Daniel Sebauer, BSc.

Berechtigungsverifizierung mittels automatischer Strukturerkennung

MASTERARBEIT

zur Erlangung des akademischen Grades eines

Diplom-Ingenieur

Informatik

Graz University of Technology

Technische Universität Graz

Betreuer:
Univ.-Prof. Dipl.-Ing. Dr. techn. Franz WOTAWA
Institut für Softwaretechnologie

Graz, im Jänner 2012

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……

date (signature)

Abstract

Every employee has several permissions on the company's IT systems. These permissions are documented in several documents and standards. In most cases, it is really tough to monitor, maintain and verify these permissions. Automatic structural pattern recognition with a permission matching mechanism is developed for this purpose. A comparison between a system to be checked and a given template structure can be performed. In this thesis, automatic structural pattern recognition is performed on a "Microsoft Fileserver" and a "Microsoft SharePoint Server". After this process, all permissions of a single node can be compared with a node's permission template. The unification process between the system node and the reference node is only performed on a syntactical level. This process can be fulfilled in a direct or indirect way. For an indirect unification, the caption of a system node has to match a pattern given by a regular expression from a template node. The automatic structural pattern recognition with permission verification from this thesis can be ported to several systems where data is based on a tree structure.

Zusammenfassung

Jeder Mitarbeiter einer Firma hat in seinem Unternehmen verschiedene Berechtigungen. Diese sind in diversen Dokumentationen bzw. definierten Standards festgelegt. Die Kontrolle dieser Berechtigungen fällt in den meisten Fällen schwer, da die Wartung und Verifizierung diverser Berechtigungen nicht trivial lösbar sind. Für dieses Szenario wurde eine automatische Strukturerkennung entwickelt, die in weiterer Folge auch die Berechtigungen in diesen Systemen mit einer vordefinierten Vorlage abgleicht. Am Beispiel eines Microsoft Dateiservers und des Microsoft SharePoints wird eine automatisierte Strukturerkennung durchgeführt. Wenn dieser Vorgang abgeschlossen ist, können die jeweiligen Berechtigungen auf den ermittelten Knoten mit einem Template abgeglichen werden. Der Unifizierungsprozess zwischen einem System- und einem Referenzknoten aus dem Template erfolgt ausschließlich auf syntaktischer Ebene. Bei dieser Unifizierung wird zwischen direkten und indirekten Unifizierungen unterschieden. Bei indirekten Unifizierungen muss der Knotenname einem bestimmten Muster entsprechen, das in Form eines regulären Ausdrucks in den Musterknoten hinterlegt ist. Die hier vorgestellte Methodik einer automatischen Strukturerkennung mit Berechtigungsverifizierung lässt sich auf verschiedenste System übertragen, die durch eine Baumstruktur beschrieben werden.

Inhaltsverzeichnis

Abbildungen

Tabellen

Definitionen

1 Einleitung

Jeder Mitarbeiter hat in einem Betrieb verschiedene Berechtigungen auf den verschiedensten Systemen. In den meisten Fällen wurde zu Beginn dokumentiert, welche Mitarbeitergruppen bzw. Abteilungen auf bestimmte Ressourcen Zugriff haben sollen. Das Problem ist aber, dass man im Laufe der Zeit von diesem definierten Standard abweicht. In gewissen Situationen muss man einfach die Spezifikation durch zusätzliche Berechtigungen erweitern bzw. gegebenenfalls sogar einschränken. Das führt später zu einem enormen Chaos in der Berechtigungsstruktur, da man nicht jede Abweichung mitprotokollieren kann. In den meisten Fällen hat man aber keine Möglichkeit zu bestimmen, welche Dimensionen die Abweichungen vom definierten Standard in der Realität dann wirklich angenommen haben. Deswegen ist eine umfassende Berechtigungskontrolle der einzelnen Mitarbeiter unter anderem in den folgenden beiden Systemen erforderlich:

- Microsoft Dateiserver
- Microsoft Office SharePoint

1.1 Anwendungsszenarien

Jeder einzelne Benutzer bzw. jede Benutzergruppe hat bestimmte Berechtigungen auf einem System und diese müssen dann mit dem jeweiligen dokumentierten Standard verglichen werden. Als kurzes Beispiel am Dateiserver sieht das folgendermaßen aus:

Ein Benutzer bzw. eine Benutzergruppe hat bestimmte Berechtigungen auf der gesamten Dateistruktur. Man hat hier leider keinerlei Möglichkeiten schnell und effizient abzufragen, auf welche Ordner eine konkrete Person Zugriff hat. Deswegen stellt sich hier auch die Frage: Hat die Person wirklich nur Berechtigungen auf Ordner, die in der Definition festgelegt wurden oder weichen sie vom dokumentierten Standard ab?

