente haben eine vernachlässigbare räumliche Ausdehnung und zählen somit zu den punktförmigen (im Modell: nulldimensionalen) Elementen. Im Modell werden ebenso die unterschiedlichen Typen von Bahnübergängen als punktförmige Objekte angesehen, da für eine Verifikation im Sinne der

Planungsrichtlinien die räumliche Ausdehnung von Bahnübergängen faktisch unbedeutend ist.

Aus Modellierungssicht enthält die *RI* Core Ontology* die Basisklasse `BaseIdName`, von der die beiden oben beschrieben Subklassen `ZeroDimObject` und `TwoDimObject` abgeleitet werden. `TwoDimObject` ist die Superklasse der Klassen aller Elemente mit räumlicher Ausdehnung wie `Track` (Gleisabschnitt), `Switch` (Weiche) und `BufferStop` (Prellbock) sowie ihrer Unterklassen. Bezogen auf die *RI* Graph Ontology* sind die Klassen `BufferStop` und `Switch` direkte Unterklassen von `Vertex`. Sie stellen somit Knotenelemente des Graphen dar. `Track` hingegen ist direkte Subklasse von `Edge` der *RI* Graph Ontology* und stellt somit das Kantenelement auf einer anderen Abstraktionsstufe dar.

4.3.2 Eisenbahn-Infrastrukturelemente mit räumlicher Ausdehnung

Wie in Abschnitt 4.3.1 beschrieben, ist im Modellierungsansatz der *RI* Core Ontology* die Gruppe der EI-Elemente mit räumlicher Ausdehnung (Klasse: `TwoDimObject`) enthalten. Zum einen zählen hierzu die Gleisabschnitte (Klasse: `Track`) sowie die unterschiedlichen Weichentypen. Zum anderen werden neben den Weichen auch die anderen Knotentypen wie `TransitionPoint` und `BufferStop` zur Klasse `TwoDimObject` gezählt, da sie über eine räumliche Ausdehnung verfügen und auf ihnen Zugbewegungen stattfinden. Nachfolgend werden die einzelnen Klassen und ihre Eigenschaften erläutert.

Die Klasse `TwoDimObject` beinhaltet eine *property*-Zuordnung zu `isRelatedTo` bei der als *range* ein Individuum der Klasse `BlockSection` angegeben werden kann. `BlockSections` - Blockabschnitte - sind Bereiche einer Zugstrecke, die aus Stellwerkssicht exklusiv für einen Zug reserviert werden. Dies hat sicheheitstechnische Relevanz (vgl. Kapitel 4.3.4.4).

Die grundlegende Eigenschaft von `TwoDimObjects` ist, dass sie miteinander verbunden werden können und als Verbund ein Gleisnetz darstellen. Diese Verbindungsmöglichkeit wird mit der irreflexiven und transitiven *property* `connect` dargestellt. `connect` besitzt sowohl als *domain* als auch als *range* die Klasse `TwoDimObject`. Sie ist eine transitive *property*, weil nicht nur eine Verbindung von Nachbarelementen darzustellen ist; Vielmehr ist die Verbindung eines jeden Elements im Graphen des Gleisnetzes mit jedem anderen Element in demselben Graphen möglich. Die *property* ist irreflexiv, da gewährleistet werden muss, dass kein Element mit sich selbst über die `connect` *property* verbunden werden kann.

Weitere *properties* von `TwoDimObject` sind die *data properties* `hasIncline` und `hasDecline`, die eine Steigung oder ein Gefälle in aufsteigender Kilometrierungsrichtung beschreiben. Der Wert wird in Prozent angegeben und hat Auswirkungen auf die Berechnung von Distanzen wie z.B. der Länge von Durchrutschwegen (vgl. 4.3.4.6).

Eine essentiell wichtige *property* ist die irreflexive *property* `directConnection`. Sie kennzeichnet, dass `TwoDimObjects` direkt miteinander verknüpft sind. Irreflexiv ist sie deshalb, weil ausgeschlossen werden muss, dass ein `TwoDimObject` mit sich selbst verknüpft werden kann. `directConnection` ist eine *sub property* von der *property chain* `vertexConntecVertex` aus der *RI* Graph Ontology* und ermöglicht auf dieser höheren Abstraktionsebene die Darstellung der topographischen Verknüpfung von Elementen mit räumlicher Ausdehnung als Schienennetz. Eine Besonderheit von `directConnection` ist, dass sie erst durch die Auswertung einer entsprechenden SWRL-Regel (genauer: einer *property*-Regel) belegt wird. Diese Methodik wird in Kapitel 4.4.3 näher beschrieben.

Eine weitere Charakteristik von Objekten mit räumlicher Ausdehnung auf denen Zugbewegungen stattfinden ist die der Befahrbarkeitsangabe. Über die *property* `hasTraffic-Direction` kann angegeben werden, ob Züge in aufsteigender oder absteigender Kilometrierungsrichtung über dieses Element manövrieren dürfen. Des Weiteren ist es möglich, über die Angabe des Individuums `both` bei Richtungen zuzulassen.

```
1  Declaration(Class(:TwoDimObject))
2  SubClassOf(:TwoDimObject :BaseIdName)
3
4  Declaration(ObjectProperty(:isRelatedTo))
5  InverseObjectProperties(:relatesTo :isRelatedTo)
6  InverseFunctionalObjectProperty(:isRelatedTo)
7  ObjectPropertyDomain(:isRelatedTo :TwoDimObject)
8  ObjectPropertyRange(:isRelatedTo :BlockSection)
9
10 Declaration(DataProperty(:hasDecline))
11 FunctionalDataProperty(:hasDecline)
12 DataPropertyDomain(:hasDecline :TwoDimObject)
13 DataPropertyRange(:hasDecline xsd:int)
14
15 Declaration(DataProperty(:hasIncline))
16 FunctionalDataProperty(:hasIncline)
17 DataPropertyDomain(:hasIncline :TwoDimObject)
18 DataPropertyRange(:hasIncline xsd:int)
19
20 Declaration(ObjectProperty(:connect))
21 TransitiveObjectProperty(:connect)
22 IrreflexiveObjectProperty(:connect)
23 ObjectPropertyDomain(:connect :TwoDimObject)
24 ObjectPropertyRange(:connect :TwoDimObject)
25
```

```
26 Declaration(ObjectProperty(:directConnection))
27 SubObjectPropertyOf(:directConnection :vertexConnectVertex)
28 IrreflexiveObjectProperty(:directConnection)
29 ObjectPropertyDomain(:directConnection :TwoDimObject)
30 ObjectPropertyRange(:directConnection :TwoDimObject)
31
32 Declaration(Class(:TrafficDirection))
33 SubClassOf(:TrafficDirection ObjectIntersectionOf(ObjectOneOf(
       :Descending :Ascending :both) :Enums))
34
35 Declaration(ObjectProperty(:hasTrafficDirection))
36 FunctionalObjectProperty(:hasTrafficDirection)
37 ObjectPropertyDomain(:hasTrafficDirection :TwoDimObject)
38 ObjectPropertyRange(:hasTrafficDirection :TrafficDirection)
```

Listing 4.6: Deklaration der Oberklasse `TwoDimObject` sowie der zugehörigen *property* `isRelatedTo`

4.3.2.1 Gleisabschnitt

Die Klasse `Track` beschreibt das Konzept der Gleisabschnitte. Sie definiert Kantenelemente und hat somit neben der Oberklasse `TwoDimObject` ebenfalls eine Vererbungsbeziehung zur Oberklasse `Edge`. Gleisabschnitte verfügen über genau zwei Anschlusspunkte und werden somit als Äquivalenzklasse zu `Edge` mit genau zwei Zuweisungen der `edgeIsDockedByPort` *property* (vgl. Listing 4.8 Zeile 2) angesehen. Gleisabschnitte haben diverse *property* Zuweisungen. Neben der Befahrbarkeitsrichtung (*object property*: `trackHasDirection`) kann ein Gleisabschnitt (je Richtung) unterschiedliche Maximalgeschwindigkeiten aufweisen (*data properties*: `maxSpeedIn` und `maxSpeedOut`). Diese beiden *properties* erben von der *super property* `maxSpeed`. Diese wird verwendet, wenn eine einheitliche Maximalgeschwindigkeit für beide Befahrbarkeitsrichtungen existiert. Die Befahrbarkeitsrichtung `trackHasDirection` ist eine spezialisierte *sub property* der allgemeinen Eigenschaft zur Befahrbarkeitsangabe `hasDirection`. Diese ist ein Aufzählungstyp und kann lediglich die Individuenzuweisung `out, in` oder `both` aufweisen (vgl. Listing 4.7).

```
1 Declaration(Class(:Direction))
2 SubClassOf(:Direction ObjectIntersectionOf(ObjectOneOf(:out :in :both)
      :Enums))
3
4 Declaration(NamedIndividual(:both))
5 ClassAssertion(:Direction :both)
6 Declaration(NamedIndividual(:in))
7 ClassAssertion(:Direction :in)
8 Declaration(NamedIndividual(:out))
9 ClassAssertion(:Direction :out)
```

Listing 4.7: Deklaration der Aufzählungsklasse `Direction` sowie der zugehörigen Individuen

Neben diesen *properties* besitzt ein `Track` jeweils eine Kilometrierungsangabe für den Anfang (`begin`) und das Ende (`end`). Diese haben den Datentyp `long` als *range*, da die Kilometrierungsangaben einen hohen Wert aufweisen können. Des Weiteren verfügt die Klasse `Track` über die *property* `hasTrackType`. Ihr Wert ist ein Individuum der Aufzählungsklasse `TrackType` (`open` oder `station`). Dies hat zur Folge, dass zwischen Gleisabschnitten in Bahnhöfen und Gleisabschnitten der freien Strecke unterschieden werden kann. Die Deklaration der Klasse `Track` sowie die relevanten *properties* sind in Listing 4.8 dargestellt.

```
1  Declaration(Class(:Track))
2  EquivalentClasses(:Track ObjectIntersectionOf(ObjectExactCardinality(2
       :edgeIsDockedByPort :Port) :Edge))
3  SubClassOf(:Track :TwoDimObject)
4
5  Declaration(ObjectProperty(:trackHasDirection))
6  SubObjectPropertyOf(:trackHasDirection :hasDirection)
7  FunctionalObjectProperty(:trackHasDirection)
8  ObjectPropertyDomain(:trackHasDirection :Track)
9  ObjectPropertyRange(:trackHasDirection :Direction)
10
11 Declaration(DataProperty(:maxSpeedIn))
12 SubDataPropertyOf(:maxSpeedIn :maxSpeed)
13 FunctionalDataProperty(:maxSpeedIn)
14 DataPropertyDomain(:maxSpeedIn :OrdinarySwitch)
15 DataPropertyDomain(:maxSpeedIn :Track)
16 DataPropertyRange(:maxSpeedIn xsd:int)
17
18 Declaration(DataProperty(:maxSpeedOut))
19 SubDataPropertyOf(:maxSpeedOut :maxSpeed)
20 FunctionalDataProperty(:maxSpeedOut)
21 DataPropertyDomain(:maxSpeedOut :OrdinarySwitch)
22 DataPropertyDomain(:maxSpeedOut :Track)
23 DataPropertyRange(:maxSpeedOut xsd:int)
24
25 Declaration(DataProperty(:begin))
26 FunctionalDataProperty(:begin)
27 DataPropertyDomain(:begin :OrdinarySwitch)
28 DataPropertyDomain(:begin :Track)
29 DataPropertyRange(:begin xsd:long)
30
31 Declaration(DataProperty(:end))
32 FunctionalDataProperty(:end)
33 DataPropertyDomain(:end :OrdinarySwitch)
34 DataPropertyDomain(:end :Track)
```

```
35 DataPropertyRange(:end xsd:long)
36
37 Declaration(Class(:TrackType))
38 SubClassOf(:TrackType ObjectIntersectionOf(ObjectOneOf(:station :open)
       :Enums))
39 Declaration(NamedIndividual(:open))
40 ClassAssertion(:TrackType :open)
41 Declaration(NamedIndividual(:station))
42 ClassAssertion(:TrackType :station)
43
44 Declaration(ObjectProperty(:hasTrackType))
45 FunctionalObjectProperty(:hasTrackType)
46 ObjectPropertyDomain(:hasTrackType :Track)
47 ObjectPropertyRange(:hasTrackType :TrackType)
```

Listing 4.8: Deklaration der Klasse `Track` sowie entsprechender Eigenschaften

4.3.2.2 Weiche

Es existieren verschiedene Arten von Weichen. Sie werden in der Klasse `Switch` zusammengefasst. `Switch` wiederum erbt von der Klasse `TwoDimObject`, beschreibt also ein Element mit räumlicher Ausdehnung. Neben dieser Klasse ist `Switch` einer Unterklasse von `FlankProtectionElement`. Eine Weiche kann - wie auch ein Signal (vgl. Kapitel 4.3.3.1) - als Flankenschutzelement zur Sicherung von Fahrstraßen verwendet werden. Die Flankenschutzthematik wird in Kapitel 4.3.4.7 beschrieben.

Die verschiedenen Arten von Weichen werden als Subklassen von `Switch` modelliert und teilweise durch sogenannte *qualified cardinality restrictions* definiert. Bezugnehmend auf eine Veröffentlichung des W3C zu dem Thema [56] sind *qualified cardinality restrictions* Einschränkungsmöglichkeiten, die sich auf die Anzahl der möglichen *object properties* zu einer Klasse beziehen. Herauszustellen ist hierbei, dass im Gegensatz zu *unqualified cardinality restrictions* auch der *range* Typ der einzuschränkenden *object property* angegeben werden muss.

Als Beispiel sei hierzu Listing 4.9 zur Deklaration der Oberklasse `Switch` aufgeführt. Der Code in Zeile 5 ist so zu lesen, dass eine Weiche dadurch gekennzeichnet ist, ein Knotenelement (Klasse `Vertex` aus der *RI* Graph Ontology*) zu sein, dass über mindestens drei *object properties* `vertexHasPort` mit dem Typ `Port` verfügt und somit mindestens drei Verbindungen ermöglicht:

```
1 Declaration(Class(:Switch))
2 SubClassOf(:Switch :TwoDimObject)
3 SubClassOf(:Switch :FlankProtectionElement)
4
```

```
5 EquivalentClasses(:Switch ObjectIntersectionOf(ObjectMinCardinality(3
    :vertexHasPort :Port) :Vertex))
```

Listing 4.9: Qualified cardinality restriction zur Beschreibung der Oberklasse `Switch`

Alle anderen Weichentypen erben von dieser Oberklasse `Switch` und übernehmen die Einschränkung der *qualified cardinality restriction*, da sämtliche Weichentypen über mindestens eine Abzweigung vom Hauptgleis verfügen. Die Klasse `OrdinarySwitch`, die die Standardweiche repräsentiert, beinhaltet eine weitere Einschränkung, in der die Auflage von genau einer Abzweigung erteilt wird, siehe Listing 4.10.

```
1 Declaration(Class(:OrdinarySwitch))
2 SubClassOf(:OrdinarySwitch :Switch)
3
4 EquivalentClasses(:OrdinarySwitch ObjectIntersectionOf(
    ObjectExactCardinality(3 :vertexHasPort :Port) :Vertex))
```

Listing 4.10: Qualified cardinality restriction zur Beschreibung der Standardweiche

Eine Sonderform stellen Kreuzungsweichen dar. Hier treffen wie bei einer Straßenkreuzung zwei Gleisstränge zusammen. Dieses Weichenelement verfügt somit über vier Verbindungspunkte (siehe Listing 4.11).

```
1 Declaration(Class(:SlipSwitch))
2 SubClassOf(:OrdinarySwitch :Switch)
3
4 EquivalentClasses(:SlipSwitch ObjectIntersectionOf(
    ObjectExactCardinality(4 :vertexHasPort :Port>) :Vertex))
```

Listing 4.11: Qualified cardinality restriction zur Beschreibung der Kreuzungsweiche

Grundsätzlich wird zwischen einfachen und doppelten Kreuzungsweichen unterschieden. Der wesentliche Unterschied besteht in den Befahrbarkeitsmöglichkeiten. So ist bei einer einfachen Kreuzungsweiche lediglich das Abbiegen eines Zuges von einem Gleis zu einem abzweigenden Gleis möglich, wohingegen bei einer doppelten Kreuzungsweiche ein Abbiegen von beiden Gleisen zu beiden Abzweigungen möglich ist.

```
1 ...
2 SubClassOf(:OrdinarySwitch ObjectSomeValuesFrom(:vertexHasPort :Branch))
3 SubClassOf(:OrdinarySwitch ObjectSomeValuesFrom(:vertexHasPort :Tip))
4 SubClassOf(:OrdinarySwitch ObjectSomeValuesFrom(:vertexHasPort :Trunk))
5 SubClassOf(:OrdinarySwitch ObjectAllValuesFrom(:vertexHasPort
    ObjectUnionOf(:Branch :Tip :Trunk)))
6 ...
```

Listing 4.12: Constraint für die Ports der `OrdinarySwitch`

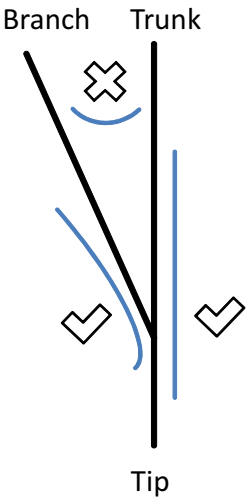

Abbildung 4.4: Schematische Darstellung der Porttypen und Befahrbarkeitsmöglichkeiten einer Standardweiche

Befahrbarkeitsrichtlinien werden über Ports definiert. Hierzu werden von der Hauptklasse `Port` die entsprechenden Subklassen `Trunk`, `Tip` und `Branch` abgeleitet. Die `OrdinarySwitch` bekommt nun als weitere Einschränkung zugeordnet, dass jeder dieser Porttypen enthalten sein muss. Dies ist in Listing 4.12 dargestellt. Diese Einschränkung besagt, dass ein `OrdinarySwitch` die Zuweisung eines Individuums der Klasse `Branch` über die *property* `vertexHasPort` enthalten muss. Das gleiche gilt für die Klassen `Tip` und `Branch`. Die letzte Aussage des Listings in Zeile 5 besagt, dass eine Zuweisung der *property* `vertexHasPort` nur Individuen der Klassen `Branch`, `Tip` sowie `Trunk` enthalten darf. Diese Modellierungstechnik wird als *Closure Axiom* bezeichnet. Zusammen mit der Aussage des Listings 4.10, dass die Standardweiche exakt drei Portzuweisungen beinhalten muss, wird so gewährleistet, dass dieses je genau ein Individuum der Klassen `Trunk`, `Tip` bzw. `Branch` ist. Die Einführung der unterschiedlichen Porttypen erklärt sich vor dem Hintergrund der Befahrbarkeitsmöglichkeiten. Eine Standardweiche kann nur über die Wege `Tip-Trunk`, `Tip-Branch` oder über die jeweilige Gegenrichtung befahren werden. Ein Befahrung `Trunk-Branch` oder `Branch-Trunk` ist nicht möglich. Die Befahrbarkeit wird für jeden Blockabschnitt einer entsprechenden Route geprüft. Hierbei werden die Verbindungen zu den Nachbarblockabschnitten hinzugezogen, um eine mögliche Befahrbarkeit zu bestätigen oder zu widerlegen.

Bei der Kreuzungsweiche bzw. der doppelten Kreuzungsweiche wird ähnlich vorgegangen. Hierbei werden jedoch Unterklassen von `Tip` verwendet. `TipOutLeft` und `TipOutRight` werden von der Unterklasse `TipOut` abgeleitet, während `TipInLeft` und `TipInRight` von der Unterklasse `TipIn` abgeleitet werden. `Out` und `In` beziehen sich

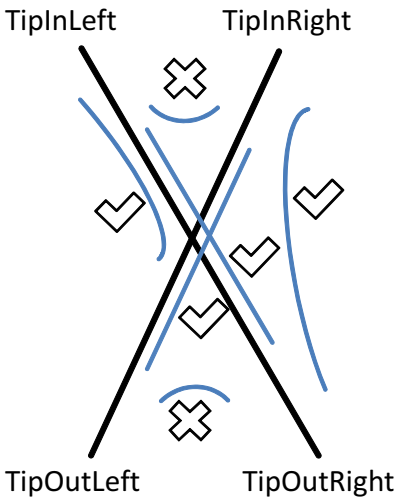

Abbildung 4.5: Schematische Darstellung der Porttypen und Befahrbarkeitsmöglichkeiten einer doppelten Kreuzungsweiche

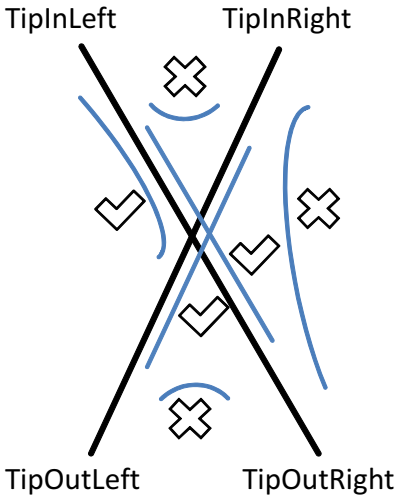

Abbildung 4.6: Schematische Darstellung der Porttypen und Befahrbarkeitsmöglichkeiten einer einfachen Kreuzungsweiche

dabei auf die aufsteigende respektive absteigende Kilometrierungsrichtung der EI. Bei der doppelten Kreuzungsweiche sind die Befahrbarkeitsmöglichkeiten dahingehend eingeschränkt, dass nicht von `TipOutLeft` nach `TipOutRight` und andersherum, respektive von `TipInLeft` nach `TipInRight` und andersherum gefahren werden kann (siehe Abbildungen 4.6 und 4.5). Dies wird gewährleistet, indem bei der Überprüfung der Vorgänger oder Nachfolger eines Blockabschnitts, dem solch eine Weiche unterliegt, sichergestellt wird, dass weder `Out`-Klassen mit `Out`-Klassen noch `In`-Klassen mit `In`-Klassen verbunden sind.

Bei der einfachen Kreuzungsweiche kommt noch die Einschränkung hinzu, dass eine Seite (entweder `Tip[In/Out]Left` oder `Tip[In/Out]Right`) nicht von `TipIn[Left/Right]` nach `TipOut[Left/Right]` befahren werden kann. Um diese Situation zu modellieren wurde die Mehrfachvererbung von OWL genutzt. Die Klassen `TipOutLeft` und `TipInLeft` werden der gemeinsamen Oberklasse `Left` zugeordnet. Analog dazu wird mit `TipInRight` und `TipOutRight` verfahren, die eine Zuordnung zu einer gemeinsamen Oberklasse `Right` erhalten. Nun ist es möglich, über eine entsprechende *object property* `trafficable` einem Individuum der einfachen Kreuzungsweiche `SingleSlipSwitch` zuzuordnen, dass eine Befahrbarkeit entweder über die linke oder die rechte Flanke erlaubt ist.

4.3.2.3 Objekte mit einfacher Verbindung

Zu den zweidimensionalen Objekten zählen neben `Track`, `Switch` und den entsprechenden Unterklassen auch EI-Abschlusselemente. Sie verfügen lediglich über einen Verbindungspunkt (`Port`: `End`) und erben von der Klasse `SingleConnectionObject` (vgl. Listing 4.13). Für diese Klasse ist eine Position als Kilometrierungsangabe auf dem Gleisnetz erforderlich. Hierzu ist die *data property* `position` definiert.

```
1  Declaration(Class(:SingleConnectionObject))
2  EquivalentClasses(:SingleConnectionObject ObjectIntersectionOf(
       ObjectExactCardinality(1 :vertexHasPort :Port) :Vertex))
3  EquivalentClasses(:SingleConnectionObject ObjectIntersectionOf(
       DataExactCardinality(1 :position xsd:long) :TwoDimObject))
4
5  SubClassOf(:SingleConnectionObject :TwoDimObject)
6  SubClassOf(:SingleConnectionObject ObjectSomeValuesFrom(:vertexHasPort
       :End))
7  SubClassOf(:SingleConnectionObject ObjectAllValuesFrom(:vertexHasPort
       :End))
8
9  Declaration(DataProperty(:position))
10 FunctionalDataProperty(:position)
11 DataPropertyDomain(:position :SingleConnectionObject)
```

```
DataPropertyDomain(:position :ZeroDimObject)
DataPropertyRange(:position xsd:long)
```

Listing 4.13: Deklaration der Klasse `SingleConnectionObject`

4.3.2.4 Prellbock

Eine der beiden Unterklassen von `SingleConnectionObject` ist `BufferStop`. Sie ermöglicht die Modellierung von Prellböcken, die beispielsweise in Kopfbahnhöfen den Gleisabschluss bilden. Auch in Rangierbereichen werden üblicherweise Prellböcke zum Abschluss verwendet. In der Praxis existieren unterschiedliche Typen von Prellböcken. Die Typenzuordnung erfolgt über die funktionale *object property* `hasBufferStopType`. Ein Objekt mit genau einer Zuweisung dieser *property* wird gleichgesetzt mit einem Prellbock. Die Deklaration dieser *property* sowie der Klasse `BufferStop` wird in Listing 4.14 dargestellt.

```
Declaration(Class(:BufferStop))
SubClassOf(:BufferStop :SingleConnectionObject)
EquivalentClasses(:BufferStop ObjectIntersectionOf(
    ObjectExactCardinality(1 :hasBufferStopType :BufferStopTypes)
    :SingleConnectionObject))

Declaration(ObjectProperty(:hasBufferStopType))
FunctionalObjectProperty(:hasBufferStopType)
ObjectPropertyDomain(:hasBufferStopType :BufferStop)
ObjectPropertyRange(:hasBufferStopType :BufferStopTypes)

Declaration(Class(:BufferStopTypes))
SubClassOf(:BufferStopTypes ObjectIntersectionOf(ObjectOneOf(
    :brakingBufferStop :absorbingBufferStop :staticBufferStop) :Enums))
Declaration(NamedIndividual(:absorbingBufferStop))
ClassAssertion(:BufferStopTypes :absorbingBufferStop)
Declaration(NamedIndividual(:brakingBufferStop))
ClassAssertion(:BufferStopTypes :brakingBufferStop)
Declaration(NamedIndividual(:staticBufferStop))
ClassAssertion(:BufferStopTypes :staticBufferStop)
```

Listing 4.14: Deklaration der Klasse `BufferStop` sowie der zugehörigen *property*

4.3.2.5 Modellierungsgrenze

Die zweite Unterklasse von `SingleConnectionObject` ist die Klasse `TransitionPoint`. Sie beschreibt Modellierungsgrenzen, also Punkte, die den Rahmen des Modells abste-

cken. Hier endet üblicherweise der Zuständigkeitsbereich einer Bedienzentrale, welche ein oder mehrere Stellwerke steuert. Die Klasse `TransitionPoint` verfügt neben den geerbten *properties* lediglich über die *object property* `neighbourInterlockingCenter`, über die ein Nachbarstellwerk angegeben werden kann (vgl. Listing 4.15).

```
Declaration(Class(:TransitionPoint))
SubClassOf(:TransitionPoint :SingleConnectionObject)
EquivalentClasses(:TransitionPoint ObjectIntersectionOf(
    ObjectExactCardinality(1 :neighbourInterlockingCenter
    :InterlockingCenter) :SingleConnectionObject))
Declaration(ObjectProperty(:neighbourInterlockingCenter))
SubObjectPropertyOf(:neighbourInterlockingCenter
    :controlledByInterlockingCenter)
FunctionalObjectProperty(:neighbourInterlockingCenter)
ObjectPropertyDomain(:neighbourInterlockingCenter :TransitionPoint)
ObjectPropertyRange(:neighbourInterlockingCenter :InterlockingCenter)
```

Listing 4.15: Deklaration der Klasse `TransitionPoint` sowie der zugehörigen *property*

4.3.3 Punktförmige Eisenbahn-Infrastrukturelemente

Eisenbahn-Infrastrukturen werden zum einen durch die Elemente beschrieben, auf denen Zugbewegungen stattfinden. Diese Elemente wurden durch Unterklassen von `TwoDimObject` modelliert und im vorigen Kapitel beschrieben. Um die Zugbewegungen kontrollieren und beeinflussen zu können, sind jedoch noch weitere Elemente nötig. Diese sicherungstechnischen Elemente, wie beispielsweise Signale, werden in der Regel punktuell auf oder an der Zugstrecke platziert. Zur Modellierung dieser punktförmigen Elemente mit OWL wurden diese in der Klasse `ZeroDimObject` (vgl. Listing 4.16) zusammengefasst und als jeweilige Unterklassen definiert. Über die *property* `on` werden `ZeroDimObjects` mit `TwoDimObjects` verknüpft. Somit wird der Zusammenhang zwischen punktförmigen Objekten wie beispielsweise Signalen und ihrer Position auf Elementen des Gleisnetzes hergestellt. Die Modellierung der wesentlichen Elemente wird in den folgenden Unterkapiteln beschrieben.

```
Declaration(Class(:ZeroDimObject))
EquivalentClasses(:ZeroDimObject ObjectIntersectionOf(
    DataExactCardinality(1 :position xsd:long) :Attachment))
SubClassOf(:ZeroDimObject :BaseIdName)

Declaration(ObjectProperty(:on))
FunctionalObjectProperty(:on)
ObjectPropertyDomain(:on :ZeroDimObject)
ObjectPropertyRange(:on :TwoDimObject)
```

Listing 4.16: Deklaration der Klasse `ZeroDimObject`

4.3.3.1 Signal

Signale stellen die wichtigste sicherungstechnische Einrichtung im Schienenverkehr dar. Sie sind für eine Vielzahl von Aufgaben zur Steuerung von Zugbewegungen zuständig. So existieren Signale für die Anzeige von Maximalgeschwindigkeiten, zur Regelung des Rangierverkehrs, zur Sperrung und Freigabe eines Streckenabschnitts und Vielem mehr. Signale sind grundsätzlich als Handlungsanweisung für den Triebfahrzeugführer zu verstehen. Er beeinflusst durch seine Handlung die Zugbewegung unmittelbar. Teilweise können jedoch Signale auch mit anderen sicherungstechnischen Elementen gekoppelt sein. So ist es üblich, dass, wenn ein Zug an ein 'Halt'-zeigendes Signal überfährt, eine Notbremsung des Zuges durch einen entsprechend geschalteten Gleismagneten hinter dem Signal ausgelöst wird.

Grundsätzlich wird ein Signal als Unterklasse von `ZeroDimObject` mit einer Reihe von notwendigen Eigenschaftszuweisungen modelliert. Dies ist in Listing 4.17 dargestellt. In Zeile 3 und 4 ist erkennbar, dass ein `Signal` als eine Unterklasse sowohl zu der Klasse `ZeroDimObject` als auch zu der in Abschnitt 4.2 beschriebenen Klasse `DirectedAttachment` modelliert ist. Das resultiert daraus, dass Signale an den dem Eisenbahnnetz unterliegenden Graphen 'angeheftet' werden und eine definierte Wirkrichtung besitzen. Diese Wirkrichtung wird über die *property* `directedAttachment-HasDirection` der *RI* Graph Ontology* ausgedrückt. Als weitere Oberklasse von `Signal` fungiert die Klasse `FlankProtectionElement`. Ein Signal kann - wie auch eine Weiche (vgl. Kapitel 4.3.2.2) - als Flankenschutzelement zur Sicherung von Fahrstraßen verwendet werden. Die Flankenschutzthematik wird in Kapitel 4.3.4.7 behandelt.

Für die Klasse `Signal` sind weitere *properties* notwendig. So ist beispielsweise die boolsche *property* `isSwitchable` definiert, die die Stellbarkeit eines Signals beschreibt. Diese *property* ist ab Zeile 14 beschrieben. Zeile 2 des Listings besagt, dass ein Signal-Individuum grundsätzlich jeweils genau eine Zuweisung zu den *properties* `hasSignal-Function`, `hasSignalType` sowie `hasSignallingSystem` beinhalten muss.

Des Weiteren kann ein Signal als Start- oder Endpunkt von Fahrstraßen fungieren. Dies ist über die *data properties* `isStartingSignalOfRoute` und `isEndingSignalOfRoute` definiert. Eine weitergehende Beschreibung über Signale als Fahrstraßenelemente ist in Kapitel 4.3.4.5 zu finden.

```
1 Declaration(Class(:Signal))
2 EquivalentClasses(:Signal ObjectIntersectionOf(:ZeroDimObject
    ObjectExactCardinality(1 :hasSignalFunction :SignalFunction)
    ObjectExactCardinality(1 :hasSignalType :SignalType)
    ObjectExactCardinality(1 :hasSignallingSystem :SignallingSystem)))
3 SubClassOf(:Signal :ZeroDimObject)
4 SubClassOf(:Signal :DirectedAttachment)
```

```
5  SubClassOf(:Signal :FlankProtectionElement)
6
7  SubClassOf(:Signal ObjectSomeValuesFrom(:hasSignalFunction
       :SignalFunction))
8  SubClassOf(:Signal ObjectSomeValuesFrom(:hasSignalType :SignalType))
9  SubClassOf(:Signal ObjectSomeValuesFrom(:hasSignallingSystem
       :SignallingSystem))
10
11 SubClassOf(:Signal ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignallingSystem :HI) ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignalType ObjectUnionOf(:Distant :Main :Repeater :Shunting))
       ObjectComplementOf(:Combined))))
12 SubClassOf(:Signal ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignallingSystem :HV) ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignalType ObjectUnionOf(:Distant :Main :Repeater :Shunting))
       ObjectComplementOf(:Combined))))
13 SubClassOf(:Signal ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignallingSystem :SV) ObjectIntersectionOf(ObjectAllValuesFrom(
       :hasSignalType ObjectUnionOf(:Distant :Main :Repeater :Shunting))
       ObjectComplementOf(:Combined))))
14
15 Declaration(DataProperty(:isSwitchable))
16 FunctionalDataProperty(:isSwitchable)
17 DataPropertyDomain(:isSwitchable :Signal)
18 DataPropertyRange(:isSwitchable xsd:boolean)
19
20 Declaration(ObjectProperty(:isStartingSignalOfRoute))
21 InverseObjectProperties(:routesStartingSignal :isStartingSignalOfRoute)
22 ObjectPropertyDomain(:isStartingSignalOfRoute :Signal)
23 ObjectPropertyRange(:isStartingSignalOfRoute :Route)
24
25 Declaration(ObjectProperty(:isEndingSignalOfRoute))
26 InverseObjectProperties(:routesEndingSignal :isEndingSignalOfRoute)
27 ObjectPropertyDomain(:isEndingSignalOfRoute :Signal)
28 ObjectPropertyRange(:isEndingSignalOfRoute :Route)
```

Listing 4.17: Deklaration der Klasse `Signal` und ihrer notwendigen *property*-Zuweisungen

Um darzustellen, dass Signale vom Typ 'Kombinationssignal' nur im KS-Signalsystem genutzt werden können, werden die OWL-Konstrukte `ObjectIntersectionOf`, `ObjectAllValuesFrom` sowie `ObjectComplementOf` verwendet (vgl. Zeilen 10-14 im Listing 4.17). Um die Voraussetzung für diese Art der Modellierung zu schaffen, sind die entsprechenden Signaltypen und Signalsysteme nicht als Individuen, sondern als Unterklassen modelliert. (vgl. Listing 4.18).

```
Declaration(ObjectProperty(:hasSignalType))
FunctionalObjectProperty(:hasSignalType)
ObjectPropertyDomain(:hasSignalType :Signal)
ObjectPropertyRange(:hasSignalType :SignalType)

Declaration(Class(:Combined))
SubClassOf(:Combined :SignalType)
Declaration(Class(:Distant))
SubClassOf(:Distant :SignalType)
Declaration(Class(:Main))
SubClassOf(:Main :SignalType)
Declaration(Class(:Repeater))
SubClassOf(:Repeater :SignalType)
Declaration(Class(:Shunting))
SubClassOf(:Shunting :SignalType)

Declaration(ObjectProperty(:hasSignallingSystem))
FunctionalObjectProperty(:hasSignallingSystem)
ObjectPropertyDomain(:hasSignallingSystem :Signal)
ObjectPropertyRange(:hasSignallingSystem :SignallingSystem)

Declaration(Class(:HI))
SubClassOf(:HI :SignallingSystem)
Declaration(Class(:HV))
SubClassOf(:HV :SignallingSystem)
Declaration(Class(:KS))
SubClassOf(:KS :SignallingSystem)
Declaration(Class(:SV))
SubClassOf(:SV :SignallingSystem)
```

Listing 4.18: Deklaration der *object properties* `hasSignalType` und `hasSignallingSystem` sowie der entsprechenden Unterklassen

Die *object property* `hasSignalFunction` legt fest, in welcher Funktion ein Signal verwendet wird. Die zugehörige Klasse `SignalFunction` ist als Aufzählungsklasse modelliert (vgl. Listing 4.19). Sie beinhaltet die Signalfunktionen Blocksignal, Zwischensignal sowie Ein- bzw. Ausfahrtsignal als OWL-Individuen.

```
Declaration(ObjectProperty(:hasSignalFunction))
FunctionalObjectProperty(:hasSignalFunction)
ObjectPropertyDomain(:hasSignalFunction :Signal)
ObjectPropertyRange(:hasSignalFunction :SignalFunction)

Declaration(Class(:SignalFunction))
SubClassOf(:SignalFunction ObjectIntersectionOf(ObjectOneOf(:Home
    :Intermediate :Exit :Blocking) :Enums))

Declaration(NamedIndividual(:Blocking))
AnnotationAssertion(rdfs:comment :Blocking "Blocksignal")
```

```
11 ClassAssertion(:SignalFunction :Blocking)
12 Declaration(NamedIndividual(:Exit))
13 AnnotationAssertion(rdfs:comment :Exit "Ausfahrsignal")
14 ClassAssertion(:SignalFunction :Exit)
15 Declaration(NamedIndividual(:Home))
16 AnnotationAssertion(rdfs:comment :Home "Einfahrsignal")
17 ClassAssertion(:SignalFunction :Home)
18 Declaration(NamedIndividual(:Intermediate))
19 AnnotationAssertion(rdfs:comment :Intermediate "Zwischensignal")
20 ClassAssertion(:SignalFunction :Intermediate)
```

Listing 4.19: Deklaration der *object property* `hasSignalFunction` sowie der zugehörigen Aufzählungsklasse

4.3.3.2 Achszähler

Achszähler gehören neben Balisen zur Gruppe der Zugdetektionselemente. Achszähler werden zur Abgrenzung von Blockabschnitten (siehe Kapitel 4.3.4.4) verwendet und zählen ebenso wie Signale zu den sicherungstechnischen Einrichtungen von Eisenbahnstrecken. Durch einen meist induktiven Prozess können einzelne Achsen eines Zuges detektiert werden.

Aus Stellwerkssicht sind Achszähler wichtig, um eine Rückmeldung über die Belegung oder Nichtbelegung von Blockabschnitten durch Züge zu ermöglichen. Aufgrund dieser Informationen können daraufhin zentral im Stellwerk Fahrstraßen reserviert oder freigegeben werden.

In Listing 4.20 wird die Modellierung eines Achszählers dargestellt. Die Klasse `AxleCounter` wird von den Klassen `TrainDetectionElements` sowie `UndirectedAttachment` abgeleitet. Achszähler-Individuen müssen eine Eigenschaft zu ihrer Typenzuweisung (`hasAxleCounterType`) haben. Des Weiteren muss über eine boolsche *data property* (`detectsDirection`) definiert sein, ob dieser Achszähler auch die Fahrtrichtung eines Zuges detektieren kann.

Über eine optionale *data property* (`maxSpeed`) kann definiert werden, ob der entsprechende Achszähler über eine Geschwindigkeitsbegrenzung verfügt.

```
1  Declaration(Class(:AxleCounter))
2  EquivalentClasses(:AxleCounter ObjectIntersectionOf(
       :TrainDetectionElement ObjectExactCardinality(1 :hasAxleCounterType
       :AxleCounterType) DataExactCardinality(1 :detectsDirection
       xsd:boolean)))
3  SubClassOf(:AxleCounter :TrainDetectionElement)
4  SubClassOf(:AxleCounter :UndirectedAttachment)
5
6  Declaration(ObjectProperty(:hasAxleCounterType))
7  FunctionalObjectProperty(:hasAxleCounterType)
8  ObjectPropertyDomain(:hasAxleCounterType :AxleCounter)
9  ObjectPropertyRange(:hasAxleCounterType :AxleCounterType)
10
11 Declaration(Class(:AxleCounterType))
12 SubClassOf(:AxleCounterType ObjectIntersectionOf(ObjectOneOf(:inductive
       :forcedField) :Enums))
13
14 Declaration(DataProperty(:detectsDirection))
15 FunctionalDataProperty(:detectsDirection)
16 DataPropertyDomain(:detectsDirection :AxleCounter)
17 DataPropertyRange(:detectsDirection xsd:boolean)
18
19 Declaration(DataProperty(:maxSpeed))
20 DataPropertyDomain(:maxSpeed :AxleCounter)
21 DataPropertyDomain(:maxSpeed :TwoDimObject)
22 DataPropertyRange(:maxSpeed xsd:int)
```

Listing 4.20: Deklaration der Klasse `AxleCounter` und ihrer *properties*

4.3.3.3 Gleismagnet

Gleismagnete sind Zugbeinflussungselemente. Sie werden dazu genutzt, einen Zug auszubremsen, der ein 'Halt'-zeigendes Signal überfährt. Des Weiteren werden Gleismagnete, auch als PZB (Punktförmige Zugbeeinflussung) oder Indusi bezeichnet, zur Geschwindigkeitsverringerung in Gefahrenbereichen eingesetzt. Gleismagnete haben keine konkrete Wirkrichtung, funktionieren also unidirektional.

Der entsprechenden Verbreitung Rechnung tragend, wird in diesem Modellierungsansatz lediglich zwischen vier verschiedenen Typen von Gleismagneten unterschieden: Magnetsysteme mit 500Hz, 1000Hz, 2000Hz oder einer Kombination aus unterschiedlichen Frequenzen. Dies wird durch die Zuweisung der *property* `hasFieldType` zur Klasse der Gleismagnete `TrainProtectionElement` gewährleistet. Des Weiteren besteht die Möglichkeit, verschiedene Energieversorgungssysteme über die *property* `hasPowerSupply` zu modellieren. Zur Auswahl stehen hierbei die Optionen netzgespeist

(*property*-Zuweisung: `grid`) oder solargespeist (*property*-Zuweisung: `solar`).

```
Declaration(Class(:TrainProtectionElement))
SubClassOf(:TrainProtectionElement :ZeroDimObject)

Declaration(Class(:FieldType))
SubClassOf(:FieldType ObjectIntersectionOf(ObjectOneOf(:combinedField
    :_1000Hz :_500Hz :_2000Hz) :Enums))
Declaration(NamedIndividual(:_1000Hz))
ClassAssertion(:FieldType :_1000Hz)
Declaration(NamedIndividual(:_2000Hz))
ClassAssertion(:FieldType :_2000Hz)
Declaration(NamedIndividual(:_500Hz))
ClassAssertion(:FieldType :_500Hz)
Declaration(NamedIndividual(:combinedField))
ClassAssertion(:FieldType :combinedField)
Declaration(ObjectProperty(:hasFieldType))
FunctionalObjectProperty(:hasFieldType)
ObjectPropertyDomain(:hasFieldType :TrainProtectionElement)
ObjectPropertyRange(:hasFieldType :FieldType)

Declaration(Class(:PowerSupply))
SubClassOf(:PowerSupply ObjectIntersectionOf(ObjectOneOf(:solar :grid)
    :Enums))
Declaration(NamedIndividual(:grid))
ClassAssertion(:PowerSupply :grid)
Declaration(NamedIndividual(:solar))
ClassAssertion(:PowerSupply :solar)
Declaration(ObjectProperty(:hasPowerSupply))
FunctionalObjectProperty(:hasPowerSupply)
ObjectPropertyDomain(:hasPowerSupply :TrainProtectionElement)
ObjectPropertyRange(:hasPowerSupply :PowerSupply)
```

Listing 4.21: Deklaration der Klasse `TrainProtectionElement` und ihrer *properties*

4.3.3.4 Bahnübergang

Bahnübergänge werden über die Klasse `LevelCrossing` modelliert und können über die *property* `hasLevelCrossingProtection` hinsichtlich der Sicherung näher beschrieben werden. Zur Auswahl stehen hierbei die Optionen Halbschranken, Vollschranken, rein akustische Kennzeichnung, Kennzeichnung durch Lichtsignalanlage, eine Kombination aus akustischer Kennzeichnung und Lichtsignalanlage sowie ohne technische Sicherung (hierbei erfolgt die Sicherung durch Beschilderung). Grundsätzlich sind beschränkte Bahnübergänge zusätzlich sowohl durch eine akustische als auch durch Lichtsignalanlage gesichert. Somit ist bei beschränkten Bahnübergängen die Angabe

der Lichtsignal-/Akustik-Sicherung redundant. Aus diesem Grund ist die entsprechende *property* als funktional deklariert.

Die Deklaration der Klasse `LevelCrossing` sowie der zugehörigen *property* und deren Enumerationsklasse sind in Listing 4.22 dargestellt.

```
Declaration(Class(:LevelCrossing))
SubClassOf(:LevelCrossing :ZeroDimObject)

Declaration(Class(:LevelCrossingProtection))
SubClassOf(:LevelCrossingProtection ObjectIntersectionOf(ObjectOneOf(
    :acousticLightProtection :half-barrierProtection :acousticProtection
     :full-barrierProtection :nonTechnicalProtection) :Enums))
Declaration(NamedIndividual(:acousticLightProtection))
ClassAssertion(:LevelCrossingProtection :acousticLightProtection)
Declaration(NamedIndividual(:acousticProtection))
ClassAssertion(:LevelCrossingProtection :acousticProtection)
Declaration(NamedIndividual(:full-barrierProtection))
ClassAssertion(:LevelCrossingProtection :full-barrierProtection)
Declaration(NamedIndividual(:half-barrierProtection))
ClassAssertion(:LevelCrossingProtection :half-barrierProtection)
Declaration(NamedIndividual(:lightProtection))
ClassAssertion(:LevelCrossingProtection :lightProtection)
Declaration(NamedIndividual(:nonTechnicalProtection))
ClassAssertion(:LevelCrossingProtection :nonTechnicalProtection)
Declaration(ObjectProperty(:hasLevelCrossingProtection))
FunctionalObjectProperty(:hasLevelCrossingProtection)
ObjectPropertyDomain(:hasLevelCrossingProtection :LevelCrossing)
ObjectPropertyRange(:hasLevelCrossingProtection :LevelCrossingProtection
    )
```

Listing 4.22: Deklaration der Klasse `LevelCrossing` und der entsprechenden *property*

4.3.3.5 Balise

Balisen (Klasse: `Balise`) sind kleine Transponder, die meist auf den Schwellen zwischen den Schienen eines Gleisnetzes verankert sind. Sie dienen dazu, Züge, die über eine entsprechende Empfangstechnik verfügen, mit spezifischen Informationen in Form von digitalen Diagrammen zu versorgen. Dies können beispielsweise Anweisungen für die Neigetechnik der Züge bei einem bevorstehenden kurvigen Streckenverlauf sein. Balisen können über eine eigene Stromversorgung verfügen oder per Induktion über die Zugempfangstechnik aktiviert werden (*property* `isActive`). Im Rahmen der ETCS-Initiative [29] werden sogenannte Eurobalisen (*property* `hasBaliseType` = 'euroBalise') unter anderem dazu eingesetzt, die Belegt- oder Nicht-Belegtheit von Blockabschnitten an nachfolgende Züge zu kommunizieren. Die Balisentechnik ist

zwar schon ausgereift, aufgrund von langen Innovationszyklen im Eisenbahnsektor und der schrittweisen Einführung von ETCS ist jedoch erst eine geringe Anzahl von Balisen im deutschen Schienennetz installiert.

In Listing 4.23 werden die Klasse `Balise` sowie die entsprechenden *properties* dargestellt.

```
1  Declaration(Class(:Balise))
2  SubClassOf(:Balise :ZeroDimObject)
3
4  Declaration(DataProperty(:isActive))
5  FunctionalDataProperty(:isActive)
6  DataPropertyDomain(:isActive :Balise)
7  DataPropertyRange(:isActive xsd:boolean)
8
9  Declaration(ObjectProperty(:hasBaliseType))
10 FunctionalObjectProperty(:hasBaliseType)
11 ObjectPropertyDomain(:hasBaliseType :Balise)
12 ObjectPropertyRange(:hasBaliseType :BaliseType)
13 Declaration(NamedIndividual(:euroBalise))
14 ClassAssertion(:BaliseType :euroBalise)
15 Declaration(NamedIndividual(:staticBalise))
16 ClassAssertion(:BaliseType :staticBalise)
17 Declaration(NamedIndividual(:variableBalise))
18 ClassAssertion(:BaliseType :variableBalise)
```

Listing 4.23: Deklaration der Klasse `Balise` sowie der entsprechenden *properties*

4.3.4 Betriebselemente

4.3.4.1 Grundlegendes

Neben den mehrdimensionalen Elementen und den punktförmigen Elementen, die in den letzten Kapiteln beschrieben wurden, existiert noch die Gruppe der Betriebselemente. Dies sind abstrakte Konzepte aus Stellwerkssicht, die für einen technisch gesicherten Ablauf des Zugverkehrs notwendig sind und zentral überwacht bzw. gesteuert werden. Neben den Stellwerken selbst und den zugehörigen Bedienzentralen, die geographisch unabhängig von der entsprechenden Zugstrecke sind, zählen weitere Elemente zu den Betriebselementen. Diese sind auf dem konkreten Gleisnetz positioniert und bilden eine weitere Abstraktionsschicht über der Schicht der Weichen und Gleisabschnitte und der ihr zugrunde liegenden Schicht des Basisgraphen. Diese gleisnetzbezogenen Betriebselemente umfassen Blockabschnitte, Fahrstraßen, Durchrutschwege und Flankenschutzelemente.

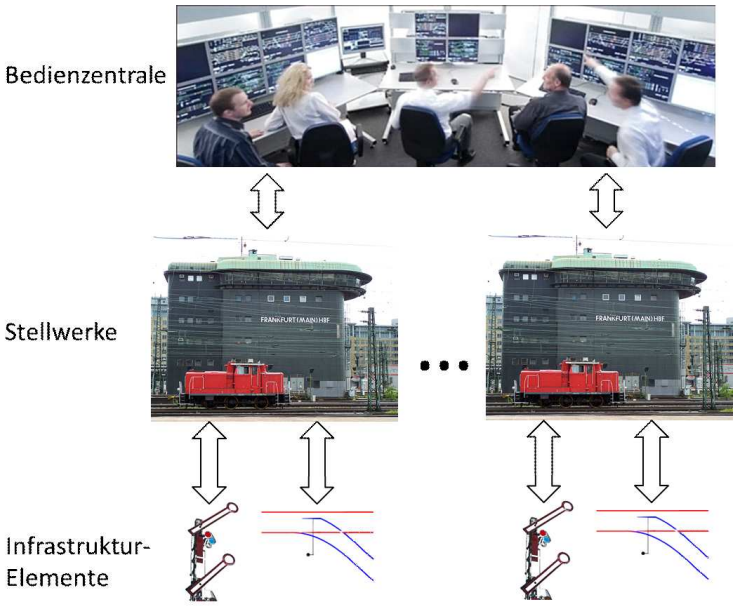

Abbildung 4.7: Zusammenhang zwischen Bedienzentrale, Stellwerken und Infrastrukturelementen

4.3.4.2 Bedienzentrale

Eine Bedienzentrale ist das Element einer Zugstrecke, welches die Zugbewegungen überwacht und freigibt. Die Bedienzentrale ist ein von der Zugstrecke geographisch unabhängiger Ort, an dem mit technischer Unterstützung Überwachungs- und Steuerungstätigkeiten ausgeführt werden. Ein oder mehrere Stellwerke werden von einer Bedienzentrale gesteuert. Diese sind direkt mit den Infrastrukturelementen wie beispielsweise Weichen und Signalen verbunden und übermitteln Steuerbefehle an diese. Außerdem werden Statusmeldungen der Infrastrukturelemente zurück an die Bedienzentrale übermittelt (vgl. Abbildung 4.7).

Die Bedienzentrale ist die Schnittstelle des Personals zur Überwachung und Steuerung der Eisenbahn-Infrastrukturen. Die eigentliche Steuerungslogik befindet sich in den einzelnen Stellwerken. Hier werden die Voraussetzungen geprüft, ob beispielsweise eine Fahrstraße eingestellt werden kann. Der Bedienzentrale werden diese Prozesse übermittelt und dem Fahrdienstleiter präsentiert.

Aus Modellierungssicht ist eine Bedienzentrale (Klasse: `ControlCenter`) kein direktes Element der Eisenbahn-Infrastruktur sondern ihr übergeordnet. Die Klasse ist eine direkte Unterklasse der Klasse `Interlocking`, in der alle Bedienelemente zusammengefasst sind. Die Bedienzentrale ist nicht direkt mit Infrastrukturelemen-

ten verknüpft, sondern lediglich einem oder mehreren Stellwerken über die *property* `hasInterlockingCenter` zugeordnet (vgl. Listing 4.24).

```
1  Declaration(Class(:Interlocking))
2  SubClassOf(:Interlocking :BaseIdName)
3
4  Declaration(Class(:ControlCenter))
5  SubClassOf(:ControlCenter :Interlocking)
6
7  Declaration(ObjectProperty(:hasInterlockingCenter))
8  InverseObjectProperties(:hasInterlockingCenter
       :isInterlockingCenterOfControlCenter)
9  ObjectPropertyDomain(:hasInterlockingCenter :ControlCenter)
10 ObjectPropertyRange(:hasInterlockingCenter :InterlockingCenter)
```

Listing 4.24: Deklaration der Klasse `ControlCenter` sowie der *property* `hasInterlockingCenter` zur Zuordnung von Stellwerken

4.3.4.3 Stellwerk

Das Stellwerk ist die zentrale Einheit eines Streckenabschnitts, in dem sämtliche Leit- und Sicherheitstechnik kontrolliert wird. Mithilfe eines Stellwerks soll ein sicherer Betrieb der Eisenbahn-Infrastruktur gewährleistet werden.