Problematisch hierbei ist die Tatsache, dass man den aktuellen Dateisystembaum mit dem gewünschten Dateisystembaum abgleichen muss, da diese nicht deckungsgleich sind. Wenn dies nicht geschieht, können die Abweichungen nicht ermittelt werden. Es ist dann nicht feststellbar, welcher Ordner des aktuellen Standes mit welchem Knoten der Musterstruktur übereinstimmen soll. Hierfür benötigt man einen Strukturvergleich, um den aktuellen Ordnerbaum am Dateiserver inklusive Berechtigungsstruktur mit dem definierten Standard zu vergleichen.

Der entwickelte Ansatz sollte in weiterer Folge möglichst universell ausgearbeitet werden, um ihn beispielsweise auch für die Überprüfung von Strukturen bei elektronischen

Zutrittskarten für Gebäude- bzw. Raumstrukturen einzusetzen. Man kann die hierarchische Struktur eines Gebäudes durchaus mit der eines Baumes, der eine Ordnerstruktur abbildet, vergleichen, wie Tabelle 1 zeigt.

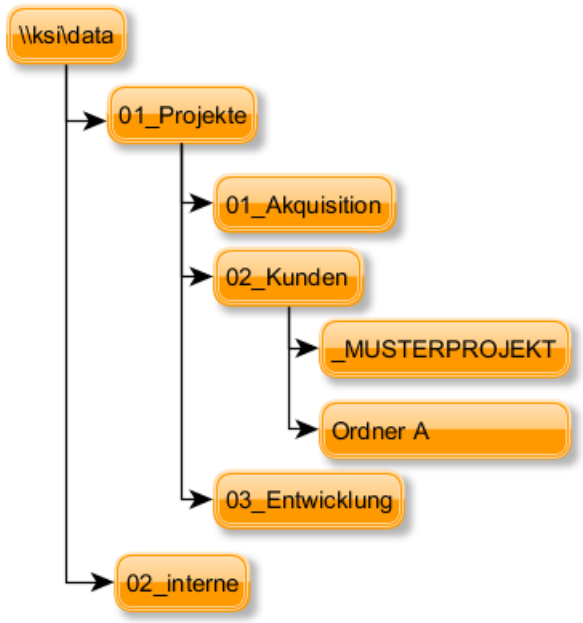
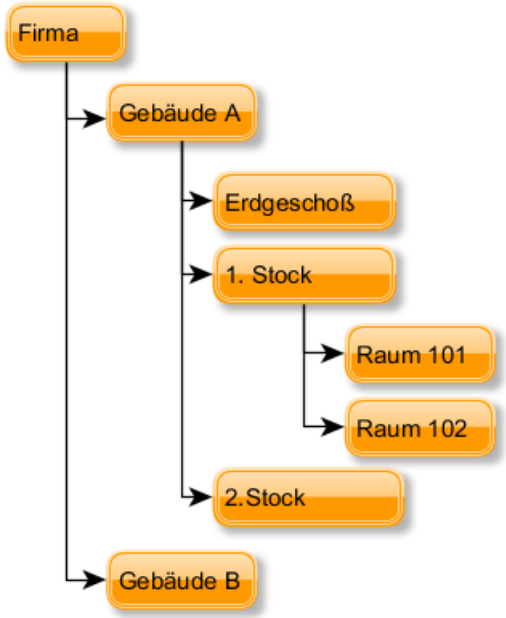
Dateiserver	Gebäudestruktur
	

Tabelle 1 Strukturelle Ähnlichkeit des Dateiservers und einer Gebäudestruktur

Aus diesem Grund muss sich der gewählte Ansatz sehr leicht für diese oder weitere ähnliche Problemstellungen adaptieren lassen.

1.2 Problemanalyse

Es handelt sich also um einen Vergleich (=Matching) von einer gegebenen Baumstruktur mit einer Musterbaumstruktur (=Template). Falls man keinen idealen Vergleichsknoten (=Referenzknoten) ermitteln kann, wird versucht den Bestmöglichen zu liefern. Dadurch kann ein Vergleich auch auf eventuell tieferen Ebenen fortgesetzt werden. Es wird versucht einen Teilbaum mit dem am besten geeignetsten Teilbaum des Templates zu vergleichen, um in weiterer Folge die Benutzer und Benutzergruppen vom aktuellen Baum mit dem Musterbaum zu vergleichen. Die Suche nach dem besten Referenzknoten ist nicht immer ganz einfach, da es oft die Möglichkeit gibt, dass aus mehreren Referenzknoten der Beste ausgewählt werden muss. Dieses Problem tritt aber nur bei indirekten Unifizierungen auf, da eine 1:n Beziehung zwischen Templatestruktur und der zu prüfenden Struktur besteht. Es werden also mit nur einem Referenzknoten mehrere Knoten im zu prüfenden System verifiziert. Hier besteht auch die Möglichkeit, dass ein Referenzknoten mehrere verschiedene Versionen dieser indirekten Vergleichsknoten als Kindknoten halten kann. Dadurch kann nicht sofort auf der aktuellen

Ebene entschieden werden, welcher der möglichen Referenzknoten der geeignetste ist. In Abbildung 1 sieht man, wie komplex eine solche Struktur dann durchaus aufgebaut sein kann. Hier ist nicht sofort ersichtlich, welcher Knoten im Template durch indirekte Unifizierung als bester Referenzknoten für einen bestimmten Knoten im zu prüfenden System fungieren muss. Wenn man Abbildung 1 betrachtet und sich die Frage nach dem besten Referenzknoten für den Knoten mit dem Namen „Knapp_AT_Leoben" aus der zu prüfenden Struktur stellt, ist das auf den ersten Blick nicht so einfach. Solche Strukturen können natürlich auch noch beliebig weiter verschachtelt sein, da der Knotentiefe keine Grenzen gesetzt sind.

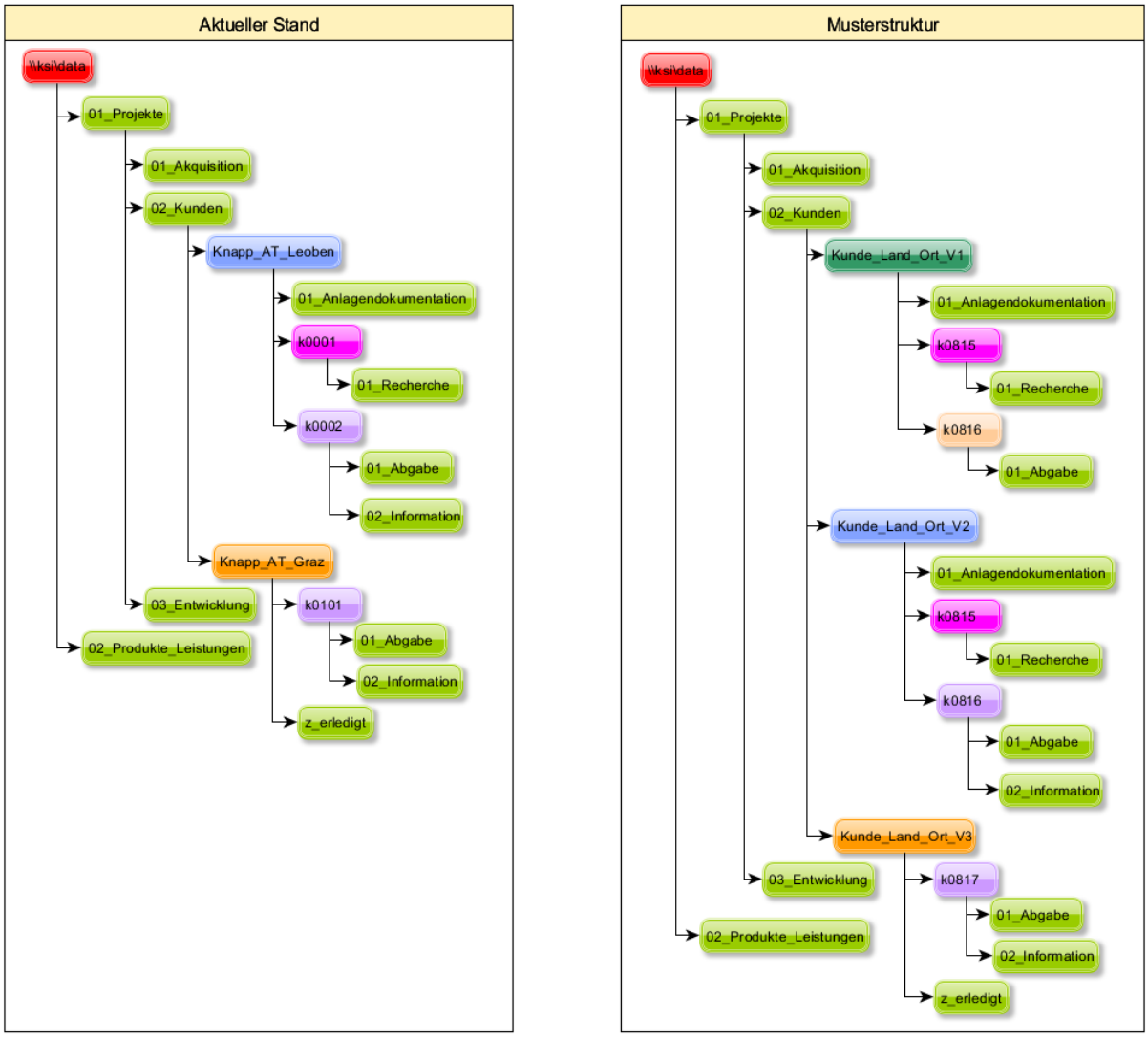


Abbildung 1 Verschiedene Versionen von Musterknoten

Es muss des Weiteren auch möglich sein gewisse Ausnahmen zu definieren, um eine Gültigkeit des aktuellen Baumes, trotz diverser Abweichungen vom Template zu ermöglichen. Diese Ausnahmen sind aber nicht nur auf Strukturebene, sondern auch auf Benutzer- bzw. Benutzergruppenebene inklusive Berechtigungen zu realisieren. Zusätzlich ist es für die Verwendung auf einem Microsoft Windows Dateiserver noch

interessant, welche Ordner und Dateipfade nicht dem NTFS Standard [**1**] entsprechen, weil sie einen zu langen absoluten Pfad haben und somit nicht von anderen Applikationen bzw. dem Windows Explorer angesprochen werden können. Ein interessanter Punkt ist die Erzeugung bzw. die Speicherung der Informationen dieses Templates.

1.3 Aufbau der Arbeit

Diese Arbeit ist folgendermaßen aufgebaut: In Kapitel zwei werden die zu lösenden Probleme genauer beschrieben und die dafür benötigten Definitionen und Techniken vorgestellt. Der Entwurf, entsprechende Ansätze und Ideen werden in Kapitel drei beschrieben. In Kapitel vier werden einzelne problematische Punkte aus Kapitel drei genauer erklärt. Des Weiteren werden die expliziten Implementierungen der jeweils gewählten Ansätze, ihre Probleme und Konsequenzen erläutert. In Kapitel fünf folgt dann eine kurze Leistungsbewertung des Prototyps. Die Schlussfolgerungen und mögliche zukünftige Verbesserungen folgen in Kapitel sechs. Als Abschluss dient in Kapitel sieben eine kurze Zusammenfassung der Arbeit.