Es existieren mehrere Typen von Stellwerken. Bei der Neuplanung von Zugstrecken werden nur noch elektronische Stellwerke eingesetzt. Diese können auch über eine entfernte Bedienzentrale gesteuert werden und sind physisch betrachtet meist nur noch ein Schaltkasten am Streckenrand. Häufig ist jedoch auch der Fall, dass Strecken teilmodernisiert werden und somit ältere Technik mit neuer kombiniert werden muss. Aus diesem Grund ist es nötig, bei der Modellierung der Stellwerksklasse (`InterlockingCenter`) auch die älteren Stellwerkstypen zu erfassen. Dies wird über die *property* `hasInterlockingType` gewährleistet. Neben der Typenzuordnung ist es wichtig, dass die einzelnen Streckenelemente einem Stellwerk zugeordnet werden können. Dies wird über die *property* `controls` ermöglicht, welche als *range*-Angabe die Klasse `BlockSection` referenziert (siehe nächster Abschnitt 4.3.4.4). Über eine `BlockSection` können alle aus Stellwerkssicht sicherheitsrelevanten Elemente ausgelesen und angesprochen werden.

```
1  Declaration(Class(:InterlockingCenter))
2  SubClassOf(:InterlockingCenter :Interlocking)
3
4  Declaration(ObjectProperty(:hasInterlockingType))
5  FunctionalObjectProperty(:hasInterlockingType)
6  ObjectPropertyDomain(:hasInterlockingType :InterlockingCenter)
7  ObjectPropertyRange(:hasInterlockingType :InterlockingCenterType)
8
9  Declaration(Class(:InterlockingCenterType))
10 SubClassOf(:InterlockingCenterType ObjectIntersectionOf(ObjectOneOf(
       :electronical :electro-mechanical :electrical :mechanical) :Enums))
11
12 Declaration(NamedIndividual(:electrical))
13 ClassAssertion(:InterlockingCenterType :electrical)
14 Declaration(NamedIndividual(:electro-mechanical))
15 ClassAssertion(:InterlockingCenterType :electro-mechanical)
16 Declaration(NamedIndividual(:electronical))
17 ClassAssertion(:InterlockingCenterType :electronical)
18 Declaration(NamedIndividual(:mechanical))
19 ClassAssertion(:InterlockingCenterType :mechanical)
20
21 Declaration(ObjectProperty(:controls))
22 InverseObjectProperties(:controlledByInterlockingCenter :controls)
23 ObjectPropertyDomain(:controls :InterlockingCenter)
24 ObjectPropertyRange(:controls :BlockSection)
```

Listing 4.25: Deklaration der Klasse `InterlockingCenter` sowie der zugeordneten *properties*

4.3.4.4 Blockabschnitt

Blockabschnitte (Klasse: `BlockSection`) sind Teilbereiche von Zugstrecken, die sicherungstechnisch darauf ausgelegt sind, jeweils nur einem Zug zu ermöglichen, sie zu befahren. Blockabschnitte können sowohl Weichen als auch Gleisabschnitte umfassen. Sie liegen als eine abstrakte Schicht über diesen Elementen. Die spezifische Zuordnung zu einem Gleisabschnitt oder einer Weiche erfolgt über die *property* `relatesTo`, welche als *range*-Angabe ein `TwoDimObject` erhält (vgl. Kapitel 4.3.2). Dieses `TwoDimObject` vereint alle Elemente, auf denen Zugbewegungen stattfinden, und die somit aus Stellwerkssicht als Blockabschnitte kontrolliert werden können.

Da ein Blockabschnitt mehrere `TwoDimObjects` umfassen kann, ist die `relatesTo` *property* auch nicht als `functional` deklariert. Im Gegenzug dazu ist jedoch die inverse *property* `isRelatedTo` als `InverseFunctionalProperty` deklariert, da jedes `TwoDimObject` nur jeweils genau einem Blockabschnitt zugeordnet werden kann.

Ein Blockabschnitt kann auch nur einen Teil eines `TwoDimObjects` umfassen, somit sind Kilometrierungsangaben erforderlich. Hierzu wurden die bestehenden *properties* `begin` und `end` zur Kilometrierungsangabe von Gleisabschnitten und Weichen um die *domain* `BlockSection` erweitert. Des Weiteren wurde die *data property* `branch` eingeführt, um eine Kilometrierungsangabe auf dem abzweigenden Gleis einer Weiche zu ermöglichen.

Über die *property* `isPartOfRoute` wird die Zuordnung eines Blockabschnitts zu einer Fahrstraße bzw. einem Durchrutschweg hergestellt. Da ein Blockabschnitt auch Teil verschiedener Fahrstraßen oder Durchrutschwege sein kann, ist diese *property* nicht als funktional deklariert und beinhaltet als *domain*-Deklaration beide Klassen. Die inverse *property* zu `isPartOfRoute` ist `routeContainsBlockSection`. Diese *property* sowie weitere Zusammenhänge werden in den entsprechenden Kapiteln (Fahrstraße: 4.3.4.5, Durchrutschweg: 4.3.4.6) erläutert.

Grundsätzlich wird zwischen nicht-selbsttätigen Blockabschnitten (*property* `hasBlockSectionType` = 'manualBlockSection') und selbsttätigen Blockabschnitten (*property* `hasBlockSectionType` = 'automaticBlockSection') unterschieden. Bei den nicht-selbsttätigen Blockabschnitten ist immer ein Mensch dafür zuständig, einen Blockabschnitt bei Zugeinfahrt zu sichern. Dies geschieht in der Regel durch das Umstellen eines Signals. Bei selbsttätigen Blockabschnitten geschieht dies maschinell. Diese Abschnitte werden in der Regel durch technische Zugdetektionseinrichtungen wie beispielsweise Achszähler begrenzt. Dadurch kann einem Stellwerk automatisch rückgemeldet werden, ob sich ein Zug in einem Blockabschnitt befindet oder ob dieser bereits (vollständig) herausgefahren ist.

```
1  Declaration(Class(:BlockSection))
2  SubClassOf(:BlockSection :Interlocking)
3
4  Declaration(ObjectProperty(:controlledByInterlockingCenter))
5  InverseObjectProperties(:controlledByInterlockingCenter :controls)
6  FunctionalObjectProperty(:controlledByInterlockingCenter)
7  ObjectPropertyDomain(:controlledByInterlockingCenter :BlockSection)
8  ObjectPropertyRange(:controlledByInterlockingCenter :InterlockingCenter)
9
10 Declaration(ObjectProperty(:hasBlockSectionType))
11 FunctionalObjectProperty(:hasBlockSectionType)
12 ObjectPropertyDomain(:hasBlockSectionType :BlockSection)
13 ObjectPropertyRange(:hasBlockSectionType :BlockSectionType)
14 Declaration(NamedIndividual(:automaticBlockSection))
15 ClassAssertion(:BlockSectionType :automaticBlockSection)
16 Declaration(NamedIndividual(:manualBlockSection))
17 ClassAssertion(:BlockSectionType :manualBlockSection)
18
19 Declaration(ObjectProperty(:relatesTo))
20 ObjectPropertyDomain(:relatesTo :BlockSection)
```

```
21 ObjectPropertyRange(:relatesTo :TwoDimObject)
22
23 Declaration(ObjectProperty(:isPartOfRoute))
24 InverseObjectProperties(:isPartOfRoute :routeContainsBlockSection)
25 ObjectPropertyDomain(:isPartOfRoute :BlockSection)
26 ObjectPropertyRange(:isPartOfRoute :Overlap)
27 ObjectPropertyRange(:isPartOfRoute :Route)
28
29 DataPropertyDomain(:begin :BlockSection)
30 DataPropertyDomain(:end :BlockSection)
31
32 Declaration(DataProperty(:branch))
33 FunctionalDataProperty(:branch)
34 DataPropertyDomain(:branch :BlockSection)
35 DataPropertyRange(:branch xsd:long)
```

Listing 4.26: Deklaration der Klasse `BlockSection` sowie der zugeordneten *properties*

4.3.4.5 Fahrstraße

Eine Fahrstraße (Klasse: `Route`) ist eine Menge von zusammenhängenden Blockabschnitten, welche durch Hauptsignale begrenzt ist. Fahrstraßen sind - ebenso wie Blockabschnitte - abstrakte sicherheitstechnische Elemente aus Stellwerkssicht und werden für eine zeitlich exklusive Zugnutzung im Stellwerk über die Bedienzentrale vom Fahrdienstleiter reserviert. Die Zuordnung zu den einzelnen Blockabschnitten erfolgt über die *property* `routeContainsBlockSection`. Eine Fahrstraße ist über Flankenschutzelemente dagegen gesichert, dass andere Züge in sie einfahren können (vgl. Kapitel 'Flankenschutzelemente' 4.3.4.7). Die Zuordnung zu den Flankenschutzelementen erfolgt über die *property* `isProtectedBy`. Die Fahrstraße begrenzende Signale werden über die entsprechenden *properties* `routesStartingSignal` und `routesEndingSignal` angegeben.

```
1  Declaration(Class(:Route))
2  SubClassOf(:Route :Interlocking)
3
4  Declaration(ObjectProperty(:routeContainsBlockSection))
5  InverseObjectProperties(:isPartOfRoute :routeContainsBlockSection)
6  ObjectPropertyDomain(:routeContainsBlockSection :Route)
7  ObjectPropertyRange(:routeContainsBlockSection :BlockSection)
8
9  Declaration(ObjectProperty(:isProtectedBy))
10 InverseObjectProperties(:protects :isProtectedBy)
11 ObjectPropertyDomain(:isProtectedBy :Route)
12 ObjectPropertyRange(:isProtectedBy :FlankProtectionElement)
13
```

```
14 Declaration(ObjectProperty(:routesEndingSignal))
15 InverseObjectProperties(:routesEndingSignal :isEndingSignalOfRoute)
16 FunctionalObjectProperty(:routesEndingSignal)
17 ObjectPropertyDomain(:routesEndingSignal :Route)
18 ObjectPropertyRange(:routesEndingSignal :Signal)
19
20 Declaration(ObjectProperty(:routesStartingSignal))
21 InverseObjectProperties(:routesStartingSignal :isStartingSignalOfRoute)
22 FunctionalObjectProperty(:routesStartingSignal)
23 ObjectPropertyDomain(:routesStartingSignal :Route)
24 ObjectPropertyRange(:routesStartingSignal :Signal)
```

Listing 4.27: Deklaration der Klasse `Route` sowie der zugeordneten *properties*

Eine Fahrstraße kann an ihrem Ende einen Durchrutschweg angehängt haben (vgl. Kapitel 'Durchrutschweg' 4.3.4.6), der als Sicherheitszone für einen nicht rechtzeitig bremsenden Zug fungiert. Nicht jede Fahrstraße besitzt einen Durchrutschweg. Ist dies jedoch der Fall, so wird eine solche Fahrstraße über die Unterklasse `ComposedRoute` von `Route` modelliert. Diese Klasse beinhaltet eine funktionale *object property* `routeHasOverlap`, über die der Bezug zwischen Fahrstraße und zugehörigem Durchrutschweg hergestellt wird.

```
1 Declaration(Class(:ComposedRoute))
2 EquivalentClasses(:ComposedRoute ObjectIntersectionOf(
      ObjectExactCardinality(1 :routeHasOverlap :Overlap) :Route))
3 SubClassOf(:ComposedRoute :Route)
4
5 Declaration(ObjectProperty(:routeHasOverlap))
6 InverseObjectProperties(:relatesToRoute :routeHasOverlap)
7 FunctionalObjectProperty(:routeHasOverlap)
8 ObjectPropertyDomain(:routeHasOverlap :ComposedRoute)
9 ObjectPropertyRange(:routeHasOverlap :Overlap)
```

Listing 4.28: Deklaration der Klasse `ComposedRoute` sowie der zugeordneten *properties*

4.3.4.6 Durchrutschweg

Ein Durchrutschweg (Klasse: `Overlap`) ist Bestandteil einer Fahrstraße (Klasse: `ComposedRoute`). Er ist ein Blockabschnitt am Ende des regulären Fahrwegs einer Fahrstraße und ist gewissermaßen die Auslaufstrecke für den Fall, dass ein Zug nicht rechtzeitig am Ende des eigentlichen Fahrweges zum Stehen kommt. Durchrutschwege müssen - ebenso wie Fahrstraßen - durch Flankenschutzelemente (vgl. Kapitel 4.3.4.7) abgesichert werden. Die Zuordnung zu Flankenschutzelementen erfolgt über eine ent-

sprechende *domain*-Angabe der *property* `isProtectedBy` (siehe Kapitel 'Flankenschutzelemente' 4.3.4.7). Durchrutschwege beanspruchen einen Blockabschnitt entweder gänzlich oder teilweise. Deswegen ist eine Längenangabe des Durchrutschweges über die funktionale *data property* `length` notwendig. Die Zuordnung eines Durchrutschweges zu einem Blockabschnitt erfolgt über eine entsprechende *domain*-Angabe der *property* `routeContainsBlockSection` (siehe Kapitel 'Fahrstraße' 4.3.4.5). Des Weiteren kann ein Durchrutschweg Bestandteil von mehreren Fahrstraßen sein. Dies wird über die nicht-funktionale *property* `relatesToRoute` modelliert.

```
1  Declaration(Class(:Overlap))
2  SubClassOf(:Overlap :Interlocking)
3
4  ObjectPropertyDomain(:routeContainsBlockSection :Overlap)
5
6  ObjectPropertyDomain(:isProtectedBy :Overlap)
7
8  Declaration(ObjectProperty(:relatesToRoute))
9  InverseObjectProperties(:relatesToRoute :routeHasOverlap)
10 ObjectPropertyDomain(:relatesToRoute :Overlap)
11 ObjectPropertyRange(:relatesToRoute :ComposedRoute)
12
13 Declaration(DataProperty(:length))
14 FunctionalDataProperty(:length)
15 DataPropertyDomain(:length :Overlap)
16 DataPropertyRange(:length xsd:int)
```

Listing 4.29: Deklaration der Klasse `Overlap` sowie der zugeordneten *properties*

4.3.4.7 Flankenschutzelemente

Flankenschutzelemente haben die Aufgabe, eine reservierte Fahrstraße und / oder einen reservierten Durchrutschweg so abzusichern, dass keine weiteren Züge hineinfahren können. Flankenschutzelemente können Weichen oder Signale sein.

Die Abbildung 4.8 zeigt eine Fahrstraße mit zugehörigem Durchrutschweg (Gelb markiert). Flankenschutz bieten hierbei die beiden Weichen, die in den Fahrweg münden. Dabei ist die Stellung der Weichen für einen korrekten Flankenschutz rot gekennzeichnet.

Aus Modellierungssicht ist ein Flankenschutzelement durch die Klasse `FlankProtectionElement` definiert. Diese ist eine Unterklasse von `Interlocking` und bildet wiederum selbst eine Oberklasse von `Signal` (vgl. Kapitel 4.3.3.1) und `Switch` (vgl. Kapitel 4.3.2.2), die, wie beschrieben, als Flankenschutzelemente agieren können. Werden Signale als Flankenschutzelemente eingesetzt, zeigen sie immer den 'Halt'-Begriff und

Abbildung 4.8: Flankenschutz einer Fahrstraße

werden somit als nicht zwingende Flankenschutzelemente bezeichnet, da ein Zug bei Nichtbeachtung des Signals trotzdem in die zu schützende Fahrstraße einfahren könnte. Weichen hingegen werden als zwingende Flankenschutzelemente bezeichnet, da sie immer in einer zur schützenden Fahrstraße abweisenden Richtung geschaltet sind und somit die Fremdeinfahrt eines Zuges unmöglich machen. Diese Tatsache hat aus Modellierungssicht zur Folge, dass zwischen Signalen und Weichen als Flankenschutzelemente unterschieden werden muss, da Weichen eine entsprechende Schaltung bezogen auf die jeweilige Fahrstraße aufweisen müssen. Diese Modellierungsdetails sind in Listing 4.30 dargestellt und werden im Folgenden erläutert: In den Zeilen 4 - 7 wird für eine Weiche eine neue *object property* `protectiveState` eingeführt, die eine Weichenlage beschreiben kann (`straight` oder `branch`). Von der in Zeile 16 deklarierten *super object property* `protects` erben jeweils die beiden spezialisierten *sub object properties* `flankingSignalProtects` und `flankingSwitchProtects`. Als *domain* haben diese beiden *properties* eine Instanz von `Signal` bzw. `Switch`. Als *range* können sowohl eine `Route` als auch ein `Overlap` als zu schützende Elemente angegeben werden. Die Besonderheit bei der *property* `flankingSwitchProtects` besteht nun darin, dass sie durch eine *property chain* definiert wird, die einer entsprechende Weiche in einer bestimmten Lage einer zu schützenden Fahrstraße oder eines Durchrutschweges zugeordnet wird. Diese *property chain* ist in der Zeile 33 des Listings aufgeführt.

```
1  Declaration(Class(:FlankProtectionElement))
2  SubClassOf(:FlankProtectionElement :Interlocking)
3
4  Declaration(ObjectProperty(:protectiveState))
5  FunctionalObjectProperty(:protectiveState)
6  ObjectPropertyDomain(:protectiveState :Switch)
7  ObjectPropertyRange(:protectiveState :SwitchBranchState)
8
9  Declaration(Class(:SwitchBranchState))
10 SubClassOf(:SwitchBranchState ObjectIntersectionOf(ObjectOneOf(:straight
       :branching) :Enums))
11 Declaration(NamedIndividual(:branching))
12 ClassAssertion(:SwitchBranchState :branching)
13 Declaration(NamedIndividual(:straight))
14 ClassAssertion(:SwitchBranchState :straight)
```

```
15
16 Declaration(ObjectProperty(:protects))
17 ObjectPropertyDomain(:protects :FlankProtectionElement)
18 ObjectPropertyRange(:protects :Overlap)
19 ObjectPropertyRange(:protects :Route)
20
21 Declaration(ObjectProperty(:flankingSignalProtects))
22 SubObjectPropertyOf(:flankingSignalProtects :protects)
23 ObjectPropertyDomain(:flankingSignalProtects :Signal)
24 ObjectPropertyRange(:flankingSignalProtects :Overlap)
25 ObjectPropertyRange(:flankingSignalProtects :Route)
26
27 Declaration(ObjectProperty(:flankingSwitchProtects))
28 SubObjectPropertyOf(:flankingSwitchProtects :protects)
29 ObjectPropertyDomain(:flankingSwitchProtects :Switch)
30 ObjectPropertyRange(:flankingSwitchProtects :Overlap)
31 ObjectPropertyRange(:flankingSwitchProtects :Route)
32
33 SubObjectPropertyOf(ObjectPropertyChain(:protects :protectiveState)
       :flankingSwitchProtects)
```

Listing 4.30: Deklaration der Klasse `FlankProtectionElement` sowie der zugeordneten *properties*

4.4 RI* Rule Ontology

4.4.1 Grundlegendes

Im vorigen Kapitel wurde die *RI* Core Ontology* beschrieben, die ein Modell der Eisenbahndomäne, ihrer Elemente und deren Zusammenhänge darstellt. In diesem Kapitel hingegen wird die Ontologie beschrieben, die sämtliche Regeln zur Verifikation von EI enthält. Die Notwendigkeit dieser Ontologie wird in den folgenden Absätzen begründet.

Angenommen es existiert eine konkrete *Specific RI Ontology* (vgl. Kapitel 4.5), die die Klassen der *RI* Core Ontology* nutzt, um von ihnen konkrete Individuen zu erstellen, die einer spezifischen Planung einer Eisenbahn-Infrastruktur entsprechen. Wenn Individuen dieser Ontologie nicht konform mit den modellierten Einschränkungen in der *RI* Core Ontology* erzeugt worden sind, wird durch einen Verarbeitungsprozess mit dem Reasoner ermittelt, dass sich die gesamte Wissensbasis in einem inkonsistenten Zustand befindet.

Wenn also beispielsweise ein Individuum der Klasse `OrdinarySwitch` in der *Specific RI Ontology* erzeugt wurde, so muss es allen Anforderungen der Klasse entsprechen. Per Definition (vgl. Listing 4.31) muss eine `OrdinarySwitch` über genau drei Verbindungen verfügen. Verfügt das Individuum über mehr als drei Verbindungen, so wird durch die Anwendung des Reasoners ein Fehler erkannt und die gesamte Wissensbasis als inkonsistent klassifiziert. Verfügt das Individuum jedoch nur über die Zuordnung von beispielsweise zwei Verbindungen, so kann durch die Verwendung des Reasoners kein Fehler erkannt werden, da im Sinne der OWA eine noch nicht explizit definierte Verbindung vorliegen könnte. Zu Verifikationszwecken ist es jedoch zwingend erforderlich, dass solch ein Fehler erkannt wird. Hierzu kann SWRL genutzt werden. Zwar unterliegt auch SWRL der OWA, jedoch verwenden die Regeln nur benannte Individuen, um entscheidbar zu bleiben. Diese Tatsache kann dazu genutzt werden, fehlende *property*-Zuordnungen zu erkennen.

```
1 ...
2 EquivalentClasses(:OrdinarySwitch ObjectIntersectionOf(
    ObjectExactCardinality(3 :vertexHasPort :Port) :Vertex))
3 ...
```

Listing 4.31: Auszug aus der Klassendefinition `OrdinarySwitch`

Neben der Detektion von fehlenden *property*-Zuordnungen existieren in der *RI* Rule Ontology* noch weitere Arten von SWRL-Regeln für Verifikationszwecke. Grund-

sätzlich können fünf Kategorien unterschieden werden, wobei manche Regeln auch mehreren Kategorien zugeordnet werden können:

- **Verifikationsregeln** - Hierbei handelt es sich, wie im Beispiel im vorigen Abschnitt dargestellt, um Regeln, die fehlerhaft attributierte Individuen aufzeigen sollen und so die OWA umgehen, um Daten-Verifikation zu ermöglichen. Hierzu ist jedoch eine externe Software nötig, die einen Vergleich zwischen den Individuen der Originalklasse (z.B.: `OrdinarySwitch`) und den Individuen der Verifikationsklasse (z.B.: `CorrectlyConnectedOrdinarySwitch`) vollzieht.

- ***property*-Regeln** - *property*-Regeln haben die Besonderheit, dass sie die Erfassung von *property*-Zuweisungen ermöglichen, die mit Standard OWL nicht erfassbar wären.

- **Negativ-Regeln** - Diese Regeln beschreiben häufige Modellierungs- und Planungsfehler. Individuen, die die Bedingung einer solchen Regel erfüllen, werden entsprechenden Fehlerklassen zugeordnet.

- **Vergleichs-, Arithmetische- und Zeichenketten-Regeln** - Diese Regeln sind nötig, um mathematische Operationen und Literalverarbeitungen zu ermöglichen. Diese Aktionen sind in OWL nicht möglich, weswegen hierzu auf SWRL zurückgegriffen werden muss. So können beispielsweise Individuen verglichen, Distanzen von EI-Elementen zueinander berechnet und Literaloperationen betrieben werden.

- **Formaler Planungsrichtlinien** - Die Planung von EI unterliegt strengen Richtlinien, die in einem Kompendium in natürlicher Sprache hinterlegt sind. Eine Formalisierung dieser Planungsrichtlinien ist für eine modellbasierte automatische Verifikation erforderlich.

Die Kategorien der in der *RI* Rule Ontology* enthaltenen Regeln werden in den folgenden Kapitel erläutert.

4.4.2 Verifikationsregeln

Verifikationsregeln sind Regeln, die zur Formulierung grundlegender Planungsbedingungen verwendet werden. Sie wurden für den Zweck der Verifikation eingeführt, sind aber nicht grundsätzlich mit formalen Regeln der Planungsrichtlinien der Deutschen Bahn gleichzusetzen. Verifikationsregeln sind durch verschiedene Kriterien als solche charakterisiert:

- Verifikationsregeln sind spezielle Regeln, die ein bis n Atome in der Regelantezedenz aufweisen, jedoch nur genau ein Atom in der Regelkonsequenz.

- Mindestens ein Atom in der Antezedenz einer Verifikationsregel beinhaltet eine Klasse der *RI* Core Ontology*. Diese Klasse wird in Bezug auf diese Regel als Kandidatenklasse bezeichnet. Gemeinhin werden weitere *properties* dieser Kandidatenklasse in der Verifikationsregel als Atome verwendet. Die Bedeutung der Verifikationsregel liegt darin, diese *properties* zu analysieren und dahingehend die entsprechenden Individuen der Kandidatenklasse zu klassifizieren.

- Die Konsequenz einer Verifikationsregel beinhaltet grundsätzlich nur ein Atom, bestehend aus einer Verifikationsklasse. Diese Verifikationsklassen sind in der *RI* Rule Ontology* definiert und dienen lediglich der Unterstützung des Verifikationsprozesses. Sie sind also nicht Bestandteil der EI-Domäne. Sie werden als Sammelpunkt für alle Individuen einer Kandidatenklasse verwendet, die der Bedingung der konkreten Verifikationsregel entsprechen.

- Die Bezeichnung der Verifikationsklassen folgt einem bestimmten Schema in englischer Sprache. Die Bezeichnung beginnt mit dem Wort 'Correct' oder 'Correctly', gefolgt von einem Verb. Der letzte Teil der Bezeichnung von Verifikationsklassen beinhaltet immer die Bezeichnung der Kandidatenklasse, auf die sich diese Verifikationsregel bezieht wie beispielsweise 'CorrectlyPlacedSignal'.

- Die Individuen der Kandidatenklasse werden als 'korrekt' in Bezug auf die Verifikationsregel erachtet, wenn sie nach Anwendung der Regel ebenso der entsprechenden Verifikationsklasse zugeordnet sind. Sie haben also der Bedingung der Regel, formuliert in der Regelantezedenz, entsprochen.

Beispiele für Verifikationsregeln sind zum einen das Listing 4.36 aus dem vorherigen Kapitel, oder aber auch die Regeln zur Prüfung, ob eine Fahrstraße an einem Hauptsignal beginnt und endet (vgl. Listing 4.32.

```
1 Route(?r), Signal(?s), hasSignalType(?s, Main),
2 routesStartingSignal(?r, ?s)
3 -> CorrectlySignalledRouteStart(?r)
4
5 Route(?r), Signal(?s), hasSignalType(?s, Main),
6 routesEndingSignal(?r, ?s)
7 -> CorrectlySignalledRouteEnd(?r)
```

Listing 4.32: Verifikationsregeln zur Überprüfung einer korrekten Signalisierung von Fahrstraßen

Hierbei wurde eine weitere Modellierungsbesonderheit von OWL genutzt: Neben den beiden Verifikationsklassen `CorrectlySignalledRouteStart` und `CorrectlySignalled-RouteEnd` existiert noch eine Verifikationsklasse `CorrectlySignalledRoute`, die die beiden Klassen entsprechend dem Ausdruck in Listing 4.33 vereinigt und hierzu eine Äquivalenzklasse bildet.

```
1 Declaration(Class(:CorrectlySignalledRouteEnd))
2 SubClassOf(:CorrectlySignalledRouteEnd :VerificationClass)
3 Declaration(Class(:CorrectlySignalledRouteStart))
4 SubClassOf(:CorrectlySignalledRouteStart :VerificationClass)
5
6 Declaration(Class(:CorrectlySignalledRoute))
7 EquivalentClasses(:CorrectlySignalledRoute ObjectIntersectionOf(
    :CorrectlySignalledRouteStart :CorrectlySignalledRouteEnd))
8 SubClassOf(:CorrectlySignalledRoute :VerificationClass)
```

Listing 4.33: Verifikationsklasse `CorrectlySignalledRoute` als vereinigte Klasse aus `CorrectlySignalledRouteStart` und `CorrectlySignalledRouteEnd`

In den jeweiligen Verifikationsklassen der *RI* Rule Ontology* werden alle als korrekt klassifizierten Individuen einer bestimmten Klasse zusammengefasst. So sind nach einem Verifikationsprozess in der Klasse `CorrectlySignalledRoute` all diejenigen `Route`-Individuen zu finden, die sowohl über eine korrekte Zuordnung zu einem Startsignal als auch einem Endsignal verfügen. Per Software wird nun abgeglichen, ob alle Routen in der Klasse `CorrectlySignalledRoute` enthalten sind. Ist dies nicht der Fall, so werden die nicht enthaltenen Routen als fehlerhaft klassifiziert.