2 Grundlagen und Stand der Technik

2.1 Allgemeine Definitionen

Die folgenden vier Definitionen sind Auszüge aus [**2**] und bilden die Grundlage für die zu behandelnde Datenstruktur.

Definition 1 Graph

„Ein Graph besteht aus einer Menge *M* und einer in *M* definierten zweistelligen Relation *R*

$$\mathbb{G} =< M, R > mit\ R \subseteq M \times M$$

Beispiel:

$M = \{A, B, C, D\}$

$R = \{< A, B >, < A, C >, < D, C >\}$"

Definition 2 Knoten

„Ist $\mathbb{G} =< M, R >$ ein Graph, dann nennt man die Elemente von M Knoten des $\mathbb{G}$. In unserem Beispiel sind die Knoten des Graphen die Elemente A, B, C und D." Jeder Knoten besitzt eine gewisse Anzahl an Gruppen. Jede Gruppe hat wiederum verschiedene Berechtigungen definiert. Knoten, die auf andere Knoten verweisen, werden nicht als Knoten betrachtet, da sich keine eigenen Gruppen auf solchen Knoten befinden können. Dadurch werden Verlinkungen innerhalb eines Baumes ignoriert.

Definition 3 Kanten

„Ist $\mathbb{G} =< M, R >$ ein Graph, dann nennt man ein Element aus R eine GERICHTETE KANTE von $\mathbb{G}$. Die gerichteten Kanten in unserem Beispiel sind also die Paare <A, B>, <A, C> und <D, C>."

Definition 4 Baum

„Ein Graph $\mathbb{G} =< M, R >$ heißt Baum, wenn er folgende Eigenschaften hat:

- R ist asymmetrisch und intransitiv;
- Es gibt genau einen Knoten k, zu dem es keinen Knoten x gibt, so dass <x, k> eine Kante aus R ist; dieser Knoten heißt Wurzel.
- Für beliebige Knoten x, y, z aus M gilt: wenn <x, z> eine Kante ist, dann ist <y, z> keine Kante und umgekehrt.

Bedingung (2) besagt, dass es genau einen Knoten gibt, zu dem keine gerichteten Kanten hinführen. Bedingung (3) garantiert, dass zu allen Knoten eine und nur eine Kante hinführt. Jeder Baum enthält Knoten, von denen keine Kanten wegführen. Diese Knoten werden TERMINALE Knoten genannt."