Da die entsprechenden Verifikationsklassen ebenso in den Antezedenzen anderer Regeln verwendet werden können, ist es somit möglich, eine hierarchische Struktur zu schaffen. So können beispielsweise Regeln, welche Verifikationsklassen als Atome enthalten, nur dann erfüllt werden, wenn alle Individuen in diesen Atomen bereits verifiziert sind. Im Beispiel aus Listing 4.32 können die beiden Verifikationsklassen `CorrectlySignalledRouteStart` und `CorrectlySignalledRouteEnd` in einer Regel verwendet werden, die die korrekte Länge eines Durchrutschweges der entsprechenden Route berechnet. Durch diese Vorgehensweise wird gewährleistet, das nur solche Routen getestet werden, die bereits über korrekte Zuordnungen zu Start- und Endsignalen verfügen.

4.4.3 *property*-Regeln

Property-Regeln ermöglichen *property*-Zuweisungen, die mit Standard OWL schwierig bis unmöglich sind. Dies ist beispielsweise der Fall, wenn die Zuweisung einer

property durch den Zusammenhang der Belegung anderer, voneinander unabhängiger *properties* abhängt und dieser Zusammenhang nicht durch eine *property chain* ausgedrückt werden kann.

property-Regeln sind dadurch gekennzeichnet, dass sie eine beliebige Anzahl von Atomen in der Antezedenz der Regel beinhalten, aber grundsätzlich nur ein Atom in der Konsequenz der Regel. Diese Atom besteht immer aus einer *property*.

Ein Beispiel hierfür ist die *object property* `directConnection`. Sie hängt zum einen von der *property chain* `vertexConnectVertex` ab, die in der *RI* Graph Ontology* definiert ist, zum anderen aber auch von den Kilometrierungsangaben der beiden Gleisabschnitte, die miteinander verbunden zu sein scheinen. Die entsprechende SWRL-Regel hierzu ist in Listing 4.34 dargestellt.

```
1 Track(?t1), Track(?t2), DifferentFrom(?t1, ?t2),
2 vertexConnectVertex(?t1, ?t2), begin(?t2, ?beg), end(?t1, ?end),
3 equal(?beg, ?end)
4 -> directConnection(?t1, ?t2)
```

Listing 4.34: *property*-Regel `directConnection`, die eine *property*-Zuweisung von direkt verknüpften Gleisabschnitten vornimmt

4.4.4 Negativ-Regeln

Eine weitere Kategorie von SWRL-Regeln, die in der *RI* Rule Ontology* Verwendung finden, ist die Kategorie der Negativ-Regeln. Regeln in dieser Kategorie beschreiben gängige Modellierungsfehler. Die Regeln sind dadurch charakterisiert, dass in der Regel-Konsequenz eine Verifikationsklasse (vgl. Kapitel 4.4.2) steht, in der die fehlerhaften Individuen gesammelt werden. In der Regel werden diese sogenannten Fehlerklassen mit dem Schlüsselwort 'Incorrect' eingeleitet. Ein Beispiel für eine Negativ-Regel bezieht sich auf die Ausrichtung von Elementen mit Wirkrichtung (Klasse `DirectedAttachment` in Kapitel 4.2) wie beispielsweise Signale. Ist ein Gleisabschnitt nur in eine bestimmte Richtung befahrbar, ist ein falsch ausgerichtetes Signal ein offensichtlicher Modellierungsfehler und sollte als solcher erkannt werden. Falsch ausgerichtet bedeutet in diesem Sinne, dass ein Signal dieselbe Richtung aufweist, wie die mögliche Fahrtrichtung auf dem Gleis. Um dieses Szenario zu erfassen, wird die Regel des Listings 4.35 verwendet. Sie besagt, dass jedes `TwoDimObject`, also jedes Gleisobjekt, welches über eine Verknüpfung zu einem `DirectedAttachment` (beispielsweise einem Signal) verfügt und nicht über beide Richtungen befahrbar ist (`DifferentFrom(?dir_tdo, both)` und dennoch eine Befahrbarkeitsrichtung aufweist, die gleich der Wirkrichtung des entsprechenden `DirectedAttachment` ist, als falsch mo-

delliert zu klassifizieren ist.

```
1 TwoDimObject(?tdo), DirectedAttachment(?da),
2 baseModelHasAttachment(?tdo, ?da),
3 hasTrafficDirection(?tdo, ?dir_tdo),
4 directedAttachmentHasDirection(?da, ?dir_da),
5 DifferentFrom (?dir_tdo, ?dir_da), DifferentFrom (?dir_tdo, both)
6 -> IncorrectlyOrientedDirectedAttachment(?da)
```

Listing 4.35: Negativ-Regel zur Erfassung von Modellierungsfehlern bezüglich der falschen Ausrichtung von `DirectAttachments`

4.4.5 Regeln für Vergleiche sowie arithmetische und Zeichenketten-Operationen

Wie bereits in Kapitel 2.3.4 beschrieben, zeigt OWL Schwächen bei der Modellierung bzw. Verarbeitung bestimmter Ausdrücke. So ist beispielsweise ein Vergleich auf Individuenebene oder die Angabe von arithmetischen Ausdrücken oder Literaloperation mit OWL nicht möglich. Bei der Modellierung von EI sind jedoch zum Teil numerische Geschwindigkeits- sowie Kilometrierungsangaben und die damit einhergehende Notwendigkeit der Berechnung von Maximalgeschwindigkeiten und Distanzen zwischen Elementen erforderlich. Die auf OWL basierende Regelsprache SWRL beinhaltet dieses Möglichkeiten in Form von Builtins, die als Regelatome verwendet werden können. So ermöglichen mathematische Builtins beispielsweise die Grundrechenarten wie Addition, Subtraktion, Multiplikation und Division von Zahlen sowie Winkelfunktionen, Rundungen, Potenz- und Modulooperationen. Vergleichs-Builtins ermöglichen den Vergleich von Zahlen inklusive Untersuchungen, ob eine Zahl größer oder kleiner als eine andere Zahl ist. Bezüglich Zeichenketten existiert eine Vielzahl von Builtins zur Verkettung von Zeichenketten, zur Bildung von Teilzeichenketten, zu Vergleichen von Zeichenketten und Vielem mehr. Mit diesem Repertoire von Builtins ist eine weitreichende Modellierung von Vergleichen, arithmetischen Handlungsanweisungen und Zeichenkettenoperation möglich.

Ein Beispiel für eine SWRL-Regel, die mathematische sowie Vergleichs-Builtins verwendet, wird in dem Listing 4.36 dargestellt. Es beschreibt eine Regel, mit der sämtliche Signale dahingehend klassifiziert werden können, ob sie sich innerhalb des Intervalls des ihnen zugeordneten Gleisabschnitts befinden.

```
Signal(?s), Track(?t), on(?s, ?t),
position(?s, ?pos), begin(?t, ?beg), end(?t, ?end),
greaterThanOrEqual(?pos, ?beg), lessThanOrEqual(?pos, ?end)
 -> CorrectlyPlacedSignal(?s)
```

Listing 4.36: SWRL-Regel zur Überprüfung der korrekten Position von Signalen in Bezug auf Gleisabschnitte

4.4.6 Formale Planungsrichtlinien

Wie schon in Kapitel 2.1.5 beschrieben, unterliegt der deutsche EI-Planungsprozess Richtlinien, die von der Deutschen Bahn AG aufgestellt wurden. Diese Richtlinien sind in einem Kompendium für den Planungsprozess verbal formuliert. Mit der vorliegenden Arbeit wird eine Möglichkeit dargelegt, diese Richtlinien zu formalisieren. Ein Beispiel einer Richtlinie aus der Ril 819.01 - 'Entwürfe und Pläne' befasst sich beispielsweise mit der Bezeichnung von Gleisabschnitten. So besagt die Richtlinie, dass Gleisbezeichnungen grundsätzlich mit einer Nummer beginnen müssen. Da in SWRL kein Builtin für die Überprüfung von Zeichenketten auf numerische Werte existiert, wird hierbei mit dem Builtin 'startsWith' gearbeitet. Aufgrund dessen, dass in SWRL keine Disjunktionen formuliert werden können, sind die Zahlen 0-9 zu überprüfen. Jedes `Track`-Individuum mit korrekter Bezeichnung wird somit durch Anwendung der Regeln des Listings 4.37 der Verifikationsklasse `CorrectlyNamedTrack` zugewiesen.

```
Track(?t), hasId(?t, ?id),
startsWith(?id, 0)
 -> CorrectlyNamedTrack(?t)

...

Track(?t), hasId(?t, ?id),
startsWith(?id, 9)
 -> CorrectlyNamedTrack(?t)
```

Listing 4.37: Planungsrichtlinie für numerische Bezeichner formalisiert in Verifikationsregeln

Eine weitere Planungsrichtlinie besagt, dass Bahnhöfe grundsätzlich mit Hauptsignalen abgesichert werden müssen. Das Listing 4.38 zeigt die entsprechende Verifikationsregel für diese Richtlinie. Hierbei wird auch eine weitere Modellierungseleganz deutlich. Innerhalb der einzelnen Regeln können die Individuen der Verifikationsklassen verwendet werden. So wird in der besagten Regel das Atom `CorrectlyPlacedSignal` verwendet, welches eine Verifikationsklasse darstellt. Diese Form der Modellierung er-

laubt eine Modularisierung und somit Verringerung der Komplexität der einzelnen Regeln. In der besagten Regel des Listings wird somit gewährleistet, dass nur mit Signalen gearbeitet wird, die schon erfolgreich einem Verifikationsprozess hinsichtlich ihrer Platzierung unterzogen wurden.

```
1 Track(?t1), Track(?t2), DifferentFrom(?t1, ?t2),
2 hasTrackType(?t1, open), hasTrackType(?t2, station),
3 directConnection(?t2, ?t1), CorrectlyPlacedSignal(?s),
4 on(?s, ?t1), hasSignalType(?s, main)
5 -> CorrectlySignalledEntryTrack(?t1)
```

Listing 4.38: Planungsrichtlinie für numerische Bezeichner formalisiert in Verifikationsregeln

4.4.7 Regeln der *RI* Rule Ontology*

Die folgende Tabelle zeigt einige relevante Regeln, die im Rahmen der Arbeit umgesetzt wurden. Da die betreffenden OWL-Klassen eine hierarchische Struktur aufweisen, sind ebenso die Regeln der Oberklasse auf die jeweiligen Unterklassen anwendbar.

Klasse	Regel	Regeltyp
`BaseIdName`		
	CorrectUniqueness	Verifikationsregel
`TwoDimObject`		
	CorrectRelation	Verifikationsregel
	CorrectDirection	Verifikationsregel
	CorrectlyConnectedTwoDimObject	Verifikationsregel
	DirectConnection	Property-Regel
`Track`		
	CorrectTrackDirection	Verifikationsregel
	CorrectSpeed	Verifikationsregel
	CorrectBegin	Verifikationsregel
	CorrectEnd	Verifikationsregel
	CorrectlyNamedTrack	Verifikationsregel, Formale Planungsregel
	CorrectlySignalledEntryTrack	Verifikationsregel
	DirectTrackConnection	Property-Regel
`Switch`		

Fortsetzung auf der nächsten Seite

Fortsetzung der vorigen Seite

Klasse	**Regel**	**Regeltyp**
	CorrectTrackType	Verifikationsregel
	CorrectlyNamedSwitch	Verifikationsregel, Formale Planungsregel
	DirectSwitchConnection	Property-Regel
`OrdinarySwitch`		
	CorrectOSConnectionCount	Verifikationsregel
	CorrectOSTrafficability	Verifikationsregel
`SlipSwitch`		
	CorrectSSConnectionCount	Verifikationsregel
	CorrectSSTrafficability	Verifikationsregel
`SingleSlipSwitch`		
	CorrectSSSConnectionCount	Verifikationsregel
	CorrectSSSTrafficability	Verifikationsregel
`DoubleSlipSwitch`		
	CorrectDSSConnectionCount	Verifikationsregel
	CorrectDSSTrafficability	Verifikationsregel
`SingleConnectionObject`		
	CorrectSCOConnectionCount	Verifikationsregel
	CorrectlyPlacedSCO	Verifikationsregel, Arithmetische Regel
`BufferStop`		
	CorrectBSType	Verifikationsregel
	CorrectlyNamedBS	Verifikationsregel, Formale Planungsregel
	CorrectlyPlacedBS	Verifikationsregel, Arithmetische Regel
`TransitionPoint`		
	CorrectInterlockingCenterTP	Verifikationsregel
	CorrectlyPlacedTP	Verifikationsregel, Arithmetische Regel
`ZeroDimObject`		
	CorrectRelation	Verifikationsregel
`Signal`		
	CorrecTSignalFunction	Verifikationsregel
	CorrectSignalType	Verifikationsregel
	CorrectSignallingSystem	Verifikationsregel
	CorrectSwitchability	Verifikationsregel

Fortsetzung auf der nächsten Seite

Fortsetzung der vorigen Seite		
Klasse	**Regel**	**Regeltyp**
	CorrectlySignalDirection	Verifikationsregel
	CorrectlyPlacedSignal	Verifikationsregel, Arithmetische Regel
	CorrectlyNamedSignal	Verifikationsregel, Formale Planungsregel
	CorrectMainDistantSignalCorrelation	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	CorrectKSSignalFunction	Verifikationsregel, Formale Planungsregel
	CorrectKSSignalTypes	Verifikationsregel, Formale Planungsregel
	CorrectDistantSignalTypes	Verifikationsregel, Formale Planungsregel
	CorrectRepeaterSignalTypes	Verifikationsregel, Formale Planungsregel
	CorrectHISignalSpeed	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	CorrectDistantMainSignalDistance	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	CorrectDistantRepeaterSignalDistance	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	CorrectRepeaterMainSignalDistance	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	IncorrectlyOrientedSignal	Negativ-Regel, Formale Planungsregel, Verifikationsregel
`TrainDetectionElement`		
	CorrectlyTDEDirection	Verifikationsregel
	CorrectlyPlacedTDE	Verifikationsregel, Arithmetische Regel
	CorrectlyNamedTDE	Verifikationsregel, Formale Planungsregel
Fortsetzung auf der nächsten Seite		

Fortsetzung der vorigen Seite

Klasse	**Regel**	**Regeltyp**
`AxleCounter`		
	CorrectACType	Verifikationsregel
	CorrectDetectsDirection	Verifikationsregel
	CorrectMaxSpeed	Verifikationsregel
	CorrectlyPlacedAC	Verifikationsregel, Arithmetische Regel
	CorrectlyNamedAC	Verifikationsregel, Formale Planungsregel
`TrainProtectionElement`		
	CorrectTPEFieldType	Verifikationsregel
	CorrectTPEPowerSupply	Verifikationsregel
	CorrectlyPlacedTPE	Verifikationsregel, Arithmetische Regel
	IncorrectlyOrienteTPE	Negativ-Regel, Formale Planungsregel, Verifikationsregel
	CorrectlyNamedTPE	Verifikationsregel, Formale Planungsregel
`LevelCrossing`		
	CorrecTLCProtection	Verifikationsregel
	CorrectlyPlacedLC	Verifikationsregel
	CorrectlyNamedLC	Verifikationsregel, Formale Planungsregel
`Balise`		
	CorrectlBAIsActive	Verifikationsregel
	CorrectBAType	Verifikationsregel
	CorrectlyPlacedBalise	Verifikationsregel, Arithmetische Regel
	CorrectlyNamedBalise	Verifikationsregel, Formale Planungsregel
`ControlCenter`		
	CorrectInterlockingCenter	Verifikationsregel
`InterlockingCenter`		
	CorrectInterlockingType	Verifikationsregel
`BlockSection`		
	CorrectBSType	Verifikationsregel
	CorrectBSRelation	Verifikationsregel

Fortsetzung auf der nächsten Seite

Fortsetzung der vorigen Seite		
Klasse	**Regel**	**Regeltyp**
`Route`	CorrectBSBegin	Verifikationsregel, Arithmetische Regel
	CorrectBSEnd	Verifikationsregel, Arithmetische Regel
	IncorrectRouteMultipleSameBS	Negativ-Regel, Formale Planungsregel, Verifikationsregel
	CorrectRouteProtection	Verifikationsregel
	CorrectRouteStartSignal	Verifikationsregel
	CorrectRouteEndingSignal	Verifikationsregel
`ComposedRoute`	CorrectRoutesOverlap	Verifikationsregel
`Overlap`	CorrectOverlapDimension	Verifikationsregel, Formale Planungsregel, Arithmetische Regel
	CorrectRouteOVRelation	Verifikationsregel
`FlankProtectionElement`	CorrectFPEProtectiveState	Verifikationsregel
	CorrectFPERelation	Verifikationsregel

Tabelle 4.1: Liste relevanter Regeln in Bezug auf OWL-Klassen der *RI* Core Ontology*

4.5 Specific RI Ontology

Die *Specific RI Ontology* enthält die konkreten EI-Daten, die verifiziert werden sollen. Im Wesentlichen besteht die *Specific RI Ontology* aus Individuen der EI-Domänen-Klassen aus der *RI* Core Ontology*. Über den OWL-Import Mechanismus können sowohl die *RI* Core Ontology* als auch deren Basis, die *RI* Graph Ontology*, eingebunden werden. Somit stehen deren Klassen und *properties* zur Verfügung.

Die Daten in der *Specific RI Ontology* stammen aus Planungsprogrammen für EI wie beispielsweise einem bei der Deutschen Bahn eingesetzten Editor. Über eine Export-Schnittstelle können die Planungsdaten des Editors in ein XML-Format überführt werden. Diese unterliegt einem (generierten) XML-Schema, welches für grundlegende Validierungen der Exportdatei genutzt werden kann. Die Exportdatei verfügt über

eine sehr flache Verschachtelung der XML-Elemente. Differenzierte XML-Techniken wie beispielsweise die Verwendung von ID/IDREF-Konstrukten zur Referenzierung bzw. Identifikation werden jedoch nicht verwendet.

Einen Auszug aus dem Export-Format zur Darstellung eines Vorsignals ist in Listing 4.39 dargestellt.

```
<Signal SignalID="80-VA">
 <ZUGLENKUNG>nein</ZUGLENKUNG>
 <GrundDaten>
  <Btrst BtrstID="4" USMax="" BfKennung=""/>
  <Stellwerk StwID="4" index="1"/>
  <Oberflaechenbezeichner>VA</Oberflaechenbezeichner>
  <ESTWBedienbezeichner>
      <Kennzahl>80</Kennzahl>
      <Kennbuchstabe/>
      <OertlicherElementname>VA</OertlicherElementname>
  </ESTWBedienbezeichner>
 </GrundDaten>
 <ZUGEHOERIGES_GLEISELEMENT>
  <Btrst BtrstID="3" USMax="" BfKennung=""/>
  <Stellwerk StwID="3" index="1"/>
  <ESTWBedienbezeichner>
      <Kennzahl>81</Kennzahl>
      <Kennbuchstabe>B</Kennbuchstabe>
      <OertlicherElementname>1</OertlicherElementname>
  </ESTWBedienbezeichner>
 </ZUGEHOERIGES_GLEISELEMENT>
 ...
 <SignalTyp typ="Vorsignal">
  <IndexVorsignal index="4"/>
 </SignalTyp>
 <KM Wert="29.800"/>
</Signal>
```

Listing 4.39: Instanz eines Vorsignals im Export-Format des *BEST Editors*

Mit Hilfe von XSLT-Skripten wird eine Transformation der XML-Daten ins OWL-Format (XML-Syntax) ermöglicht. Bei der Transformation werden jedoch nicht nur einzelne Individuen erzeugt, sondern es werden in einem mehrstufigen Prozess noch weitere Transformationsschritte unternommen, da zum einen das XML-Dokument noch vorformatiert werden muss, damit es entsprechend transformiert werden kann. Zum anderen wird nicht nur von einem XML-Format in ein anderes transformiert, sondern von einem Modellierungsparadigma in ein anderes. Das Export-Format bezieht sich auf das Paradigma der CWA, währenddessen OWL, wie schon hinlänglich beschrieben, dem Paradigma der OWA unterliegt.

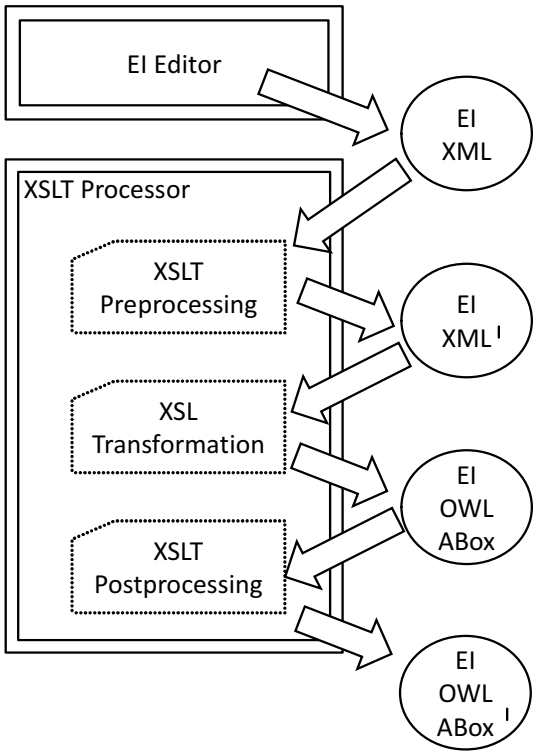

Abbildung 4.9: Ablauf der XSL-Transformation

Die Abbildung 4.9 zeigt die stufenweise Transformation bis zur verifizierbaren *Specific RI Ontology* (In der Abbildung: *EI OWL Abox'*). Aus dem EI Editor wird das *EI XML* Dokument exportiert. Diese wird mithilfe eines XSLT-Processors und dem *XSLT Preprocessing* Skript in das *EI XML'* Dokument transformiert.

Das XSLT Preprocessing Skript beinhaltet Templates für folgende Aktionen:

- **Trimming** - Beim Trimming werden unnötige Leerzeichen aus Strings entfernt, die sich nachteilig für eine Transformation in das OWL-Format auswirken können.

- **Normalisierung** - Die Normalisierung bezieht sich auf das Angleichen von Elementen und Inhalten nach bestimmten Kriterien. So wird beispielsweise aus mehreren Elemente ein Element generiert, welches eine ID darstellt (`<Kennzahl>80 </Kennzahl><OertlicherElementname>VA</OertlicherElementname> -> <identifier>80-VA</identifier>`).

- **Vereinheitlichung** - Hierbei werden alle Sonderzeichen wie Umlaute, scharfes S etc. ersetzt. Außerdem werden manche Strukturen vereinheitlicht. So ist bei-

spielsweise im Ausgangsdokument teilweise das Element `<ZUGEHOERIGES_GLEISELEMENT>` und an anderer Stelle das Element `<ZugehoerigesGleiselement>` zu finden. Beide Elemente beinhalten jedoch die gleichen Unterstrukturen. Solche Synonyme werden zu einer einheitlichen Elementbezeichnung aufgelöst.

Nach der Anwendung des *XSLT Preprocessing* Skripts wird das Ausgangsdokument (*EI XML'*) mithilfe des *XSL Transformations* Skripts in die XML-Repräsentation von OWL transformiert. Ein Auszug dieses Skripts ist in Listing 4.40 zu finden. Dargestellt wird hier die Transformation von Signalen.

```
<xsl:template match="fwml:Signal">
  <oc:Signal>
    <xsl:attribute name="rdf:ID">
      <xsl:value-of select="translate(@SignalID, '-', '')" />
    </xsl:attribute>
    <oc:baseName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      <xsl:value-of select="translate(@SignalID, '-', '')" />
    </oc:baseName>
    <oc:position rdf:datatype="http://www.w3.org/2001/XMLSchema#float">
      <xsl:value-of select="fwml:KM/@Wert" />
    </oc:position>
    <oc:on>
      <xsl:attribute name="rdf:resource">
        <xsl:value-of select="concat('#',fwml:ZUGEHOERIGES_GLEISELEMENT/
            fwml:ESTWBedienbezeichner/fwml:Kennzahl,
          fwml:ZUGEHOERIGES_GLEISELEMENT/fwml:ESTWBedienbezeichner/
              fwml:Kennbuchstabe,
          fwml:ZUGEHOERIGES_GLEISELEMENT/fwml:ESTWBedienbezeichner/
              fwml:OertlicherElementname)" />
      </xsl:attribute>
    </oc:on>
    ...
    <oc:hasSignalType>
      <xsl:attribute name="rdf:resource">
        <xsl:value-of select="concat('http://www.comsys.informatik.uni-
            kiel.de/svn/milo/trunk/ontologies/ont-classes.owl#',
            fwml:SignalTyp/@typ)" />
      </xsl:attribute>
    </oc:hasSignalTyp>
  </oc:Signal>
</xsl:template>
```

Listing 4.40: Auszug aus dem XSLT-Skript zur Transformation eines Signals aus dem Export-Format in die *Specific RI Ontology*

Für Verifikationszwecke sind manche Begleiterscheinungen der OWA hinderlich, weswegen schon auf der Transformationsebene versucht wird, diese Nachteile auszugleichen. Dieser Ausgleich erfolgt durch das *XSLT Postprocessing* Skript. Bei dessen

Anwendung werden alle Individuen über ein `AllDifferent`-Axiom als unterschiedlich zueinander deklariert werden. Somit greift auch die UNA. Eine weitere Vorgehensweise zu Verringerung der Einflüsse der OWA ist das Beschränken der der Menge der Individuen einer Klasse auf die benannten Individuen. Mithilfe des *XSLT Post-processing* Skripts wird eine Verwendung von OWL *value partitions* in der Ontology ermöglicht. So wird beispielsweise die Klasse `Signal` als Äquivalenzklasse zu einer anonymen Klasse bestehend aus der Menge aller definierten `Signal`-Individuen deklariert (vgl. Listing 4.41).

```
1 EquivalentClasses(:Signal ObjectIntersectionOf(ObjectOneOf(:80-VA :80-A
    :81-VVA :81-VA :81-A ...) :ZeroDimObject))
```

Listing 4.41: *Value Partition* für die Klasse `Signal` zur Minimierung der OWA-Effekte

4.6 Verifikation

Im Verifikationsvorgang werden konkrete Planungsdaten dahingehend überprüft, ob sie behördlichen Vorgaben und Planungsrichtlinien entsprechen. Im Rahmen dieser Arbeit werden Möglichkeiten aufgezeigt, diese händische Tätigkeit durch die Verwendung formaler, semantischer Repräsentationsformen sowohl der Planungsdaten, als auch der Planungsrichtlinien und EI-Konzepte, zu automatisieren.

Sind nun die Planungsdaten im XML-Format mithilfe der XSLT-Skripte in die ABox, die *Specific RI Ontology* (vgl. voriges Kapitel), transformiert und die TBox, die *RI* Ontologie Sammlung*, in diese importiert worden, kann eine Verifikation angestoßen werden. Der automatisierte Verifikationsprozess wird mithilfe eines Reasoners durchgeführt. Hierbei wird die ABox gegen die TBox verifiziert und so ein Verifikationsergebnis produziert.

Das Verifikationsergebnis ist eine erweiterte ABox, also im Wesentlichen die *Specific RI Ontology*, die durch den Reasoningprozess um weitere Fakten (OWL: *Assertions*) ergänzt wurde. Im Rahmen des Verifikationsprozesses können diese Fakten Klassenzugehörigkeiten von Individuen oder aber auch Eigenschaftszuweisungen zwischen zwei Individuen (*ObjectProperty*) oder zwischen einem Individuum und einem Wert (*DataypeProperty*) sein. Wenn widersprüchliche Fakten im Reasoningprozess entstanden sind, wird die gesamte Wissensbasis inkonsistent. Dies ist einer von mehreren möglichen Hinweisen auf eine fehlerhafte Modellierung der konkreten Planungsdaten. Inkonsistenzen können beispielsweise auftreten, wenn ein Individuum zwei Klassenzuweisungen erhält, bei denen die Klassen disjunkt zueinander sind.

Das Verifikationsergebnis des in dieser Arbeit vorgestellten automatisierten Verifikationsprozesses kann folgende Ausprägungen haben:

1. **Inkonsistente Wissensbasis** - Ist das Ergebnis einer Verifikation eine inkonsistente Wissensbasis, wurde dies durch Entdeckung von Widersprüchen in den Fakten während des Reasoningprozesses verursacht. Durch die OWL unterliegende Beschreibungslogik konnten diese Widersprüche ermittelt werden. Die Planungsdaten sind somit fehlerhaft und die Verifikation gilt als fehlgeschlagen. Eine genaue Eingrenzung der Fehler und ihrer Ursache ist aufgrund der komplexen Zusammenhänge in der Wissensbasis und der verketteten Wechselwirkungen der Individuen zueinander in der Regel nur eingeschränkt möglich.

2. **Individuenzuweisungen zu Fehlerklassen** - Wie im Kapitel der *RI* Rule Ontology* beschrieben, wurden Fehlerklassen eingeführt, die gängige und offensichtliche Fehler in der Modellierung während des Verifikationsprozesses abfangen. Sind diese Fehlerklassen nach der Verifikation nicht leer, befinden sich also Individuenzuweisungen zu diesen Fehlerklassen in der Wissensbasis, ist ebenso von fehlerhaften Planungsdaten auszugehen und die Verifikation gilt als fehlgeschlagen. Da die entsprechenden Individuen der Fehlerklassen auf die Ursachen der Fehlplanung hindeuten, ist eine Eingrenzung der Fehlerursachen möglich.

3. **Fehlende Individuen in Verifikationsklassen** - Durch den Verifikationsprozess werden sämtliche Individuen, die die Planungsdaten repräsentieren, überprüft und im Falle einer positiven Entsprechung der Regeln der *RI* Rule Ontology* den jeweiligen Verifikationsklassen zugeordnet. Eine Software zählt nach dem Verifikationsprozess alle Individuen einer Basisklasse und vergleicht sie mit der Anzahl der Individuen der entsprechenden Verifikationsklassen. Sind diese Mengen nicht identisch, fehlen somit Individuen in den Verifikationsklassen. Diese Individuen konnten dementsprechend nicht positiv verifiziert werden und die Verifikation gilt als fehlgeschlagen. Die fehlenden Individuen in den entsprechenden Verifikationsklassen sind ein Indikator für die Ursachen der Fehlplanung. So ist eine Eingrenzung der Fehlerursachen möglich.

4. **Positives Verifikationsergebnis** - Treffen keine der Punkte 1 - 3 zu, ist das Ergebnis eine positive Verifikation. Sämtliche Individuen, die die Planungsdaten repräsentieren, entsprechen somit den Vorgaben der Regeln.

Der angestrebte, durch die vorliegende Arbeit optimierte Prozess der Ausführungsplanung beinhaltet folgende Schritte:

1. Bearbeitung eines Modells in einem Planungseditor gemäß der Daten aus der

Entwurfsplanung.

2. Export der Planungsdaten in ein XML Export-Format zur weiteren Verarbeitung.

3. Transformation des exportierten Dokuments in die *Specific RI Ontology* (vgl. Kapitel 4.5).

   a) Validierung des exportierten Dokuments gegen das entsprechende Export-Format-Schema.

   b) Anwendung des XSLT-Preprocessing-Skripts, zur Vorverarbeitung des Export-Dokuments.

   c) Anwendung des Haupt-XSLT-Skripts zur Transformation in OWL XML Syntax. Hierbei wird eine Instanz der *Specific RI Ontology* erzeugt. Die Importe zu den Ontologien aus der TBox (*RI* Graph Ontology, RI* Core Ontology und RI* Rule Ontology*) werden hinzugefügt.

   d) Anwendung des XSLT-Skripts zur Verminderung der OWA-Einflüsse (All-Different-Axiome, Value Partitions).

4. Semantischer Verifikationsprozesses:

   a) Konsistenzchecks der Wissensbasis durch den Reasoner.

   b) Zuordnung von Individuen genereller Klassen zu spezialisierten Klassen (`Signal` -> `MainSignal`).

   c) Verifikation anhand der in der *RI* Rule Ontology* definierten Regeln (vgl. Kapitel 4.4):

      i. Anwendung der *property*-Regeln.

      ii. Anwendung der Negativ-Regeln.

      iii. Anwendung der arithmetischen und Vergleichs-Regeln.

      iv. Anwendung der Verifikationsregeln.

      v. Anwendung der formalen Planungsrichtlinien.

d) Am Ende des Verifikationsprozesses steht ein softwaregestützter Vergleich der Anzahl der Individuen aus den Verifikationsklassen in Bezug auf deren Basisklassen. Hieraus wird das Verifikationsergebnis abgeleitet. Dieses benennt potentielle Fehler in der Planung. Die entsprechenden Planungselemente müssen somit angepasst werden.

5. Der Verifikationsprozess ist idealerweise ohne Erkennung von Inkonsistenzen oder Planungsfehlern beendet worden. Somit kann von einer positiv verifizierten Planungsdatenbasis ausgegangen werden.

Werden im Verifikationsprozess Fehler oder Inkonsistenzen in den Planungsdaten ermittelt, werden die Schritte 1 - 4 wiederholt.

4.7 Abschließende Betrachtung

Es ist hinzuzufügen, dass die Modellierung der EI stark auf die Bedürfnisse bei der Projektierung elektronischer Stellwerke im Regionalbereich fokussiert ist. Dies ist dem Umstand gefolgt, dass somit eine Grundlage geschaffen wurde, einen ersten Teilbereich der gesamten EI-Planung und -Verifikation betrachten zu können. Aufgrund der modularen Ausprägung des ontologischen Modells können mit vergleichsweise geringem Aufwand Erweiterungen implementiert werden. So kann beispielsweise das ontologische Domänenmodell um weitere eisenbahntechnische Elemente ergänzt werden. Es können aber auch ebenso auf dem bestehende Domänenmodell aufsetzend weitere Regeln implementiert werden, die weitere Teile der Planungsrichtlinien abbilden. So kann der teilautomatisierte Verifikationsprozess mit semantischen Technologien sukzessiv in die gegenwärtige Vorgehensweise der EI-Planung integriert werden.

Die Aufteilung in die unterschiedlichen ontologischen Schichten ermöglicht eine Entkopplung der Abstraktionsebenen und minimiert den Wartungs- und Erweiterungsaufwand. Insbesondere die Kapselung der Eisenbahndomäne in die *RI* Core Ontology* bietet die Möglichkeit, dass auch Benutzer, die eher von der eisenbahntechnischen Seite vorgebildet und mit semantischen Technologien nicht tiefgehend vertraut sind, sehr schnell den Modellierungsumfang erfassen. So können sie beispielsweise bewerten, ob für spezielle Planungsvorgaben das Modell verwendet werden kann, oder ob noch Erweiterungen nötig sind.

Die Umsetzung der Verifikation mithilfe von unterschiedlichen Arten von SWRL-Regeln ermöglicht auch hier eine Übersicht über den Umfang der bereits formalisierten Planungsrichtlinie der Deutschen Bahn und zeigt eine Vorgehensweise auf, wie

weitere Teile der Ril 819 in diesen formalen Regeln modelliert werden können.

OWL unterliegt dem Paradigma der OWA. Dieses bietet eine Reihe von Vorteilen, wie die Herleitung von Fakten aus impliziten Zusammenhängen auf Basis formaler Beschreibungslogik. Die OWA ist jedoch hinsichtlich der Fähigkeit zur Datenverifikation nicht ohne Weiteres verwendbar. Durch ein entsprechendes Vorgehen zur Einschränkung der Individuenmenge sowie der Unifikation von Individuen ist es jedoch möglich OWL-Ontologien zur Datenverifikation einzusetzen. Hierbei unterstützend wirkt die Tatsache, dass die Regelsprache SWRL aus Komplexitätsgründen nur mit benannten Individuen von gängigen Reasonern verarbeitet wird. Hierdurch ist beispielsweise eine explizite Zuordnung von *properties* zu bestimmten Individuen einer Klasse überprüfbar. Über diesen Umweg können die An- oder Abwesenheit obligatorischer Eigenschaften von EI-Elementen innerhalb des Modells identifiziert werden. Diese Fähigkeit ist zwingend erforderlich für einen erfolgreichen Verifikationsprozess.

5 Debugging mit Semantic Constraints

Semantic Constraints stellen ein Konzept zur Formulierung komplexer Bedingungen an die ABox einer Ontologie und die Ermittlung von Ursachen für Verstöße gegen diese Bedingungen dar. Das Kapitel 5.1 enthält eine kurze Einführung. In Kapitel 5.2 werden die Bestandteile eines SCs sowie dessen Verarbeitung erläutert. Implementierungsdetails sind in Kapitel 5.3 beschrieben.

5.1 Einführung

In der bisherigen Arbeit wurde die Verifikation von EI mithilfe von Ontologien vorgestellt und diskutiert. Wie in Kapitel 4 beschrieben, besteht die TBox zum einen aus Klassen- und Eigenschaftsdefinitionen aus der Eisenbahndomäne sowie domänenunabhängigen Definitionen. Zum anderen besteht die TBox aus verschiedenen Arten von in SWRL formulierten Regeln zur Unterstützung der Verifikation. Der Verifikationsprozess stellt insbesondere eine Überprüfung der ABox gegen diese Verifikationsregeln dar. Wie in Kapitel 4.6 dargelegt, ist das Ergebnis des Verifikationsprozesses eine erweiterte ABox-Ontologie. Bei einer fehlerhaften EI-Datenbasis kann die ABox-Ontologie inkonsistent sein sowie Zuweisungen zu Fehlerklassen enthalten. Des Weiteren können Individuenzuweisungen zu entsprechenden Verifikationsklassen fehlen.

Die Aussage, ob die Verifikation mit einem positiven oder einem negativen Ergebnis endet, ist jedoch in vielen Fällen nicht befriedigend, da nur sehr eingeschränkt auf Fehlerursachen rückgeschlossen werden kann. Aus dieser Situation heraus entstand das Konzept der Semantic Constraints, mit dessen Hilfe es möglich ist, Ontologie-Debugging zu betreiben, und somit eine tiefergehende Analyse der ontologischen Wissensbasis durchzuführen. So können automatisiert Hinweise auf Fehlerursachen erzeugt und dem Benutzer präsentiert werden.

SC sind im Wesentlichen SWRL-Regeln, die es ermöglichen, benannte Individuen einer bestimmten Klasse zu markieren, ob sie valide sind, oder ob sie einem Debugging-Prozess unterzogen werden sollten. Die Bedingung zur Erfüllung des SC besteht aus mehreren Atomen. Sie wird in der Antezedenz der Regel formuliert. Für jene Individuen, die diese Bedingung erfüllen, wird dementsprechend die Konsequenz der Regel ausgeführt. Diese enthält eine *data property*-Zuweisung, um diese Individuen zu markieren. Erfüllen alle Individuen der Klasse die Bedingung und werden demzufolge markiert, so gilt diese Klasse als valide im Sinne des Debugging-Mechanismus. Wenn mindestens ein Individuum nicht markiert wurde, beginnt ein Reduktionsprozess der ursprünglichen SWRL-Antezedenz des SC. Der SC-Algorithmus verfolgt die Reduktionen und ordnet sie den entsprechend markierten Individuen zu. Somit kann ermittelt werden, welches Atom (allein oder zusammen mit anderen Atomen) der Antezedenz zu die Nichterfüllung der Gesamtregel in Bezug auf das spezifische Individuum beigetragen hat. Diese Atome werden dem Benutzer daraufhin als mögliche Fehlerursachen präsentiert. Der Reduktionsprozess endet, wenn alle Individuen der Klasse durch entsprechend unvollständige Sub-Constraints (SSC) markiert und dem Benutzer als fehlerhaft in Bezug auf die reduzierten Atome präsentiert worden sind. Der Reduktionsprozess wird in Kapitel 5.2.3.1 ausführlich beschrieben.

Wie bereits in Kapitel 3.3 dargestellt, beschreibt der Begriff des Debuggings üblicherweise den ganzheitlichen Prozess der Fehlerbeseitigung in einem Programm. Dies ist dem in dieser Arbeit beschriebenen Ansatz der SC nicht im vollem Umfang gleichzusetzen. Es ist jedoch anzumerken, dass durchaus Parallelen zum Debugging von Programmen existieren. Die Komposition von SCs, wie auch Programmcode von Programmiersprachen, können eine hohe Komplexität aufweisen. Somit sollte das hier vorgestellte Debugging-Konzept als mögliche Hilfestellung zur Fehlersuche aufgefasst werden. Durchaus sind Szenarien wie auch beim Debugging von Programmcodes denkbar, in denen die Fehlerursache vielschichtig ist und auch von diesem Konzept nicht direkt erfasst werden kann. Gleichwohl stellt das Debugging-Konzept der SC ein nützliches Werkzeug dar, um effizient Fehlerquellen im ontologischen Datenmodell einzugrenzen, wenngleich es sich nicht auf einen ganzheitlichen Debugging-Prozess bezieht, sondern lediglich das Auffinden von Fehlerursachen unterstützt.

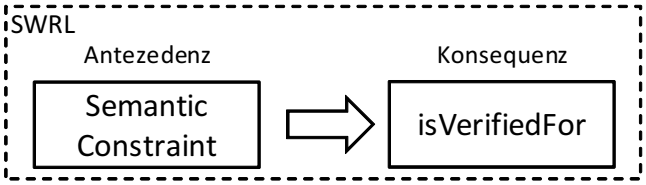

Abbildung 5.1: Schematische Darstellung der SWRL-Regel eines Semantic Constraints

5.2 Konzeptueller Aufbau des Debugging Systems

5.2.1 Bestandteile eines Semantic Constraints

Wie beschrieben, besteht ein SC im Wesentlichen aus einer Verifikationsregel. Diese folgt einem bestimmten Schema (vgl. Abbildung 5.1). Im Antezedenzteil der Regel wird vom Benutzer die Bedingung in Form von Atomen formuliert. Diese bestehen aus Konzepten, Eigenschaften oder *SWRL Builtins*. Der Konsequenzteil der Regel ist für den Benutzer nicht sicht- und editierbar. Er wird für den Auswertungsalgorithmus benötigt. Der Konsequenzteil enthält lediglich die in der SC-Ontologie definierte *data property* `isVerifiedFor`. Diese hat, wie in Listing 5.1 beschrieben, als Domain `owl:Thing` und ist somit auf jede Klasse anwendbar. Der Range von `isVerifiedFor` bezieht sich auf einen String. Dieser beinhaltet die jeweilige ID des ursprünglichen SC oder einer von ihm abgeleiteten Reduktion (SSC).

```
1 Declaration(DataProperty(:isVerifiedFor))
2 DataPropertyDomain(:isVerifiedFor owl:Thing)
3 DataPropertyRange(:isVerifiedFor xsd:string)
```

Listing 5.1: Deklaration der `isVerifiedFor` *data property*

Neben der Verifikationsregel besteht ein SC aus einer Instanz der Klasse `Semantic-Constraint` und diversen Eigenschaftszuweisungen. Die Klasse und die zugehörigen Eigenschaften sind auf der Abbildung 5.2 dargestellt und werden im Folgenden erläutert.

Jedem Individuum der SC-Klasse wird über die *data property* ein Name in Form eines Strings zugeordnet. Hierbei wird durch die Verwendung des `HasKey`-Konstrukts aus OWL 2 eine Eindeutigkeit des Namens erzwungen.

Über die boolsche *data property* `isEnabled` ist die Einsatzfähigkeit des SC gesteuert. So existieren beispielsweise Szenarien, in denen konkurrierende Mengen von SCs vorhan-

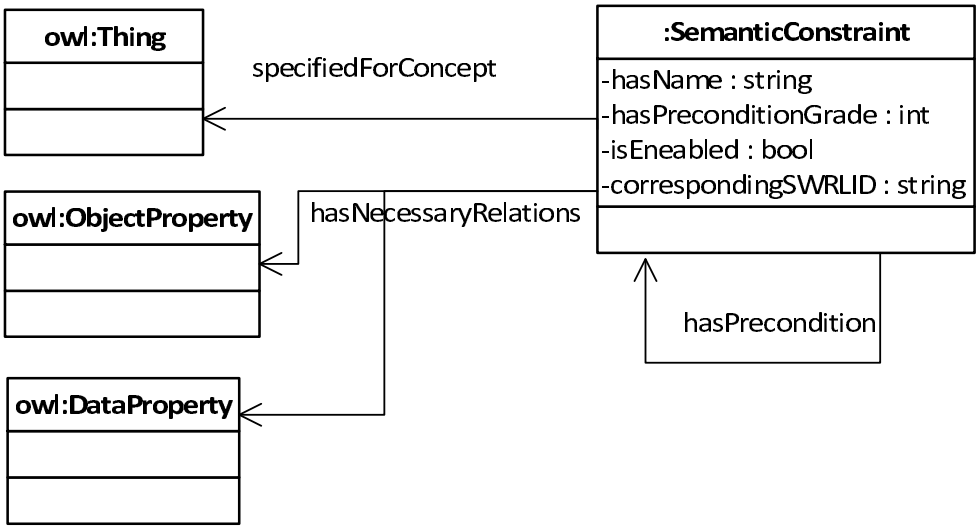

Abbildung 5.2: Bestandteile der OWL-Klasse `Semantic Constraint`

den und nicht gleichzeitig eingesetzt werden sollen. Als Beispiel aus der Eisenbahnwelt existieren beispielsweise konträre Bedingungen für die Planung von Regional- oder Vollstellwerken. Das Domänenmodell der TBox ist in diesem Fall gleich, jedoch die Planungsregeln unterscheiden sich in einigen Aspekten. Aus diesem Grund ist es möglich, gezielt einzelne oder mehrere SCs zu deaktivieren.

Die ID der Basis-Verifikationsregel, in der die Bedingungen des SC formuliert sind, wird dem SC-Individuum über die Eigenschaft `correspondingSWRLID` zugeordnet. Die IDs sind für den Reduktionsprozess wichtig, denn über sie werden die Abschwächungen der jeweiligen Bedingung protokolliert.

Grundsätzlich ist ein SC für die Überprüfung aller Individuen einer einzelnen Klasse zuständig. Um welche Klasse es sich hierbei handelt, wird in der *object property* `specifiedForConcept` angegeben. Durch die Range-Angabe `owl:Thing` ist festgelegt, dass jede beliebige Klasse einer Überprüfung mit SCs unterzogen werden kann. Die *object property* ist funktional, somit kann sich jedes SC-Individuum auf nur eine zu prüfende Klasse beziehen.

SCs werden von Benutzern erzeugt. Der Reduktionsprozess erfolgt jedoch automatisiert. Um dem Benutzer Einfluss darauf zu geben, die Sinnhaftigkeit der in der Regel-Antezedenz formulierten Bedingung zu gewährleisten, können Teile dieser Bedingung als notwendig markieren. Diese werden daraufhin nicht im Rahmen des Reduktionsprozesses aus dem Basis-Constraint entfernt. Um die besagten Atome zu markieren, verfügt ein SC-Individuum über die `hasNecessaryRelations` Eigenschaft,

in der entsprechende, nicht zu entfernende *object* oder *data properties* zugewiesen werden können.

Um die Komplexität bei der Formulierung von SC zu vermindern, und dem DRY[1] Prinzip entsprechen zu können, wurde ein Hierarchisierungsmechanismus entworfen. Somit können SCs andere SCs als Vorbedingung beinhalten. Wenn diese Vorbedingungen nicht erfüllt sind, wird der eigentliche SC gar nicht erst verarbeitet. Um einem SC-Individuum eine Vorbedingung zuzuordnen, wurde die irreflexive *object property* `hasPrecondition` eingeführt (vgl. Listing 5.2). Sowohl als Domain als auch als Range hat diese *object property* als Wertebereich die Klasse `SemanticConstraint`. Des Weiteren existiert noch die *data property* `hasPreconditionGrade`. Diese wird intern dazu verwendet, die Tiefe in der Vorbedingungshierarchie zu verfolgen. Der Hierarchisierungsmechanismus wird detailliert in Kapitel 5.2.4 beschrieben.

```
1 Declaration(DataProperty(:hasPrecondition))
2 IrreflexiveObjectProperty(:hasPrecondition)
3 DataPropertyDomain(:hasPrecondition :Constraint)
4 DataPropertyRange(:hasPrecondition :Constraint)
```

Listing 5.2: Deklaration der `hasPrecondition` *data property*

Nach dieser abstrakten Einführung in das Thema folgt ein Beispiel, in dem die Anwendung der Konzepte näher erläutert wird.

5.2.2 Anwendungsbeispiel

Geprüft werden soll die korrekte Positionierung aller Signale in Bezug auf die ihnen zugeordneten Blockabschnitte. Für dieses vereinfachte Beispiel ist davon auszugehen, dass ein Signal entweder korrekt innerhalb des Intervalls des ihm zugeordneten Blockabschnitts positioniert ist, oder aber, dass es sich vor oder hinter dem Gleisanfang bzw. -ende und damit außerhalb des Blockabschnittintervalls befindet. Die Beispielontologie beinhaltet drei Blockabschnitt- und drei Signalindividuen. Gemäß der Abbildung 5.3 wäre das Signal `S1` richtig positioniert in Bezug auf den ihm zugeordneten Blockabschnitt `B1`. `S2` und `S3` hingegen sind nicht richtig positioniert in Bezug auf die ihnen zugeordneten Blockabschnitte.

Um nun den SC in SWRL zu formulieren, werden die Definitionen der *RI* Core Ontology* hinzugezogen:

[1] DRY - Don't repeat yourself

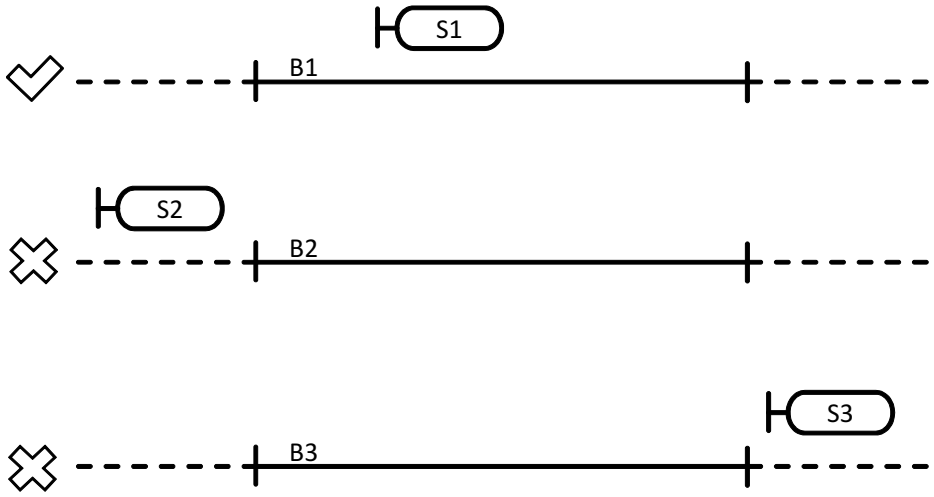

Abbildung 5.3: Korrekte und inkorrekte Positionierungen eines Signals in Bezug auf den zugeordneten Blockabschnitt

Der Klasse `Signal` der *RI* Core Ontology* ist eine *data property* `position` zugeordnet. Deren Wertebereich umfasst einen Long-Integer. Ein Gleis (Klasse: `Track`) hingegen hat eine Zuordnung zu jeweils einer *data property* für den Gleisanfang (`begin`) und für das Gleisende (`end`), ebenfalls mit den Wertebereichen von Long-Integer. Die Zuordnung eines Signals zu einem Gleis erfolgt über die *object property* `isOnTrack` der Domänenontologie. Innerhalb des SC wird nun formuliert, dass für eine Erfüllung der Wert der Signalposition größer oder gleich dem Anfang des Gleises und kleiner oder gleich dem Ende des Gleises ist. Für die Vergleiche werden die *SWRL builtins* `greaterThanOrEqual` und `lessThanOrEqual` verwendet. Die Formulierung dieser Bedingung erfolgt im Antezedenz-Teil der SWRL-Regel. Dieser ist in Listing 5.3 dargestellt. Nicht sichtbar ist der Konsequenz-Teil der Regel, da er nur vom SC-Algorithmus und nicht vom Benutzer verwendet wird. Wie bereits beschrieben, besteht der Konsequenz-Teil immer aus einem einzelnen Atom, einem `isVerifiedFor`-Ausdruck. Bezüglich dieses Beispiels lautet der generierte Ausdruck: `isVerifiedFor(?s,'C')`. '?s' ist gemäß der SC-Bedingung (dem Antezedenz-Teil der SWRL-Regel) eine Stellvertreter-Variable für alle `Signal`-Individuen der Ontologie. Der String 'C' bezieht sich auf die ID der SWRL-Regel des SC. Alle `Signal`-Individuen, welche die Antezedenz erfüllen, erhalten in der Konsequenz diese `isVerifiedFor`-Zuweisung mit der entsprechenden SC-ID oder im Zuge des Reduktionsprozesses generierten Sub-SC-ID. Somit kann nachvollzogen werden, welche Individuen welche SCs erfüllen.

```
1 Signal(?s), Track(?t), isOnTrack(?s,?t), position(?s,?p), begin(?t,?b),
2 end(?t,?e), swrlb:greaterThanOrEqual(?p,?b) swrlb:lessThanOrEqual(?p,?e)
```

Listing 5.3: Semantic Constraint zur Signalpositionierung

Das entsprechende SC-Individuum 'SC_I1' des Beispiels ist in Listing 5.4 dargestellt. Wie im letzten Absatz beschrieben, bezieht sich der SC nur auf Individuen der `Signal`-Klasse (property: `specifiedForConcept`). Des Weiteren ist er der SWRL-Regel mit der ID 'SC1' zugeordnet (property: `correspondingSWRLID`). In dieser ist die eigentliche Bedingung formuliert und sie wird im Zuge des Reduktionsprozesses verarbeitet. Der SC ist aktiviert (property: `isEnabled`) und verfügt über keine Vorbedingungen, die sich in Zuweisungen der property `hasPrecondition` widerspiegeln würden. Allerdings existiert eine Zuweisung der property `hasNecessaryRelations`. In ihr wurde vom Benutzer definiert, dass das SWRL-Antezedenz Atom `dom:on` nicht im Rahmen des Reduktionsprozesses entfernt werden soll. Eine Entfernung dieses Atoms hätte zur Folge, das der Bezug zwischen einem Signal und dem zugeordneten Gleis verloren gehen würde. Die Bedingung würde dann keinen Sinn ergeben, da alle Signale mit allen Gleisen verglichen werden würden und nicht nur jene, die wirklich im definierten geographischen Zusammenhang stehen.

```
1 Declaration(NamedIndividual(:SC_I1))
2 ClassAssertion(:SemanticConstraint :SC_I1)
3 DataPropertyAssertion(:correspondingSWRLID :SC_I1 "C"^^xsd:string)
4 DataPropertyAssertion(:hasName :SC_I1 "SC1"^^xsd:string)
5 DataPropertyAssertion(:isEnabled :SC_I1 "true"^^xsd:boolean)
6 ObjectPropertyAssertion(:specifiedForConcept :SC_I1 dom:Signal)
7 ObjectPropertyAssertion(:hasNecessaryRelations :SC_I1 dom:on)
```

Listing 5.4: OWL-Individuum des Semantic Constraint zur Signalpositionierung

5.2.3 Verarbeitung von Semantic Constraints

5.2.3.1 Reduktionsalgorithmus

Die Verarbeitung der SCs erfolgt stufenweise. Jeder SC wird für sich verarbeitet. Erfüllen nach einer Anwendung des SC nicht alle Individuen der zu prüfenden Klasse den Antezedenz-Teil der SC-Regel, so ist dies ersichtlich, da diese Individuen nicht über eine Zuweisung der `isVerifiedFor`-property verfügen.