Das bedeutet, dass der Beispielgraph aus Definition 1 kein Baum ist, da die Relationen <A, C> und <D, C> der Bedingung (3) dieser Definition nicht entsprechen.

Definition 5 Knotennamen

$$\mathbb{A} = Menge\ der\ Knotennamen\ aller\ direkten\ Kindknoten\ von\ A \in M;$$
$$a, b \in \mathbb{A}: \nexists a = b$$

Die Knotennamen der direkt untergeordneten Kindknoten müssen eindeutig sein.

Ein einfaches Beispiel für einen Baum, der diese Definitionen erfüllt, ist der Baum, den ein Dateisystem aufspannt. Man geht von einem konkreten Ordner aus, der die Wurzel repräsentiert. Darunter befinden sich dann weitere Ordner, die jeweils nur einem einzigen Knoten direkt untergeordnet sind. Falls ein Ordner keine Unterordner besitzt, kann er laut Definition 4 *als TERMINALER Knoten betrachtet werden.*

Die Struktur eines zu prüfenden Systems, sowie die jeweilige Musterstruktur müssen diesen Definitionen entsprechen. Falls eine der beiden Strukturen diesen Definitionen nicht entspricht, wird sie als ungültig betrachtet und es ist kein Abgleich der beiden Strukturen möglich.

Definition 6 Gewichtung der Knoten

Für jeden direkt unifizierbaren Knoten innerhalb einer wiederkehrenden Unterstruktur gibt es einen Pluspunkt. Wenn man einen Knoten aus dem Template keinem passenden Knoten aus dem System zuordnen konnte, dann wird ein Minuspunkt vergeben.
Handelt es sich um einen Knoten, der mittels indirekter Unifizierung abgeglichen werden muss, dann wird er nur einmal als Pluspunkt gewertet, auch wenn es für diesen Knoten mehrere Versionen gibt. Mit dem „Quantifier" Attribut wird festgelegt, wie oft ein Musterknoten im Template als Referenzknoten fungieren muss. Abhängig von diesem Wert wird dann eine dementsprechende Anzahl an Minuspunkten vergeben.

*Der Knoten im Template mit der Bezeichnung „Kunde_Land_Ort" hat beispielsweise als „Quantifier" den Wert fünf zugewiesen. Das bedeutet, dass dieser Knoten genau fünfmal in der zu prüfenden Struktur vorkommen muss. Wenn er jetzt nur zweimal vorkommt, muss die Differenz der beiden Werte ermittelt werden. Das bedeutet, dass sich die Gewichtung folgendermaßen ändert: Gewicht = Gewicht + MISMATCH * Differenz.*
Da der Knoten dreimal nicht erfolgreich unifiziert wurde, muss er dieser Anzahl entsprechend oft negativ gewichtet werden. Dabei ist die Konstante MISMATCH negativ und die Differenz immer positiv definiert. Somit ergibt sich ein geringeres Gewicht als vor der Änderung.

Definition 7 Zwei Knoten mit exakt gleicher Gewichtung

Wenn zwei potentielle Referenzknoten einen exakt gleichen Gewichtungskoeffizienten liefern, dann ist in erster Linie die Anzahl der Gruppen auf diesen Knoten entscheidend. Der Knoten aus der Musterstruktur, dessen Gruppenanzahl eher mit dem zu prüfenden Systemknoten übereinstimmt, wird als besserer Referenzknoten betrachtet und dadurch auch ausgewählt. Falls die Gruppenanzahl der beiden potentiellen Knoten mit der Gruppenanzahl auf dem Systemknoten übereinstimmt, werden die entsprechenden Berechtigungen der Gruppen dieser Knoten miteinander verglichen und der mit der geringeren Fehleranzahl wird als Referenzknoten gewählt.

2.2 Problemstellungen beim Abgleich der beiden Bäume

Wie in der Einleitung beschrieben wird ein Vergleich von zwei Bäumen angestrebt. Es handelt sich aber nicht nur um einen strukturellen Vergleich der beiden Bäume, sondern auch um einen Vergleich der Knoten auf syntaktischer Ebene und der darüber liegenden Struktur. Ein Knoten des Templates kann als Referenzknoten eines Knotens des zu prüfenden Systems fungieren, wenn die Struktur bis zu diesem Knoten ident ist und eine Unifizierung der Knotennamen möglich ist. Bei dieser Unifizierung wird zwischen den folgenden beiden Varianten unterschieden:

Definition 8 Direkte Unifizierung

Hier handelt es sich um eine 1:1 Beziehung zwischen einem Knoten „R" in der Musterstruktur und einem Knoten „A" innerhalb der zu prüfenden Struktur. Das bedeutet, dass die beiden in „A" und „R" enthaltenen Knotennamen ident sind. Die beiden Knoten müssen sich aber auch unter der gleichen Struktur befinden. In Abbildung 2 und Abbildung 3 ist der Unterschied zwischen einer gültigen und einer ungültigen direkten Unifizierung visualisiert. In Abbildung 2 gibt es jeweils einen Knoten, der in beiden Strukturen unter der identen Struktur liegt und den Knotennamen „Projekte" trägt. Dadurch ist eine direkte Unifizierung dieser beiden Knoten möglich. In Abbildung 3 gibt es zwar wieder jeweils einen Knoten, der den Namen „Projekte" trägt, jedoch liegen diese beiden nicht unter der identen Struktur. Dadurch ist eine direkte Unifizierung dieser beiden Knoten nicht möglich.

Direkte Unifizierung:

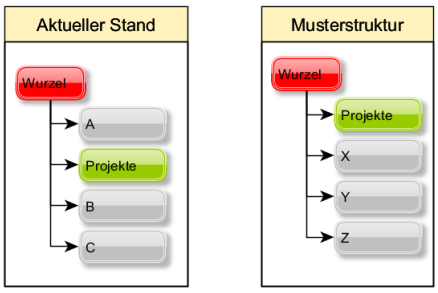


Abbildung 2 Gültige Unifizierung

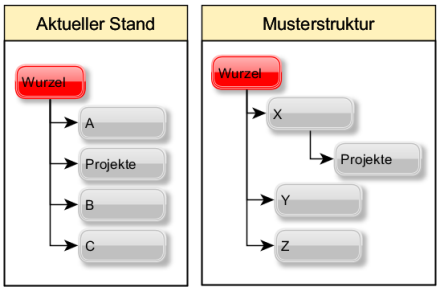


Abbildung 3 Ungültige Unifizierung

Definition 9 Indirekte Unifizierung

Es besteht eine 1:n Beziehung zwischen den Strukturen, da ein Knoten „R" der Musterstruktur als Referenzknoten für mehrere Knoten „A", „B",… der zu prüfenden Struktur dient. In Abbildung 4 ist es gelungen mehrere Knoten („k0001" und „k0002") des zu prüfenden Systems mit dem Knoten „k0815" des Templates indirekt zu unifizieren. Die Namen der beiden Knoten entsprechen einem bestimmten Muster, das im Knoten „k0815" enthalten ist. Dadurch können die beiden von diesem Muster abgeleitet werden. Da sie auch wie ihr entsprechender Referenzknoten unter der identen Struktur liegen, handelt es sich um eine gültige indirekte Unifizierung. In Abbildung 5 gelingt dies nicht, da die Knotennamen „a0001" und „a0002" nicht vom Knoten „k0815" des Templates abgeleitet werden können.

Indirekte Unifizierung:

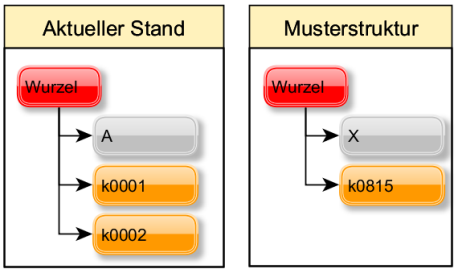


Abbildung 4 Gültige Unifizierung

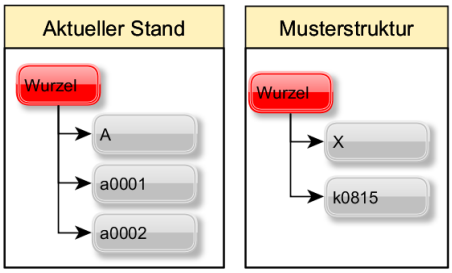


Abbildung 5 Ungültige Unifizierung

Wenn man einen Knoten des zu prüfenden Systems mit einem Knoten des Templates unifizieren kann, spricht man von Gleichheit (nach Definition 10) der beiden Knoten.

Definition 10 Gleichheit

Gleichheit zwischen einem Knoten „A" des zu prüfenden Systems und einem Knoten „R" des Templates ist genau dann gegeben, wenn eine direkte oder indirekte Unifizierung

laut Definition 8 bzw. Definition 9 von „A" und „R" möglich ist. „R" wird dann in weiterer Folge als Referenzknoten von „A" bezeichnet.