Nun wird der Reduktionsalgorithmus angewendet, der die schrittweise Abschwächung des Original-SC zu SSC vornimmt. Hierbei werden nach jedem Schritt einzelne Atome aus dem aktuellen SC entfernt und die noch nicht verifizierten Individuen mit diesem reduzierten SC einer erneuten Prüfung unterzogen. Dieser Prozess wird solange fortgeführt, bis alle Individuen verifiziert sind und eine `isVerifiedFor`-property-Zuweisung erhalten haben. Da es sich bei dem Reduktionsprozess um Variantenbildungen handelt, beläuft sich die Anzahl der Sub-Constraints auf 2^n, wobei n die Anzahl der

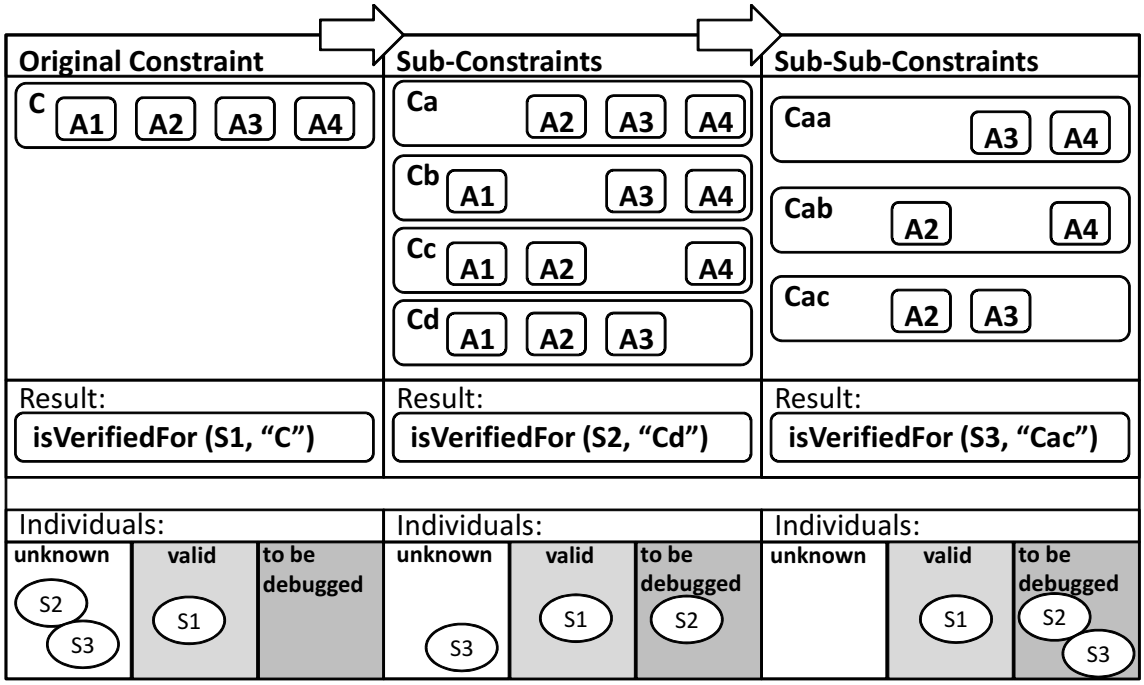

Abbildung 5.4: Reduktion eines Semantic Constraints

Atome beschreibt. Dieses exponentielle Wachstum von SSC bei linearem Wachstum der Atome ist kritisch. Es wird durch die Anwendung von Reduktionsprioritäten deutlich abgeschwächt. Detailliert wird dieses Verfahren in Kapitel 5.2.3.2 erläutert.

Während des Reduktionsprozesses wird verfolgt, welcher SSC um welche Atome reduziert wurde. Nach der erfolgten Individuen-Markierung kann so darauf geschlossen werden, welche Atome für eine Nichterfüllung des Original-SC des entsprechenden Individuums verantwortlich sind.

Das Schaubild 5.4 zeigt den Reduktionsprozess exemplarisch und lehnt sich an das Beispiel des vorigen Kapitels 5.2.2 an: Der Original-SC 'C' besteht beispielhaft aus vier Atomen. Drei `Signal`-Individuen 'S1', 'S2' und 'S3' sollen verifiziert werden. Nach Verarbeitung des Basis-SC ist erhält das Individuum 'S1' die `isVerifiedFor-`*property*-Zuweisung und ist somit positiv verifiziert. Es wurde ermittelt, dass die beiden anderen Individuen im ersten Schritt nicht positiv verifiziert worden sind. Somit beginnt der Reduktionsprozess. Hierbei werden für jedes Atom des Basis-SC ein SSC erzeugt, bei dem jeweils ein anderes Atom entfernt wurde. Nun wird die Verarbeitung mit den SSCs und den verbliebenen Individuen angestoßen. Nach der ersten Reduktion wurde 'S2' als verifiziert gegen den SSC 'Cd' durch die `isVerifiedFor-`*property* markiert. Hieraus kann geschlossen werden, dass das Atom 'A4' für die Nichterfüllung des Basis-SC in Bezug auf das Individuum 'S2' verantwortlich ist. Da 'S3' auch nach der ers-

ten Reduktion nicht positiv verifiziert worden ist, erfolgt eine weitere Reduktion. Im Schaubild ist exemplarisch die Reduktion des SSC 'Ca' dargestellt. Es erfolgen jedoch analog dazu auch die Reduktionen und Verarbeitungen der übrigen SSCs. Nach dieser zweiten Reduktion wird nun auch 'S3' als letztes verbliebenes Individuum positiv verifiziert gegen den Sub-SSC 'Cac'. Hieraus kann geschlossen werden, dass die Atome 'A1' und 'A4' für die Nichterfüllung des Basis-SC in Bezug auf das Individuum 'S3' verantwortlich sind. Da nun alle Individuen eine `isVerifiedFor-`*property*-Zuweisung erhalten haben, endet hiermit der Reduktionsprozess.

Der Pseudo-Code aus dem Listing 5.5 veranschaulicht die Vorgehensweise der Verarbeitung von Semantic Constraints. Für die Verarbeitung jedes Basis-SC wird eine temporäre Ontologie erzeugt, welche die A- und TBox der Wissensbasis importiert. Jedoch wird nur der Basis-SC (und dessen Vorbedingungen - vgl. Kapitel 5.2.4) und die im Zuge des Reduktionsprozesses dynamisch erzeugten Sub-SC in diese temporäre Ontologie integriert und verarbeitet. Die dynamische Erzeugung der Sub-SC wird durch die Funktion `getNextCombination` (Zeile 20 des Listings) vollzogen. Dabei werden unterschiedlichen Atome unterschiedliche Reduktionsprioritäten zugeordnet, um möglichst schnell und ressourceneffizient den Verifikationsprozess abzuschließen. Die Reduktionsprioritäten werden im folgenden Kapitel erläutert.

```
1  foreach ( SemanticConstrain c ) {
2    to = new TempConstraintIndividualOntology ()
3    to.import ( domainOntology )
4    cExp = c.getSWRL_Expression ()
5
6    while ( ! allIndividualsVerified ) {
7      to.addRule ( cExp )
8
9      while ( c.hasPreconditions () ) {
10       pc = c.getNextPrecondition ()
11       to.addRules ( pc.getSWRL_Expression () )
12     }
13
14     axioms = SWRL-Processor.processRules ( to )
15
16     if ( axioms.containsAllIndividuals ) {
17       allIndividualsVerified = true
18     } else {
19       to.clearRules ()
20       cExp = getNextCombination ( c.getSWRL_Expression )
21   }
22 }
```

Listing 5.5: Pseudocode des Reduktionsalgorithmus'

5.2.3.2 Reduktionsprioritäten

Wie bereits dargelegt, beträgt bei einer vollständigen Abschwächung des Basis-SC die Anzahl aller SSCs 2^n. Es ist also von einem exponentiellen Zuwachs von SCs bei einem linearen Zuwachs von Atomen in der Verifikationsregel auszugehen. Diese Tatsache ist kritisch bezüglich der Skalierung. In der praktischen Umsetzung und Implementierung des Debuggingsystems wurden Maßnahmen ergriffen, die zu einer deutlichen Reduktion der Anzahl der SSCs führen.

Bereits während der Definition eines SC wird der Benutzer dazu aufgefordert, ein oder mehrere Atome des SC als elementar zu definieren. Intern werden dem SC-Individuum entsprechende `hasNecessaryRelations`-Zuweisungen zugeordnet. Diese elementaren Atome werden im Reduktionsprozess nicht entfernt. Ist ein zu überprüfendes Individuum nach dem gesamten Reduktionsprozess immer noch nicht verifiziert, so ist dies auf diese elementaren Atome zurückzuführen. Dies wird dem Benutzer entsprechend mitgeteilt.

Neben der Beibehaltung der elementaren Atome verfügt der Reduktionsalgorithmus über einen Priorisierungsmechanismus, bei dem bestimmte Atomarten bevorzugt reduziert werden. Diese Priorisierung erfolgt auf der Basis von Heuristiken, die ergeben haben, dass die entsprechenden Atom-Arten vornehmlich für Fehler im Planungsprozess verantwortlich sind. Die Prioritäten der Entfernung der Atomarten sind hierbei:

1. *SWRL builtins*
2. *individual identities*
3. *data properties*
4. *object properties*

SWRL builtins (vgl. Kapitel 2.3.4) werden vornehmlich dazu verwendet, mathematische Vergleiche durchzuführen sowie Zeichenketten-Operationen oder arithmetische Berechnungen mit SWRL vorzunehmen. Sie stellen eine Erweiterung zu Web Ontology Language dar, in der diese Operationen nicht unterstützt werden. Da sich SCs häufig auf Berechnungen und Vergleichen beziehen, ist die Wahrscheinlichkeit eines Modellierungsfehlers bei SWRL-Builtins am höchsten. Somit trägt eine Entfernung der *SWRL builtins* zu einem frühen Zeitpunkt des Reduktionsprozesses in der Regel zu einem schnellen Debugging-Resultat und damit zu einer Terminierung des Reduktionsprozesses bei.

Zu den *individual identities* zählen die Konstrukte `owl:sameAs` sowie `owl:differentFrom`. Da OWL und SWRL nicht der UNA unterliegen, können unterschiedlich benannte Individuen potentiell gleich sein. Mithilfe von *individual identities* kann eine definitive Aussage über Gleich- oder Ungleichheit von Individuen getroffen werden. Diese Konstrukte können ebenso als Atome in SWRL verwendet werden. Die falsche Verwendung von Gleich- oder Ungleichheit stellt einen essentiellen Modellierungsfehler dar. Somit ist einer Entfernung dieser Atome im Reduktionsprozess eine hohe Priorität zugeordnet.

Data properties sind als Klassen-Attribute primitiven Datentyps anzusehen. Eine Entfernung dieser Attribute aus einem SC und die darauf erfolgende Validität von Individuen in Bezug auf den abgeleiteten SSC deutet daher auf eine fehlerhafte oder fehlende Attributzuweisung der entsprechenden Individuen hin. Da auch diese Situation einen häufigen Fehlerfall widerspiegelt, werden *data properties* mit der dritthöchsten Priorität beim Reduktionsprozess beachtet.

Object properties repräsentieren Klassenbeziehungen. Diese sind prägnant im Modellierungsprozess und sind in der Regel nicht sehr fehlerbehaftet, da diese Mängel meist während der Modellierung erkannt werden. Daher sind *object properties* in der Reduktionspriorität am niedrigsten angesiedelt.

Versuche mit der RI* Ontology Assemblage haben gezeigt, dass diese Priorisierungen einen positiven Effekt haben auf die Anzahl der zu generierenden SSC im Reduktionsprozess. Ein weiterer Faktor, der zur erheblichen Reduktion beigetragen hat ist die systematische Analyse der SC Atome. Hierbei werden nicht mehr relevante Atome im Zuge der Reduktion ebenfalls entfernt. Das Listing 5.6 verdeutlicht diese Aussage: Es zeigt, dass die Entfernung eines *SWRL builtins* für einen Vergleich (hier: `swrlb:greaterThenOrEqual`) ebenfalls jene Atome (hier: `begin`) entfernt werden, wenn deren Werte-Variablen in keinem weiteren Atom des SCs Verwendung finden.

```
1 Signal(?s), Track(?t), isOnTrack(?s,?t), position(?s,?p), begin(?t,?b),
2 end(?t,?e), swrlb:greaterThanOrEqual(?p,?b) swrlb:lessThanOrEqual(?p,?e)
```

Listing 5.6: Entfernung eines *SWRL builtins* mit gleichzeitiger Entfernung einer nicht mehr relevanten *data property*

5.2.4 Hierarchisierungsmechanismus

SCs können viele Atome enthalten und somit eine hohe Komplexität aufweisen. Des Weiteren ist es denkbar, dass Teile von SCs sich in anderen SCs wiederholen, was einen hohen Wartungsaufwand bei Änderungen verursacht. Um diesen beiden Fak-

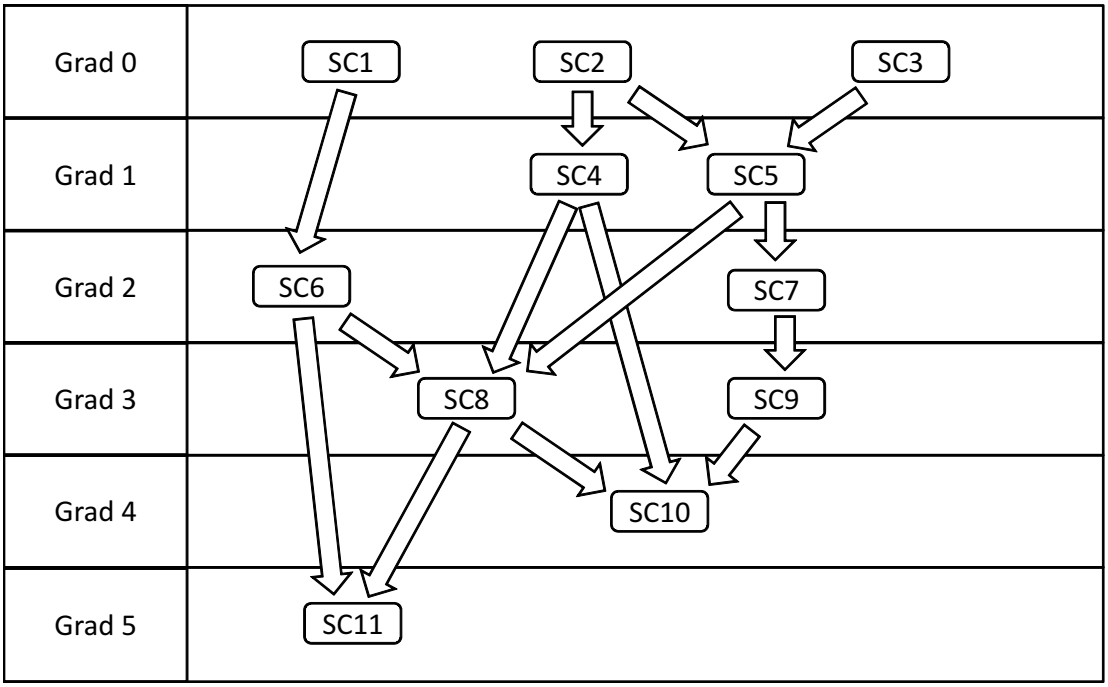

Abbildung 5.5: Hierarchien von Semantic Constraints

ten entgegenzuwirken, wurde eine Fähigkeit zur Modularisierung konzipiert. Dieser Hierarchisierungsmechanismus ermöglicht die Verwendung von SCs als Vorbedingungen zu anderen SCs. So kann beispielsweise der SC aus dem Beispiel aus Kapitel 5.2.2 zur Überprüfung der korrekten Positionierung von Signalen als Vorbedingung für einen SC definiert werden, der eine korrekte Ausrichtung von Signalen in Bezug auf die Befahrbarkeitsmöglichkeiten der entsprechenden Blockabschnitte überprüft. Vor einer Überprüfung des entsprechenden Haupt-SC werden alle Vorbedingungs-SC ausgewertet. Der Haupt-SC wird bei den Überprüfungen des Vorbedingungs-SC ausgeschlossen, da hier nur Individuen überprüft werden sollen, die die Vorbedingungen passiert haben. Über ein Caching-Mechanismus werden Ergebnisse von Vorbedingungen zwischengespeichert, um eine mehrfache Auswertung zu unterbinden.

Die Abbildung 5.5 zeigt beispielhaft die Hierarchisierung von SCs. Wie in der Abbildung erkennbar, entsteht durch die Hierarchisierung ein gerichteter Graph. Dieser darf keine Zyklen beinhalten, da diese eine Terminierung des Debugging-Prozesses verhindern. Die Gewährleistung der Azyklizität wird mithilfe des Hierarchie-Grades eines SC ermöglicht. Ein SC ohne Vorbedingung hat den Grad 0. Der Grad eines SC errechnet sich aus dem höchsten Grad seiner Vorbedingungen inkrementiert um 1. In der Implementierung (Kapitel 5.3.3) ist umgesetzt, dass ein SC nicht als Vorbedingung für einen SC niedrigeren Grades eingesetzt werden kann.

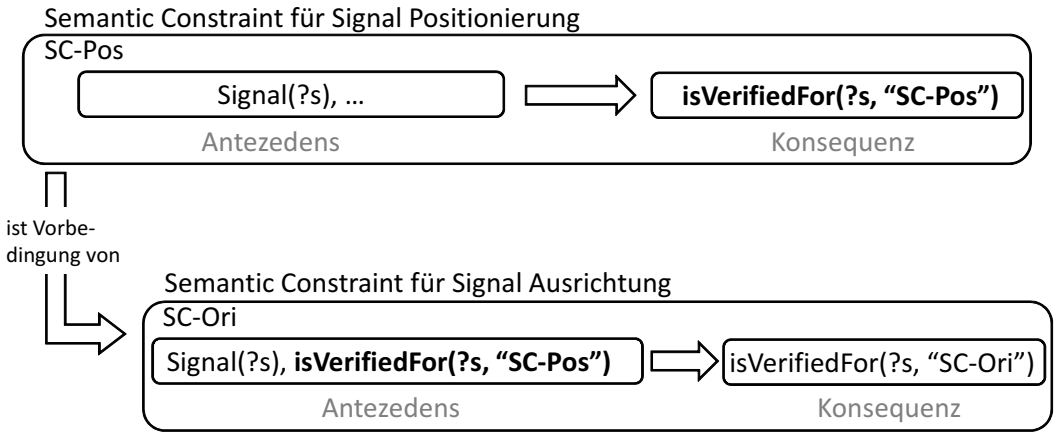

Abbildung 5.6: Vorbedingung eines Semantic Constraints

Die technische Umsetzung erfolgt mittels der SC-Klasse und den zugehörigen *properties* `hasPreconditionGrade`, `hasPrecondition` sowie `isVerifiedFor`, die bereits in Kapitel 5.2 beschrieben wurden.

Hat der Benutzer einen SC definiert (Hier: `SC-Ori` für die Überprüfung der Signal-Ausrichtung) und möchte nun einen SC als Vorbedingung hinzufügen (Hier: `SC-Pos` für die Überprüfung der Signal-Positionierung), so wird zuerst geprüft, ob der Vorbedingungs-SC `SC-Pos` über die *data property* `hasPreconditionGrade` einen gleichwertigen oder geringeren Grad aufweist. Ist dies der Fall, wird das Ergebnis des Vorbedingungs-SCs, die Konsequenz der Verifikationsregel `isVerifiedFor(?self, 'SC-Pos')`, in die Antezedenz des Haupt-SC eingebunden (vgl. Abbildung 5.6). Somit ist gewährleistet, dass nur jene Individuen von dem Haupt-SC berücksichtigt werden, die den Vorbedingungs-SC passiert und eine entsprechende `isVerifiedFor`-*property*-Zuweisung erhalten haben. Bezogen auf das Beispiel bedeutet dies: Nur alle korrekt positionierten Signale werden dahingehend überprüft, ob sie eine korrekte Ausrichtung in Bezug auf die Befahrbarkeitsmöglichkeiten des Blockabschnitts aufweisen. Der vollständige SC zur Überprüfung der korrekten Signal-Ausrichtung inklusive der Vorbedingung der korrekten Signal-Positionierung (`'SC-Pos'`) ist in Listing 5.7 dargestellt. Die Vorbedingung ist in der `isVerifiedFor`-*property* codiert. Neben den Klassen-Atomen `Signal` und `Track` und der Verknüpfung von Signal und Blockabschnitt `isOnTrack`, wird in diesem SC die *object property* `hasDirection` verwendet. Diese ist eine Eigenschaft sowohl von der Klasse `Signal` als auch der Klasse `Track`. Bei `Signal` beschreibt die Eigenschaft `hasDirection` die Wirkrichtung, während bei `Track` mit der gleichen Eigenschaft die Befahrbarkeitsmöglichkeit beschrieben wird. Das Atom `owl:sameAs(?d1,?d2)` vergleicht diese beiden Richtungen und liefert `true` zurück, wenn diese übereinstimmen.

```
1 Signal(?s), isVerifiedFor(?s,'SC-Pos'), Track(?t), isOnTrack(?s,?t),
```

```
2 hasDirection(?s,?d1), hasDirection(?t,?d2), owl:sameAs(?d1,?d2)
```

Listing 5.7: Semantic Constraint zur Überprüfung korrekter Signal-Ausrichtungen mit Vorbedingung

In dem beschrieben Beispiel beziehen sich beide SCs auf das Debuggen der Klasse `Signal` (Zugeordnete *data property* `specifiedForConcept`). Es kann jedoch der Fall eintreten, dass ein Haupt-SC und ein entsprechender Vorbedingungs-SC unterschiedliche Klassenbezüge haben. In diesem Fall werden die Atome des Haupt-SC der Vorbedingungsprüfung unterzogen, die einem Klassenbezug des Vorbedingungs-SC entsprechen oder jeweilige Unterklassen darstellen. So zeigt beispielsweise das Listing 5.8 einen SC zur Fahrstraßenüberprüfung, der gewährleistet, dass das Start- und das Endsignal der Fahrstraße nicht identisch sind. Dieser SC hat den SC zur Überprüfung korrekt ausgerichteter Signale (vgl. Listing 5.7) als Vorbedingung. Das entsprechende OWL-Individuum des SC zur Fahrstraßenüberprüfung mit der Angabe der Vorbedingung ist in Listing 5.9 dargestellt.

```
1 Route(?r), Signal(?s1), routesStartSignal(?r, s1),
2 Signal(?s2), routesEndingSignal(?r, s2),
3 DifferentFrom(?s1, ?s2)
```

Listing 5.8: Semantic Constraint zur Fahrstraßenüberprüfung

```
1 Declaration(NamedIndividual(:SC_Route1))
2 ClassAssertion(:SemanticConstraint :SC_Route1)
3 DataPropertyAssertion(:correspondingSWRLID :SC_Route1 "CR1"^^xsd:string)
4 DataPropertyAssertion(:hasName :SC_Route1 "SC_R1"^^xsd:string)
5 DataPropertyAssertion(:isEnabled :SC_Route1 "true"^^xsd:boolean)
6 ObjectPropertyAssertion(:specifiedForConcept :SC_Route1 dom:Route)
7 ObjectPropertyAssertion(:hasNecessaryRelations :SC_Route1
    owl:DifferentFrom)
8 ObjectPropertyAssertion(:hasPrecondition :SC_Route1 :SC_SignalOrient1)
```

Listing 5.9: OWL-Individuum des Semantic Constraint zur Fahrstraßenüberprüfung

5.3 Implementierte Werkzeuge

5.3.1 Technologische Basis

Die Implementierung des SC Debugging Systems wurde als Plugin für den Protégé-Editor (vgl. Kapitel 2.3.5) der Version 3.x realisiert. Protégé verfügt über einen umfangreichen Pluginmechanismus. Das SC-Plugin hat den Entwicklungsnamen 'Con-

trôlé', was aus dem französischen übersetzt 'geprüft' oder 'kontrolliert' bedeutet. Contrôlé ist ein graphische Plugin, welches über einen eigenen Reiter (ein sogenanntes *Tab Widget*) in der Protégé-GUI verfügt. Neben *Tab Widgets* existieren eine Reihe von anderen Plugin-Arten (siehe Protégé Plugins Library by Type: [4]), die die GUI erweitern oder als Backend-Erweiterung im Hintergrund ihre Funktionalitäten bereitstellen. Des Weiteren können Plugins auch nur die Ausprägung einer API haben und die Kernfunktionen von Protégé erweitern. So ist beispielsweise die Verarbeitungsmöglichkeit von OWL-Ontologien ebenso als Plugin realisiert, da Protégé ursprünglich dafür konzipiert wurde, mit einem anderen Wissensrepräsentationskonzept (Frames and Slots: siehe [58]) zu arbeiten.

Über APIs können Protégé-Kernfunktionen aufgerufen und zur Pluginsteuerung genutzt werden. Hierzu gehören beispielsweise die programmatische Erzeugung und Modifikation von OWL-Axiomen, das Einbinden eines Reasoners oder das Anstoßen eines Reasoningprozesses. Einen Überblick über die Protégé-API ist unter [28] zu finden.

Bezüglich des Contrôlé-Plugins wird als Auswertungskomponente für die in SWRL-Regeln formulierten SC-Bedingungen in Protégé 3.x die Jess Rule Engine [40] eingesetzt. Eine Anbindung eines SWRL-fähigen Reasoners zur Verarbeitung der SWRL-Regeln wurde erst mit der Protégé-Version 4.x eingeführt. Zur Auswertung von SWRL-Regeln mit Protégé 3.x existiert ein entsprechendes Plugin. Dieses steuert die Übersetzung von SWRL in die Jess Rule Language, welche der LISP-Syntax [49] ähnelt. Zur Regelauswertung verwendet Jess intern eine angepasste Version des Rete Algorithmus [15]. Die Verarbeitung von SWRL-Regeln mithilfe der SWRL-Jess-Bridge ermöglicht die Rückgabe der inferierten Fakten aus der Jess Rule Engine und das Speichern dieser in der ursprünglichen Ontologie.

Intern werden in Contrôlé mehrere Ontologien dazu verwendet, die einzelnen SC-Definitionen zu verwalten und die Abschwächungen der Original-SC zu überwachen. Die zu überprüfenden Ontologien werden hierbei nicht verändert. Die ontologische Architektur wird in Kapitel 5.3.2 detaillierter erläutert.

5.3.2 Ontologische Architektur

Grundsätzlich wird im Contrôlé Plugin der Import-Mechanismus von OWL verwendet. Somit können Ontologien untereinander verknüpft werden. Dies ermöglicht die logische Aufteilung der Wissensbasis. Beispielsweise können so Klassendefinitionen, Regeldefinitionen und Individuendekalarationen separat gespeichert werden (Aufteilung von TBox und ABox). Jede OWL-Ontologie importiert implizit eine Meta-

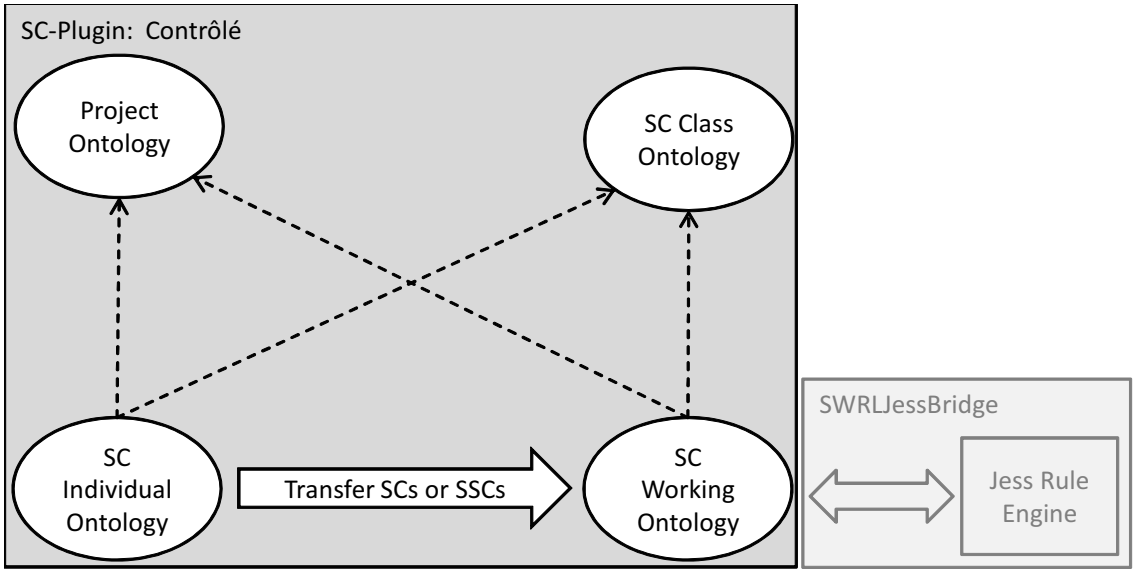

Abbildung 5.7: Ontologische Struktur des Contrôlé-Plugins

Ontologie[2]. In ihr werden die grundlegenden Konzepte wie beispielsweise `owl:Class`, `owl:DatatypeProperty` definiert. Für SWRL existiert ebenfalls eine Meta-Ontologie[3]. In ihr werden beispielsweise die SWRL-Builtins definiert.

Contrôlé verwendet intern drei Ontologien, um den Debugging-Prozess zu unterstützen. Die zu überprüfende Ontologie wird nachfolgend als *Project Ontology* bezeichnet. In Bezug auf die Eisenbahn ist die *Project Ontology* mit der *RI* Individual Ontology* gleichzusetzen. Mithilfe von Contrôlé werden die Individuen einer Klasse mit SCs überprüft. Um die Individuen der *RI* Individual Ontology* zu untersuchen, ist es notwendig, die entsprechenden Basis-Ontologien der RI* Ontology Assemblage zu importieren. Diese Import-Abhängigkeiten sind in der Abbildung 5.7 der Vereinfachung wegen nicht dargestellt.

In der *SC Class Ontology* sind die Klasse `SemanticConstraint` sowie die ihr zugeordneten *properties* definiert (vgl. Abbildung 5.2 und Kapitel 5.2.1). Diese Ontologie stellt die Basis des Contrôlé Plugins dar. Jeder SC ist ein Individuum der Klasse `SemanticConstraint` aus dieser Ontologie. Die vom Anwender definierten SC-Individuen werden in einer separaten Ontologie, der *SC Individual Ontology*, gespeichert. Diese muss somit sowohl die *SC Class Ontology* als auch die *Project Ontology* importieren. Die SC-Individuen sind die Ausgangsbasis für den Markierungsprozess. Wird dieser angstoßen, so werden bei jedem Markierungsschritt vom Algorithmus

[2] www.w3.org/2002/07/owl
[3] http://www.w3.org/2003/11/swrl

die zu verwendenden SCs oder SSCs ausgewählt und in die *SC Working Ontology* übertragen. Diese wird somit bei jedem Markierungsschritt neu generiert. Vom Algorithmus wird gewährleistet, das nur diejenigen SCs oder SSCs verwendet werden, die zu dem entsprechenden Markierungszeitpunkt relevant sind.

Die eigentliche Verarbeitung der den SCs zugrundeliegenden SWRL-Regeln erfolgt bei gemäß der Protégé 3.x Architektur in der Jess Rule Engine. Hierzu werden die SWRL-Regeln mithilfe der SWRLJessBridge in die Jess Language transformiert, die Rule Engine angestoßen und die neu erzeugten Fakten wieder nach OWL zurücktransformiert.

5.3.3 GUI Architektur

Das Contrôlé-Plugin ist eine Erweiterung der Funktionalität von Protégé. Die Protégé-GUI wird dazu um einen weiteren Reiter ergänzt. In diesem Reiter sind alle pluginspezifischen Elemente angeordnet. Die Abbildung 5.8 zeigt eine schematische Darstellung des Plugins. Im linken Bereich des Reiters ist eine Klassenhierarchie dargestellt. Sie unterscheidet sich augenscheinlich nicht von der Standard-Klassenhierarchie von Protégé. Die Auswahl einer Klasse durch den Benutzer in dieser Klassenhierarchie triggert jedoch die Anzeige der entsprechend definierten SCs im oberen rechten Bereich *Semantic Constraints* des Plugins. Hier werden in einer Liste sämtliche SCs dargestellt mit denen das in der Klassenhierarchie ausgewählte Konzept überprüft werden soll. Intern wird die Zuordnung eines SC zu dem entsprechenden Konzept über die `specifiedForConcept`-*property* hergestellt (vgl.: Kapitel 5.2). Die Liste der SC verfügt über drei Spalten. In der ersten Spalte befindet sich eine Checkbox, die es ermöglicht, zu steuern, ob ein SC aktiviert ist oder nicht (*Property*: `isEnabled`) Die zweite Spalte beinhaltet den Namen des jeweiligen SC (*Property*: `hasName`), während in der dritten Spalte verkürzt die eigentliche SC-Bedingung (*Property*: `correspondingSWRLID`) in Form von SWRL-Atomen dargestellt wird. Diese Darstellung ist gut in dem Screenshot 5.9 zu erkennen. Hier sind ebenso die Schaltflächen am oberen rechten Rand erkennbar, welche die Erstellung, Modifikation und Löschung von SCs ermöglichen. Nach einem Klick auf die entsprechenden Schaltflächen zum Erstellen oder Bearbeiten eines SCs, wird ein neues Fenster geöffnet, in dem die SC-Eigenschaften modifiziert werden können (vgl. Screenshot 5.9). Für jeden erzeugten SC wird intern ein entsprechendes SC-Individuum der Klasse `SemanticConstraint` der intern verwendeten *SC Class Ontology* erstellt.

Im rechten unteren Bereich des Contrôlé-Plugins befindet sich die Auswertungsansicht. Hier werden nach einer erfolgten Überprüfung, ebenfalls in Listenform, sämtliche Individuen der ausgewählten und überprüften Klasse dargestellt. Neben den Namen der Individuen in der ersten Spalte, werden in der zweiten und dritten Spalte die zur Überprüfung verwendeten SCs sowie die Ergebnisse dieser Verwendung dargestellt. Als Ergebnis kann, wie bereits beschrieben, entweder die Konformität mit dem Original-SC durch ein grünes 'OK' bestätigt werden, oder aber das entsprechende Individuum ist nur mit einem abgeschwächten SSC konform . Die bei der Abschwächung entfernten Regel-Atome werden daraufhin in einer entsprechenden Meldung dem Benutzer präsentiert. Des Weiteren kann in dieser Nachricht die Situation beschrieben werden, dass ein Individuum nicht mit einem Vorbedingung-SC eines entsprechenden Haupt-SC konform ist. In diesem Fall wird dem Benutzer der Name des Vorbedingungs-SC als Ursache präsentiert.

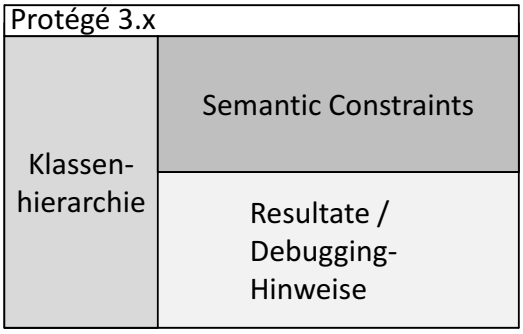

Abbildung 5.8: Schematische Darstellung des Contrôlé-Plugins

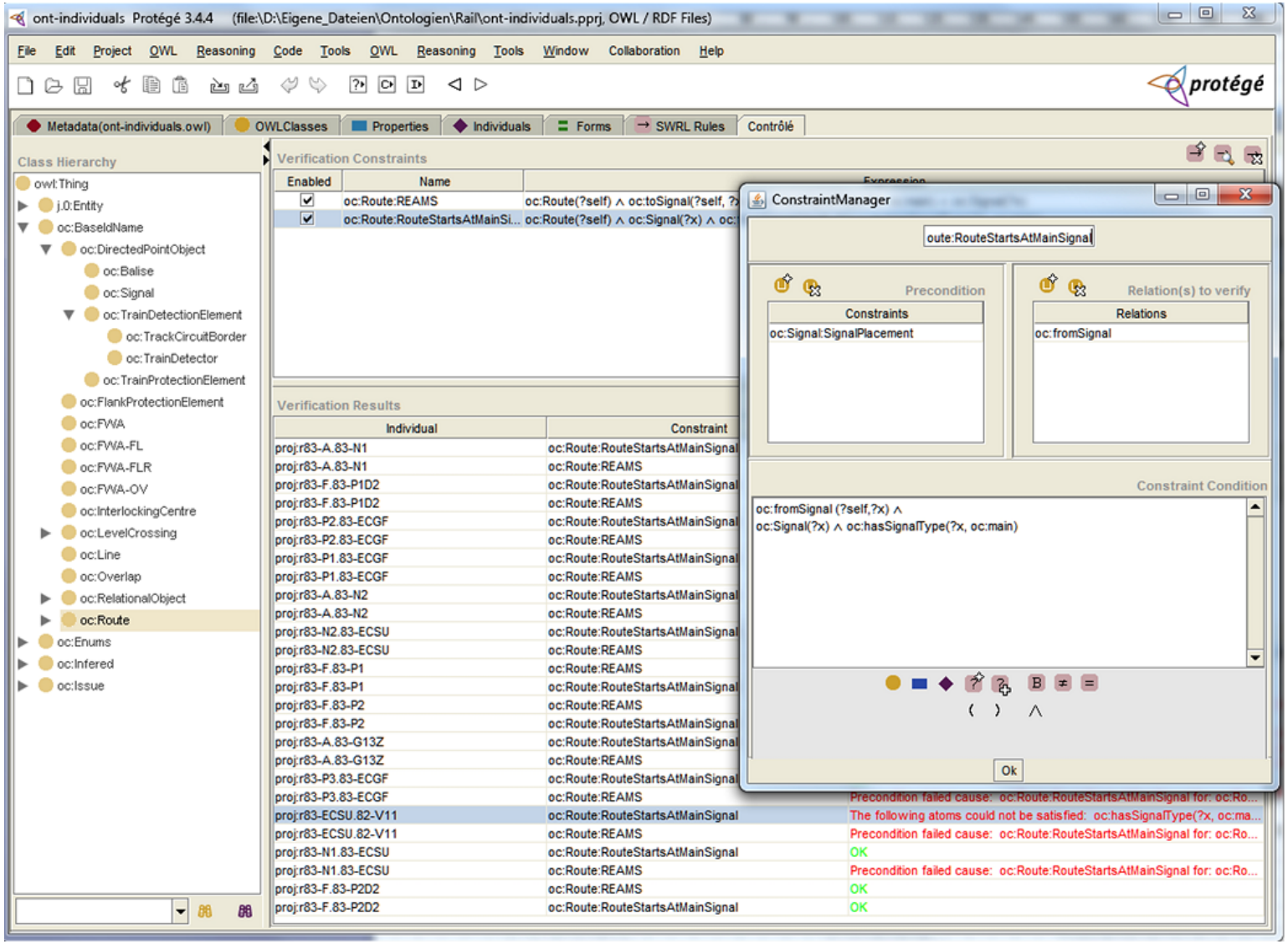

Abbildung 5.9: Screenshot des Contrôlé-Plugins

5.3.4 Software Architektur

Architektonisch nutzt Contrôlé die Pluginschnittstelle von Protégé. Hierbei ist Contrôlé als Erweiterung der von der Protégé-API vorgegebenen Klasse `AbstractTabWidget` implementiert. Abbildung 5.10 veranschaulicht die Softwarearchitektur von Contrôlé. Das Plugin ist in unterschiedliche Schichten gegliedert. Die Plugin-Schicht bildet die Schnittstelle zur Protégé-Plugin-API. Hier ist die Integration des Plugins als Reiter in die Protégé-Benutzeroberfläche implementiert. In der Business-Schicht ist die Logik für die Erzeugung der ontologischen Modelle und der Semantic Constraints implementiert sowie die Persistenzfunktionalität. Die View-Schicht beinhaltet die Klassen für die Darstellung wie beispielsweise die spezialisierte Klassenhierarchie, die Listen-Komponente der Semantic Constraints und den Resultatsbereich (vgl. Abbildung 5.8). In der Controller-Schicht sind die Klassen für die Verarbeitung der SC, die implementierten Algorithmen zur SC-Reduktion und die Anbindung des Reasoners realisiert. Von einer detaillierten Beschreibung der Implementierung der einzelnen Schichten und ihrer Klassen wird in der vorliegenden Arbeit abgesehen, stattdessen wird auf die Arbeit [36] verwiesen, in der diese Details hinreichend erläutert werden.

5.4 Abschließende Betrachtung

Das Konzept der Semantic Constraints sowie dessen Implementierung als Protégé-Plugin Contrôlé stellt eine Bereicherung für den Verifikationsprozess von EI dar. Mit Hilfe von SC können Fehler in der Wissensbasis gefunden und eingegrenzt werden. So wird der Verifikationsprozess effizienter gestaltet und beschleunigt. Auch die Robustheit des Prozesses ist von Vorteil. Es kann garantiert werden, dass die untersuchten Individuen definitiv den Vorgaben der SC entsprechen oder gegen sie verstoßen. Hierbei werden die Atome der SC identifiziert, die in diesen Verstoß involviert sind.

Die Verwendung von Vorbedingungen einzelner SC ermöglicht den Entwurf hierarchischer Strukturen. So können grundlegende SC definiert und in weiteren SC als Basis genutzt werden. Die so geschaffenen Strukturen weisen eine hohe Ausdrucksstärke und Komplexität auf, sind aber aufgrund ihres modularen Charakters gut nachvollziehbar.

Ein Problem bei der Verwendung von SC ist die mögliche hohe Anzahl an abgeleiteten Sub-SC. Es können theoretisch 2^n Sub-Constraints von einem Basis-Constraint abgeleitet werden, wobei n die Anzahl der Atome des Basis-Constraints beschreibt. Dieses Problem wurde erkannt und durch den Einsatz von Heuristiken minimiert. Die priorisierte Reduktion verschiedener Arten von Regel-Atomen, wie Klassen-Atomen,

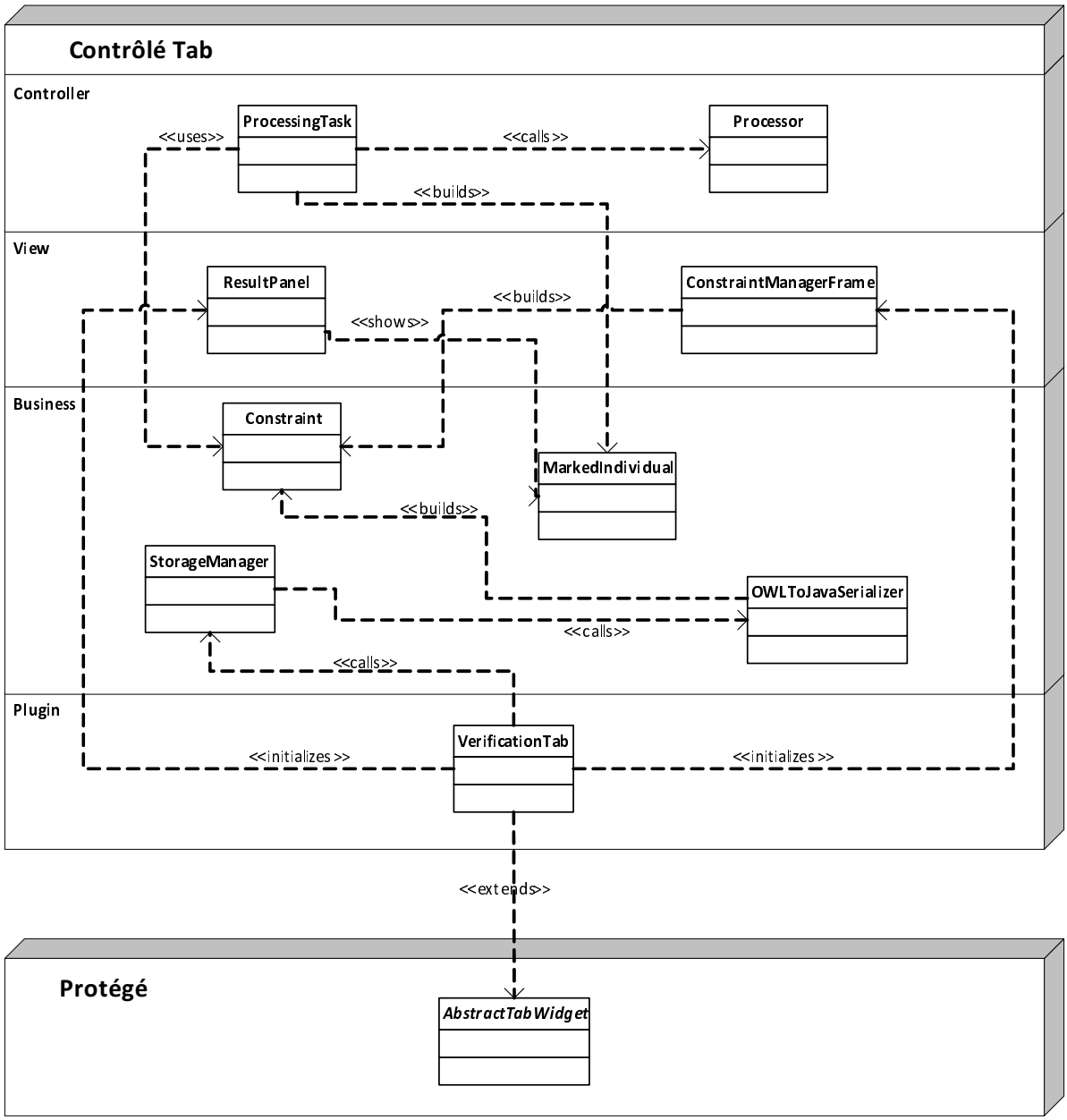

Abbildung 5.10: Software-Architektur des Contrôlé-Plugins

SWRL-Builtins usw. bewirkt, dass die Verarbeitung einzelner SC schneller abgeschlossen werden kann und somit zeitnah und ressourcensparend ein Ergebnis des Verifikations- und Debuggingprozesses vorliegt.

Die Entwicklung eines Protégé-Plugins stellt eine sinnvolle Erweiterung dar, um Datenverifikation zu ermöglichen und Fehler in der Modellierung von Ontologien im Allgemeinen einzugrenzen. Für den Verifikations- und Debuggingprozess von EI-Modellen bietet Protégé durch seine gut strukturierte Benutzeroberfläche, die Integration von SWRL mit Code-Vervollständigung sowie die Anbindung von Reasonern eine pragmatische und effiziente Möglichkeit, diese Prozesse zu beschleunigen.

6 Bewertung

6.1 Evaluation der vorgestellten Ansätze

Während der Testphase unseres Ansatzes zur Verifikation von EI zeigte sich, dass die beteiligten Eisenbahningenieure sehr schnell mit den semantischen Technologien vertraut wurden. Der Umstand, dass OWL-Ontologien auf RDF basieren und somit einer natürlichen Darstellungsweise von Subjekt-Prädikat-Objekt-Verkettungen folgen, ermöglichte dieses schnelle Erfassen der Konzepte und ihrer Ausdrucksfähigkeit. Dadurch dass die Ingenieure die EI-Planungsdaten im entsprechenden Editor modelliert, über die XML-Schnittstelle exportiert und dann den automatisierten Transformations- und Verifikationsprozess angestoßen haben, war der Wiedererkennungswert beim Studieren des Verifikationsergebnisses hoch. Hier wurden Modellierungsdetails unmittelbar erkannt, die präsentierten Fehler konnten nachvollzogen werden und wurden von den Ingenieuren als Modifikationsaufforderungen der entsprechenden Stellen im Planungsmodell verstanden.

Die Verfolgung einer strikten Trennung von Zuständigkeiten wurde als klar nachvollziehbar empfunden und ermöglicht ein einfaches Verfahren, die Wissensbasis in den unterschiedlichen Bereichen unabhängig voneinander zu erweitern. Die Aufteilung der Wissensbasis in eine EI-unabhängige Basisgraph-Ontologie (*RI* Graph Ontology*), eine Ontologie zur Repräsentation der Eisenbahndomäne (*RI* Core Ontology*), eine Ontologie, die sämtliche Planungsregeln und Verifikationsklassen beinhaltet (*RI* Rule Ontology*) sowie die ABox-Ontologie (*Specific RI Ontology*), die aus dem XML-Export des Planungs-Editors generiert wird und die zu verifizierenden Planungsdaten beinhaltet, verdeutlicht die unterschiedlichen Quellen einer solchen Wissensbasis und entspricht den inhomogenen Zuständigkeiten der Akteure, die an einem Planungsprozess von Eisenbahn-Infrastrukturen beteiligt sind. So ist die *RI* Graph Ontology* als unabhängiges Fundament zu interpretieren, das eine Hilfestellung für den gesamten Verifikationsprozess darstellt und als notwendiges Korsett für die höheren Schichten des ontologischen Modells zu verstehen ist. Im Gegensatz dazu setzen sowohl die *RI* Core Ontology* als auch die *RI* Rule Ontology* tiefes Wissen in der Eisenbahntechnik und in Normen und Richtlinien zur Planung von EI voraus. Die Daten für die

Specific RI Ontology hingegen stammen aus der Feder von Eisenbahningenieuren, die mit konkreten Planungsaufgaben betreut sind. Anwender des Verifikationssystems empfinden diese ontologische Struktur als natürlich und intuitiv benutzbar.

Ein weiterer Gegenstand unserer Untersuchungen war die Vergleichbarkeit des XML-Formats, welches von dem Planungseditor exportiert wird, und der daraus generierten *Specific RI Ontology*, die schließlich einer Verifikation durch unser System unterzogen wird. Diese Untersuchungen sind insofern notwendig, als dass im engeren Sinne das XML-Format, bzw. dessen Ausprägung im Planungseditor verifiziert werden soll, da die Planung grundsätzlich mithilfe dieses Editors erfolgt und somit auch eine Modifikation der Planungsdaten lediglich im Editor vorgenommen wird. Ein systematischer Vergleich des XML- und des OWL-Formats wurde in enger Zusammenarbeit mit Ingenieuren der Eisenbahntechnik durchgeführt. Dabei wurde untersucht, ob sämtliche Elemente mit sämtlichen Attributen der XML-Repräsentation ebenso als Klassen und *properties* in der OWL-Repräsentation wiederzufinden sind. Auch die Abhängigkeitsbeziehungen der XML-Elemente müssen als Relationen in entsprechenden *object properties* in OWL präsent sein. Im Zuge der Untersuchungen hat sich gezeigt, dass die wesentlichen Aspekte der XML-Repräsentation in OWL abgebildet sind. Da das XML-Format jedoch über die Abbildung von Regionalstellwerken hinausgeht, sind die entsprechenden Elemente nicht in OWL repräsentiert. Diesem Umstand ist es geschuldet, dass in der Proof-of-Concept-Umsetzung der vorliegenden Arbeit lediglich die Planungsdaten elektronischer Regionalstellwerke verifiziert wurden. Ebenfalls geht das XML-Format über eine höhere Abstraktionsebene hinaus, bzw. ermöglicht ebenso die Abbildung tieferer technischer Details und anderer Aspekte des Planungsprozesses, die in der gegenwärtigen OWL-Repräsentation zur Planung und Verifikation von EI (noch) nicht modelliert sind. So ist es beispielsweise möglich, die Bedienoberfläche der Stellwerke in das XML-Format zu exportieren. Hierbei werden insbesondere die Anordnungen der graphischen Repräsentationen von EI-Elementen auf einem Computerbildschirm behandelt. Auch diese Daten sind nicht Gegenstand des Verifikationssystems und haben somit keine Repräsentation im OWL-Format dieses Systems.