2.3 Related Work

Da es sich hier, wie oben erwähnt, um einen Vergleich von Bäumen handelt, ist der erste Ansatz einen passenden Algorithmus zu finden, der sich genau dieser Problematik widmet. In den meisten Fällen, wie beispielsweise in der Arbeit [**3**], werden Bäume durch eine Änderung der Struktur an das Zielformat angeglichen. Dies kann durch Entfernen, Ersetzen oder Einfügen von Knoten realisiert werden. Wenn man eine dieser Operationen anwendet, verändert man dadurch aber entweder die Struktur des zu prüfenden Systems oder das Template, mit dem der Abgleich durchführt werden muss. Deswegen sind Ansätze, die Änderungen an den vorhandenen Strukturen vornehmen müssen, nicht geeignet.
Man kann das Problem aber auch von einer anderen Seite betrachten, indem man die Unifizierung der jeweiligen Knoten des Baumes in den Vordergrund rückt. Dadurch kann man auf den MGU (=Most General Unifier), der auch in Forschungsarbeit [**4**] verwendet wurde, zurückgreifen. Da es sich bei dieser Arbeit auch im weiteren Sinn um einen Unifizierungsprozess handelt, habe ich versucht eine Relation zur Arbeit [**4**] zu finden. Der MGU [**5**] wird für die Unifizierung von logischen Ausdrücken verwendet. Wenn man einen Knotennamen als logischen Ausdruck betrachtet, kann dieser aus Variablen und Konstanten (=Termen) bestehen. Dieser Ansatz funktioniert problemlos bei Knoten, die durch eine direkte Übereinstimmung unifiziert werden. Hier handelt es sich um eine 1:1 Beziehung zwischen einem Referenzknoten in der Musterstruktur und dem Knoten innerhalb der zu prüfenden Struktur. Dadurch erhält man zwei Terme, die ident und somit mittels MGU unifizierbar sind. Problematisch wird es allerdings bei den indirekten Unifizierungen, da man mit Variablen in den Knotennamen arbeiten muss. Die folgenden zwei Beispiele beleuchten diese Thematik etwas genauer:

Der Knotenname in der Musterstruktur lautet beispielsweise „a0815", dieser muss mit mehreren Knoten in der zu prüfenden Struktur verglichen werden. In diesem Fall wird „a" als Konstante bzw. Term betrachtet, die von vier Variablen, die jeweils eine Ziffer von 0-9 annehmen können, gefolgt werden. Diese Problematik ist also mit dem MGU ohne weiteres lösbar.

In einem anderen Szenario lautet der Knotenname in der Referenzstruktur „Kunde_Land_Ort" und muss wieder mit mehreren Knoten der zu prüfenden Struktur verglichen werden. In diesem Fall werden „Kunde", „Land" und „Ort" als Variablen betrachtet, die durch eine Konstante „_" getrennt werden.

Hier fungieren die Zeichenketten als Variablen und sind nicht durch Ziffern definiert. Diese eignen sich nicht unbedingt für Operationen mit logischen Ausdrücken. Man benötigt einen sehr hohen Aufwand bei der Spezifikation der indirekten Referenzknoten, da sehr viele Regeln für die Unifizierung der Zeichenketten definiert werden müssen. Dieser Ansatz ist aus den genannten Gründen somit nicht zielführend.
Da die beiden oben erwähnten Lösungsansätze nicht ausreichend bzw. zielführend sind, wird in dieser Arbeit ein neuer Ansatz verfolgt, der auf den Vergleich von zwei Baumstrukturen (siehe Kapitel 2.1) ausgelegt ist. Die Indirekten Unifizierungen laut Definition 9 werden durch reguläre Ausdrücke ermöglicht. Im Anschluss an diesen Vergleichsprozess stellt der Berechtigungsabgleich keine große Problematik mehr dar. Der in Kapitel 3 vorgestellte Entwurf versucht zu jedem Knoten der zu prüfenden Struktur einen Referenzknoten zu ermitteln. Als Ergebnis erhält man einen Baum, der das zu überprüfende System inklusive detektierter struktureller Fehler sowie Abweichungen der Berechtigungen repräsentiert. Der vorgestellte Algorithmus überprüft die Struktur rekursiv und terminiert in jedem Fall.