Neben der Benutzerakzeptanz des Verifikationssystems und seiner Komponenten wurde ebenso die Umfang der in SWRL formalisierten Planungsregel untersucht. Grundsätzlich ist das System wie schon beschrieben als Proof-of-Concept anzusehen. Die Intention dieser Arbeit ist, aufzuzeigen, wie ein Regelwerk der Deutschen Bahn zur Planung von EI abgebildet werden kann. Wie unter 2.1.5 beschrieben, ist dieses Regelwerk in eine Vielzahl unterschiedlicher Planungsrichtlinien (Ril 819 ff.) unterteilt, um den gesamten Planungsprozess abzubilden. Das in der vorliegenden Arbeit entworfene System ist auf einer höheren Schicht des Planungsprozesses angesiedelt. Grundlegende physikalische Eigenschaften der verwendeten Materialien, aber auch beispielsweise Kurvenradien der Gleisformationen oder Verkabelungen der einzelnen elektronischen

Elemente wurden nicht im Verifikationssystem modelliert. Diese Arbeit beschränkt sich darauf, die wechselseitigen Relationen der Elemente der Leit- und Sicherungstechnik abzubilden. Hierbei wurde der Fokus besonders auf die Stellwerkssicht gesetzt, da dies einem pragmatischen Planungsprozess der oberen Schichten der Leit- und Sicherheitstechnik entspricht. Bezogen auf den Umfang der formalisierten Planungsregeln wurden ca. 15% der entsprechenden Richtlinien der Leit- und Sicherheitstechnik aus den Ril 819.02, 819.04, 819.05 und 819.13 (vgl. Auflistung in Kapitel 2.1.5) modelliert. Dieser verhältnismäßig geringe Umfang ermöglicht jedoch die Verifikation von üblichen Zusammenhängen der Elemente der Leit- und Sicherheitstechnik in elektronischen Regionalstellwerken.

SWRL eignet sich sehr gut für die Formalisierung der Planungsrichtlinien. Die Regelsprache ermöglicht über die intuitive, implikations-basierte Syntax die Formulierung von *Wenn-Dann*-Bedingungen. Die Verwendung von SWRL-Builtins verleiht dieser Sprache eine notwendige Mächtigkeit und genügt damit den Anforderungen an die Modellierung von Planungsrichtlinien. So können mithilfe der Builtins mathematische Berechnungen, Vergleiche oder auch Zeichenkettenoperationen durchgeführt werden, die für die formale Abbildung der Planungsrichtlinien in entsprechenden SWRL-Regeln zwingend erforderlich sind.

Bezüglich der Performance bei der Verwendung von SC wurden verschiedene Szenarien analysiert. Das grundsätzliche Testszenario setzt sich zusammen aus der TBox der *RI* Ontology*-Sammlung und einer *Specific RI Ontology*, die aus den Planungsdaten eines mittelgroßen Bahnhofs besteht. Mit diesem Testaufbau wurde die Verarbeitung von unterschiedlichen Anzahlen von SCs unterschiedlichen Umfangs getestet. Der Vergleich der beiden Testszenarien ist in der Tabelle 6.1 dargestellt.

Anzahl SCs	**max. Atome pro SC**	**Verarbeitungsdauer**
500	10	15 Min. - 2 Std.
8	20	2 Std. - 4 Std.

Tabelle 6.1: Vergleich der Verarbeitungsgeschwindigkeit von Semantic Constraints

Es zeigt sich, dass die Verarbeitungsgeschwindigkeit zum einen von der Komplexität der Struktur von SCs abhängt. So sind SCs mit einer hohen Anzahl von Atomen wesentlich langsamer in der Verarbeitung als SCs mit einer geringen Anzahl von Atomen. Zum anderen ist die Verarbeitungsgeschwindigkeit von den internen Algorithmen von Protégé 3.x und der Jess Rule Engine abhängig. Insbesondere die interne Transformation des Protégé-Editors von OWL-Axiomen und SWRL-Atomen in die Repräsentationsform der Jess Rule Engine ist zeitintensiv. Die hohe Fluktuation der Verarbeitungsgeschwindigkeit hat ihre Ursache in der unterschiedlichen Form der Speicherung der Ontologie-Fakten in den entsprechenden Dateien. OWL und SWRL

sind zwar deklarative Sprachen, jedoch werden die Modelle, bedingt durch die allgemeine Computerarchitektur, intern sequentiell verarbeitet. Somit ist bei semantisch identischen Modellen, die in gleichwertigen aber syntaktisch unterschiedlichen Strukturen gespeichert sind, eine abweichende Verarbeitungsgeschwindigkeit zu beobachten.

Die Möglichkeit der Formulierung von Vorbedingungen zu SCs befähigt den Anwender dazu, modulare Strukturen aufzubauen und somit die Anzahl der Atome pro SC gering zu halten. Dies trägt nicht nur zu einer Verbesserung der Verständlichkeit durch eine verminderte Komplexität bei, sondern erhöht auch die interne Verarbeitungsgeschwindigkeit der SCs.

7 Ausblick

Im Rahmen dieser Arbeit wurde ein Ansatz konzeptioniert und umgesetzt, der die Verwendung semantischer Technologien für die Verifikation von Eisenbahn-Infrastrukturen ermöglicht. Hierzu wurde eine mehrschichtiges Modell entworfen, das sowohl die Facetten der einzelnen Elemente von EI als auch Richtlinien des Planungsprozesses abbildet und in Beziehung zueinander setzt. Zukünftige Tätigkeiten bezüglich dieses Teils der Arbeit sind insbesondere in der Vervollkommnung des Modells wahrzunehmen. So ist eine Erweiterung des semantischen Domänenmodells um zusätzliche Klassen und Eigenschaften, die weitere EI-Elemente beschreiben, eine mögliche Tätigkeit. Ferner ist es zielführend, weitere Planungsrichtlinien formal in SWRL zu modellieren und diese in verschiedenen Planungsmodulen zu gliedern. So können unterschiedliche Aspekte des Planungsprozesses gezielt für Verifikationszwecke genutzt werden. Beispielsweise können auf diese Weise lediglich einzelne Bahnhöfe, nur die Weichen- oder Signalsysteme oder Ähnliches überprüft werden.

Die vorliegende Arbeit befasst sich mit der Verifikation von Regionalstellwerken. Um die Verifikation vollwertiger Stellwerke zu ermöglichen, sind beispielsweise Elemente für die Modellierung und Überprüfung von Rangierfahrstraßen erforderlich. Hierzu sind, ebenso wie die Fahrstraßenelemente des Modells, die entsprechenden Signalbegriffe zu erweitern. Analog müssen die korrespondierenden Planungsrichtlinien identifiziert und formalisiert werden.

Das Konzept der Semantic Constraints stellt ein Verfahren dar, um den Eisenbahn-Ingenieur bei der Suche nach Planungsfehlern zu unterstützen. Eine mögliche Verbesserung dieses Verfahrens liegt in der weiteren Verfeinerung des Reduktions-Algorithmus'. Diese stellt sich dadurch dar, dass die zu reduzierenden Atome der entsprechenden Regeln gezielter identifiziert werden. So ist es denkbar, Korrelationen zwischen den einzelnen Regelatomen über Abhängigkeitsanalysen gezielter zu bestimmen. Hierzu sind jedoch umfangreiche Tests mit praxisrelevanten Planungsdaten vonnöten. Von der technischen Seite betrachtet, ist eine Steigerung der Performance durch die Verwendung einer aktuellen Version der OWL-API und der direkten Anbindung des Reasoners ohne den Umweg über die Jess Rule Engine möglich.

Generell ist es wünschenswert, eine detaillierte Bewertung der Verifikationsergebnisse

in weiterer Zusammenarbeit mit Domänenexperten und Planungsbeauftragten von EI durchzuführen. Hierzu ist eine entsprechende Ressourcenplanung seitens involvierter EI-Planungsunternehmen erforderlich.

Zusammenfassend ist anzumerken, dass die Unterstützung des EI-Planungsprozesses mithilfe automatisierter Prüfverfahren in Zukunft an Bedeutung gewinnen wird. Der in der vorliegenden Arbeit vorgestellte Ansatz der Verwendung semantischer Technologien für Verifikationszwecke ist in vielfacher Hinsicht eine Bereicherung und zugleich ein Novum. Er eröffnet diesen Technologien ein anwendungsbezogenes Einsatzfeld, welches über das ursprüngliche Kerngebiet hinausgeht.

Literaturverzeichnis

[1] *Pellet: The Open Source OWL Reasoner*. http://clarkparsia.com/pellet/, 2009

[2] ABeele, Van deb: System Requirement Specification For InteGRail system / ALSTOM. 2007. – Forschungsbericht

[3] AG, Deutsche B.: *Deutsche Bahn AG: Richtlinie 819 - 'LST Anlagen planen'*. 1998

[4] al., Jennifer V.: *Protégé Plugin Library*. `http://protegewiki.stanford.edu/wiki/Protege_Plugin_Library`. Version: 03 2013

[5] Ambrosi, Cristina D. ; Ghersi, Cristiano ; Tacchella, Armando: An Ontology-Based Condition Analyzer for Fault Classification on Railway Vehicles. In: *IEA/AIE*, 2009, S. 449–458

[6] Baader, Franz (Hrsg.) ; Alvanese, CDiego (Hrsg.) ; McGuinness, Deborah L. (Hrsg.) ; Nardi, Daniele (Hrsg.) ; Patel-Schneider, Peter F. (Hrsg.): *The Description Logic Handbook: Theory, Implementation, and Applications*. New York, NY, USA : Cambridge University Press, 2003. – ISBN 0–521–78176–0

[7] Baader, Franz ; Sattler, Ulrike: Tableau Algorithms for Description Logics. In: *Studia Logica* 69 (2000), S. 2001

[8] Beech, David ; Mendelsohn, Noah ; Maloney, Murray ; Thompson, Henry: XML Schema Part 1: Structures Second Edition / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[9] Berners-Lee, Sir T. ; Hendler, James ; Lassila, Ora: The Semantic Web. In: *Scientific American Magazine* (2001)

[10] Boley, Harold: The RuleML Family of Web Rule Languages. In: Alferes, José J. (Hrsg.) ; Bailey, James (Hrsg.) ; May, Wolfgang (Hrsg.) ; Schwertel, Uta (Hrsg.): *PPSWR* Bd. 4187, Springer, 2006 (Lecture Notes in Computer

Science), 1-17

[11] Brickley, Dan ; Miller, Libby: FOAF Vocabulary Specification 0.97. Version: January 2010. http://xmlns.com/foaf/spec/20100101.html. 2010. – Namespace document

[12] Bundesamt, Statistisches: *Verkehrsmittelbestand und Infrastruktur*. https://www.destatis.de/DE/ZahlenFakten/Wirtschaftsbereiche/TransportVerkehr/UnternehmenInfrastrukturFahrzeugbestand/Tabellen/Schieneninfrastruktur.html. Version: 12 2010

[13] Clark, James: XSL Transformations (XSLT) Version 1.0 / W3C. 1999. – W3C Recommendation. – http://www.w3.org/TR/1999/REC-xslt-19991116

[14] Ferstl, Otto K. ; Sinz, Elmar J.: *Grundlagen der Wirtschaftsinformatik*. 5, überarb. u. erw. A. Oldenbourg, 2006. – ISBN 3486579428

[15] Forgy, Charles: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In: *Artificial Intelligences* 19 (1982), Nr. 1, 17-37. http://dx.doi.org/10.1016/0004-3702(82)90020-0. – ISSN 0004–3702

[16] Foundation, Apache S.: *Apache Software Foundation - Xerces*. http://xerces.apache.org/. Version: 1999

[17] Foundation, The E.: *Eclipse integrated development environment*. http://www.eclipse.org. Version: 2008. – http://www.eclipse.org

[18] Groups, W3C XSL/XML Query W.: *The XPath 2.0 Standard*. Version: 2007. http://www.network-theory.co.uk/w3c/xpath/

[19] Hayes, Patrick: *RDF Semantics*. W3C Recommendation, 2004

[20] Horridge, Matthew ; Knublauch, Holger ; Rector, Alan ; Stevens, Robert ; Wroe, Chris: *A Practical Guide To Building OWL Ontologies With The Protege-OWL Plugin*. 1. University of Manchester, 2004. http://home.skku.edu/~samoh/class/sw/ProtegeOWLTutorial.pdf

[21] Horridge, Matthew ; Patel-Schneider, Peter: OWL 2 Web Ontology Language Manchester Syntax (Second Edition) / W3C. 2012. – W3C Note. – http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/

[22] Horrocks, Ian ; Harmelen, Frank van ; Patel-Schneider, Peter F. ;

Berners-Lee, Sir T. ; Brickley, Dan ; Connoly, Dan ; Dean, Mike ; Decker, Stefan ; Fensel, Dieter ; Hayes, Pat ; Heflin, Jeff ; Hendler, Jim ; Lassila, Ora ; McGuinness, Deborah ; Stein, Lynn A.: *DAML+OIL*. http://www.daml.org/2001/03/daml+oil-index.html, March 2001

[23] Horrocks, Ian ; Patel-schneider, Peter F. ; Bechhofer, Sean ; Tsarkov, Dmitry: OWL Rules: A Proposal and Prototype Implementation. In: *Journal of Web Semantics* 3 (2005), S. 23–40

[24] Horrocks, Ian ; Patel-Schneider, Peter F. ; Boley, Harold ; Tabet, Said ; Grosof, Benjamin ; Bean, Mike: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C - Submission, May 2004. – http://www.w3.org/Submission/SWRL

[25] Jelliffe, Rick: *Academia Sinica Computing Centre's Schematron Home Page*. http://xml.ascc.net/resource/schematron/. Version: 2001

[26] Kay, Michael: XSLT and XPath Optimization. In: *XML Europe*, 2004

[27] Knublauch, Holger: SPIN - Modeling Vocabulary / W3C. 2010. – W3C Member Submission. – http://www.w3.org/Submission/spin-modeling/

[28] Knublauch, Holger ; Horridge, Matthew: *The Protégé-OWL API*. http://protege.stanford.edu/plugins/owl/api/. Version: 2005

[29] Kommission, Europäische: *Mitteilung der Kommission an das Europäische Parlament und den Rat über die Einführung des europäischen Zugsicherungs-/Zugsteuerungs- und Signalgebungssystems ERTMS/ETCS [SEC(2005) 903]*. 07 2005

[30] Kuba, Martin: *OWL 2 and SWRL Tutorial*. http://dior.ics.muni.cz/~makub/owl/#keys. Version: 2012

[31] Lodemann, Michael ; Luttenberger, Norbert: Beschreibung von Eisenbahninfrastrukturen mit railML und ihre Verifikation. In: *Signal + Draht* 102 (2010), S. 37–42

[32] Lodemann, Michael ; Luttenberger, Norbert: Ontology-based Railway Infrastructure Verification - Planning Benefits. In: *KMIS*, 2010, S. 176–181

[33] Lodemann, Michael ; Luttenberger, Norbert ; Schulz, Elferik: Semantic Computing for Railway Infrastructure Verification. In: *IEEE ICSC Proceedings*

- *Computer Society Press*, 2013

[34] Lodemann, Michael ; Marnau, Rita ; Luttenberger, Norbert: Using Semantic Constraints for Verification in an Open World. In: *Knowledge Technology. Springer*, 2012, S. 206–215

[35] Luethi, Marco ; Huerlimann, Daniel ; Nash, Andrew: Understanding the Timetable Planning Process as a Closed Control Loop. In: *Institute for Transport Planning and Systems, ETH Zurich* (2005). http://dx.doi.org/http://dx.doi.org/10.3929/ethz-a-005704049. – DOI http://dx.doi.org/10.3929/ethz–a–005704049

[36] Marnau, Rita: *Benutzerdefinierte Constaints für ontologische Wissensbasen - ihre Formulierung und die Diagnose von Verletzungen*, Christian-Albrechts-Universität zu Kiel, Diplomarbeit, 2010

[37] Metzger, Robert C.: *Debugging by Thinking: A Multidisciplinary Approach.* Digital Press, 2004 (HP Technologies Series). http://books.google.de/books?id=v9bH-7GoD74C. – ISBN 9781555583071

[38] Mohan, A. R. ; G.Arumugam: Constructing Railway Ontology using Web Ontology Language and Semantic Web Rule Language. In: *International Journal of Computer Technology and Applications* (2005)

[39] Nash, Andrew ; Huerlimann, Daniel ; Schuette, Jörg ; Krauss, Vasco P.: RailML - a standard data interface for railroad applications. In: *Proc. of the Ninth International Conference on Computer in Railways (Comprail IX)* 15 (2004), Nr. 5, S. 233–240

[40] O'connor, Martin ; Knublauch, Holger ; Tu, Samson ; Musen, Mark: Writing Rules for the Semantic Web Using SWRL and Jess. In: *8th International Protege Conference, Protege with Rules Workshop*, 2005

[41] omg: *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[42] Paoli, Jean ; Sperberg-McQueen, Michael ; Maler, Eve ; Yergeau, François ; Bray, Tim: Extensible Markup Language (XML) 1.0 (Third Edition) / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-xml-20040204

[43] Parsia, Bijan ; Motik, Boris ; Patel-Schneider, Peter: OWL 2

Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition) / W3C. 2012. – W3C Recommendation. – http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

[44] Parsia, Bijan ; Sirin, Evren ; Kalyanpur, Aditya: Debugging OWL ontologies. In: *Proc. of the 14th World Wide Web Conference (WWW2005)*. Chiba, Japan, Mai 2005

[45] Patel-Schneider, Peter ; Parsia, Bijan ; Motik, Boris: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax / W3C. 2009. – W3C Recommendation. – http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/

[46] Patel-Schneider, Peter ; Parsia, Bijan ; Motik, Boris: OWL 2 Web Ontology Language XML Serialization / W3C. 2009. – W3C Recommendation. – http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/

[47] Patel-Schneider, Peter F. ; Horrocks, Ian: *A comparison of two modelling paradigms in the Semantic Web*. 2007

[48] Prud'hommeaux, Eric ; Seaborne, Andy: SPARQL Query Language for RDF / W3C. 2008. – W3C Recommendation. – http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

[49] Steele, Guy L.: *Common LISP: the language (2nd ed.)*. Newton, MA, USA : Digital Press, 1990. – ISBN 1–55558–041–6

[50] Stuckenschmidt, Heiner: Debugging OWL Ontologies - A Reality Check. In: Garcia-Castro, Raul (Hrsg.) ; Gómez-Pérez, Asunción (Hrsg.) ; Petrie, Charles J. (Hrsg.) ; Valle, Emanuele D. (Hrsg.) ; Küster, Ulrich (Hrsg.) ; Zaremba, Michal (Hrsg.) ; Shafiq, M. O. (Hrsg.): *EON* Bd. 359, CEUR-WS.org, 2008 (CEUR Workshop Proceedings)

[51] Tao, Jiao: Adding Integrity Constraints to the Semantic Web for Instance Data Evaluation. In: *9th International Semantic Web Conference (ISWC2010)*, 2010

[52] Tao, Jiao ; Sirin, Evren ; Bao, Jie ; McGuinness, Deborah L.: Integrity Constraints in OWL. In: *AAAI*, 2010

[53] TopQuadrant: *TopQuadrant | Products | TopBraid Composer*. `http://www.topquadrant.com/products/TB_Composer.html`. Version: 2011, Abruf: 02-03-2011

[54] Verkehrssicherung GmbH, IVV I. u.: *ProSig - Planungssoftware für Bahnanlagen*. Innotrans, 09 2012

[55] Vlist, Eric van d.: *Schematron*. First. O'Reilly, 2007. – ISBN 9780596527716

[56] W3C: *Qualified cardinality restrictions (QCRs): Constraining the number of values of a particular type for a property*. http://www.w3.org/2001/sw/BestPractices/OEP/QCR/. Version: 2005

[57] W3C: *OWL 2 Web Ontology Language New Features and Rationale (Second Edition)*. W3C Working Draft. http://www.w3.org/TR/owl2-new-features/. Version: December 2012

[58] Wang, Hai H. ; Noy, Natasha ; Rector, Alan ; Musen, Mark ; Redmond, Timothy ; Rubin, Daniel ; Tu, Samson ; Tudorache, Tania ; Drummond, Nick ; Horridge, Matthew ; Seidenberg, Julian: Frames and OWL Side by Side. In: *9th International Protégé Conference*, 2006

[59] Wunsch, Susanne: *Verifizierung von railML-Daten mithilfe von Schematron*, Teschnische Universität Dresden, Diplomarbeit, 2010

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

ABox ..... Assertional Box - Die Abox beinhaltet in einer Wissenbasis, üblicherweise gespeichert in Ontologien, die Angaben über konkrete Entitäten und Instanzen von Konzepten der TBox. Verglichen mit der objektorientierten Programmierung stellt die ABox die Klasseninstanzen dar. Zusammen mit der TBox bildet die Abox die gesamte Wissensbasis.

API ..... Application Programming Interface - Programmierschnittstelle, die Standardklassen und -funktionen zur allgemeinen Verwendung freigibt. Eine API wird dazu verwendet, Ansatzpunkte zur Funktionalitätserweiterung von Programmen oder die Entwicklung von entsprechenden Plugins zu ermöglichen.

BDZ ..... Bedienzentrale - Einrichtung zur zentralen Überwachung und Steuerung der Leit und Sicherheitstechnik von Eisenbahn-Infrastrukturen.

CWA ..... Closed World Assumption - Modellierungsparadigma, bei dem nur explizit definierte Fakten berücksichtigt werden. Eine automatische Negation eines Fakts durch sein Nicht-Vorhandensein im Modell entfällt bei diesem Paradigma.

DLR ..... Deutsches Zentrum für Luft- und Raumfahrt

DPE ..... OWL data property expression - Ein OWL-Axiom welches die Relation einer Klasse zu einem Datentyp ausdrückt.

DRY ..... Don't Repeat Yourself - Entwicklungsprinzip zur Vermeidung von Redundanzen.

DTD ..... Document Type Definition - Ein Verfahren zur Überprüfung der Struktur von XML-Dokumenten.

EBA ..... Eisenbahnbundesamt - Deutsche Behörde, die unter anderem die Planung neuer Eisenbahnstrecken zertifiziert.

EI ..... Eisenbahn-Infrastruktur - Hierzu zählen alle stationären Elemen-

te eines Schienennetzes wie z.B. Gleise, Weichen, Signale, Bahnübergänge, etc. Nicht zu EI zählen beispielsweise Züge. oder der Fahrplan etc.

ESTW ..... Elektronisches Stellwerk - Moderner Typ einer Schaltzentrale, mit der Zugbewegungen überwacht und gesteuert werden. Wird in der BDZ bedient.

ESTW-R ..... Elektronisches Regionalstellwerk - Moderner Typ einer Schaltzentrale, mit der Zugbewegungen des Regionalverkehrs überwacht und gesteuert werden. Wird in der BDZ bedient.

FMA ..... Freimeldeabschnitt - Bereich einer Eisenbahn-Infrastruktur, welcher von jeweils nur einem Zug zurzeit beansprucht werden kann. Verfügt über entsprechende Zugdetektionseinrichtungen.

GUI ..... Graphical User Interface - Graphische Oberfläche, die auf einem Bildschirm dargestellt wird und eine Interaktion des Benutzers mit einer (Software-)Anwendung zur Aufgabe hat.

IEEE ..... Institute of Electrical and Electronical Engineers - US-amerikanisches Normierungsinstitut

KI ..... Künstliche Intelligenz - Die künstliche Intelligenz ist ein Fachgebiet der Informatik und hat zum Ziel, Maschinen und Computersysteme zu entwickeln, die aufgrund von Informationseingabedaten und deren Verarbeitung 'intelligente' Ausgabereaktionen produzieren können. Generalisierungsfähigkeiten sowie Lernverhalten in Computersystemen zu implementieren, ist ein Kerngebiet der KI, welches durch andere Fachrichtungen wie Wissensrepräsentation und Kognition angeregt wird.

NaF ..... Negation as Failure - In der CWA wird fehlendes Wissen als trivial falsch angesehen. Dies wird als NaF bezeichnet. Dieses Paradigma gilt jedoch nicht für die OWA.

OPE ..... OWL object property expression - Ein OWL-Axiom, welches die Relation von Klassen untereinander beschreibt.

OWA ..... Open World Assumption - Modellierungsparadigma, bei dem fehlende Fakten eine Negation zur Folge haben. Grundlegendes Paradigma von Datenbanksystemen oder klassische Programmiersprachen wie Jave, C etc.

OWL ..... Web Ontology Language - Sprache zur Modellierung von Wissens-

basen - sogenannten Ontologien. Nutzt die OWA als grundlegendes Modellierungsparadigma.

PZB ..... Punktförmige Zugbeeinflussung - Eisenbahn-Infrastrukturelement zur Zwangsbremsung von Zügen. Wird umgangssprachlich auch als Gleismagnet bezeichnet.

SC ..... Semantic Constraint - Auf SWRL basierende Definition einer Anforderung an eine ontologische Wissensbasis zur Debugging-Unterstützung.

SSC ..... Sub Semantic Constraint - bezeichnet einen vom ursprünglichen SC um ein oder mehrere Atome reduzierten SC.

SWRL ..... Semantic Web Rule Language - Eine auf OWL und ruleML basierende Sprache zur Formulierung von Regeln in einer ontologischen Wissenbasis. Die Regeln folgen einem Implikationsschema (Antecendent -> Consequent) und versetzen einen Benutzer bspw. in die Lage, mathematische Ausdrücke zu formulieren. SWRL Regeln können durch einen Reasoner oder im Falle des Protégé Editors 3.x durch die Jess Rule Engine ausgewertet werden.

TBox ..... Terminological Box - Die Tbox beinhaltet das terminologische Wissen einer Wissenbasis, üblicherweise gespeichert in Ontologien. Gemeint sind damit alle Aussagen bezüglich der Konzepte, ihrer Attribute und Relationen. Verglichen mit der objektorientierten Programmierung stellt die TBox die Klassendefinitionen dar. Zusammen mit der ABox bildet die TBox die gesamte Wissensbasis.

UNA ..... Unique Name Assumption - Modellierungsparadigma, bei dem jedes Element über einen Bezeichner eindeutig identifiziert werden kann.

XML ..... Extensible Markup Language - Textbasierte Notation zur Strukturierung von Daten mithilfe von Tags.

XSLT ..... Extensible Stylesheet Language Transformation - XML-basierte Sprache zur Definition von Transformationsanweisungen für (XML)-Dokumente.

Hiermit erkläre ich, Michael Lodemann, an Eides statt, dass die vorliegende Dissertation - abgesehen von der Beratung durch meine wissenschaftlichen Lehrer und der Verwendung der angegebenen Hilfsmittel - nach Inhalt und Form meine Eigene ist.

Ferner erkläre ich, dass die Arbeit an keiner anderen Stelle im Rahmen eines Prüfungsverfahrens vorgelegen hat, veröffentlicht worden ist oder zur Veröffentlichung eingereicht wurde.

Die Arbeit ist unter Einhaltung der Regeln guter wissenschaftlicher Praxis der Deutschen Forschungsgemeinschaft entstanden.

Kiel, den 27. Januar 2015

(Michael Lodemann)