2.4 XML

Wenn man eine bestimmte Struktur mit einem dokumentierten Schema vergleichen möchte, muss man sich zuerst überlegen, wie man seine gegebenen Strukturen, die man überprüfen möchte, auf ein Format bringt, das sich gut mit der gegebenen Spezifikation vergleichen lässt. In diesem Fall werden die Baumstrukturen von einem Dateiserver bzw. einem Microsoft SharePoint als aktueller Stand betrachtet und müssen hinsichtlich Berechtigungen und Struktur überprüft werden. Für beide Systeme gibt es Dokumentationen, die sich aus Tabellen und diversen Dokumenten zusammensetzen. Die Strukturen, die in diesen Aufzeichnungen enthalten sind, werden mittels eines XML Dokuments beschrieben, welches durch ein Schema verifiziert werden muss. Für die Erstellung dieser XML Vorlagen ist nur ein grundlegendes Verständnis von XML erforderlich. Die Schemata Erstellung ist mit einer DTD (=Document Type Definition) viel komplizierter zu realisieren, da man beispielsweise keine Möglichkeit hat den Attributen fixe Datentypen zuzuweisen [**6**][1]. Deswegen wird die Verifizierung des XML Dokuments durch ein XML Schema realisiert. Die genauen Unterschiede dieser beiden Techniken werden im weiteren Verlauf noch genauer erläutert. Kenntnisse zur Erstellung einer Document Type Definition sind nicht erforderlich, da man die grundlegenden Unterschiede auch ohne explizite Kenntnisse sehr gut erkennen kann.

[1] Vergleiche Seite 73-111

2.5 Reguläre Ausdrücke

Ausgehend vom dokumentierten Standard wird ein XML Dokument erzeugt, dass das prüfende System in Form eines Templates repräsentiert. Wenn man das fertige XML Dokument erstellt hat, kann man es theoretisch schon mit dem aktuellen Stand vergleichen. Problematisch ist nur, dass man überall wiederkehrende Unterstrukturen hat, die man mehrfach im Template definiert hat. Für diese Problematik werden reguläre Ausdrücke benötigt. Dadurch ist es möglich, mit nur einem Ausdruck mehrere verschiedene Knotennamen zu unifizieren. Hierfür sind wiederum nur die grundlegendsten Techniken für die Arbeit mit regulären Ausdrücken erforderlich.
Tabelle 2 zeigt die relevanten Muster mit ihren Erklärungen, die in weiterer Folge öfters verwendet werden. Diese Übersicht ist eine Kurzfassung aus [**7**].

Muster	Bedeutung
.	Beliebiges Zeichen
k	Nur das Zeichen „k" wird gematcht
[a-c]	Bereich von a bis c
[a-zA-Z]	Bereich von a bis z und von A bis Z
\w	Zeichen aus dem Alphabet [a-z, A-Z, 0-9, ä, ö, ü, Ä, Ö, Ü, ß, _]
\d	Ziffer [0-9]
\s	Whitespace [Leerzeichen, Tabulatur, Zeilenumbruch,…]
Verknüpfungen	
|	Oder
Quantifizierungen	
?	Muster darf maximal einmal vorkommen
*	Muster darf beliebig oft vorkommen
+	Muster muss mindestens einmal vorkommen
{n}	Legt die Anzahl der Übereinstimmungen fest
Position	
^	Der gewünschte Ausdruck muss am Anfang stehen
$	Der gewünschte Ausdruck muss am Ende stehen

Tabelle 2 Übersicht der gebräuchlichsten regulären Ausdrücke

Die folgenden Beispiele veranschaulichen die Verwendung von Kombinationen der oben abgebildeten Muster:

Regulärer Ausdruck	^[a-zA-Z]{2}([a-zA-Z])?$
Matches	AT, DE, AUT,
Mismatches	A, F, Österreich, 123
Erklärung	Repräsentiert einen Ländercode

Regulärer Ausdruck	^[a-zA-Z\d-&]+_[a-zA-Z]{2}([a-zA-Z])?(_[a-zA-Z\d-&]+)?$
Matches	Kunde_AUT_Wien, Kunde_AT_Graz, Kunde_AUT, Kunde_DE
Mismatches	Kunde AUT Leoben, Kunde_D, Kunde_AUT_Leoben_, 01Kunde_AT
Erklärung	Wird für den Abgleich der Kundenordner in den einzelnen Strukturen verwendet.

Regulärer Ausdruck	^k\d\d\d\d bzw. ^k\d{4}
Matches	k1234, k1234_Dokumentation, k6782 Projekt
Mismatches	a1234, 1234, c123
Erklärung	Wird für einzelne Kundenprojekte innerhalb eines Kundenordners verwendet.

Tabelle 3 Beispielkombinationen für reguläre Ausdrücke

3 Entwurf

3.1 Zwei mögliche Lösungsansätze

3.1.1 Datenbankansatz

Die zu prüfende Baumstruktur wird traversiert und dabei werden die Knotennamen, Benutzergruppen und Berechtigungen in einer Datenbank gespeichert. Abbildung 6 veranschaulicht die Architektur dieses Ansatzes.

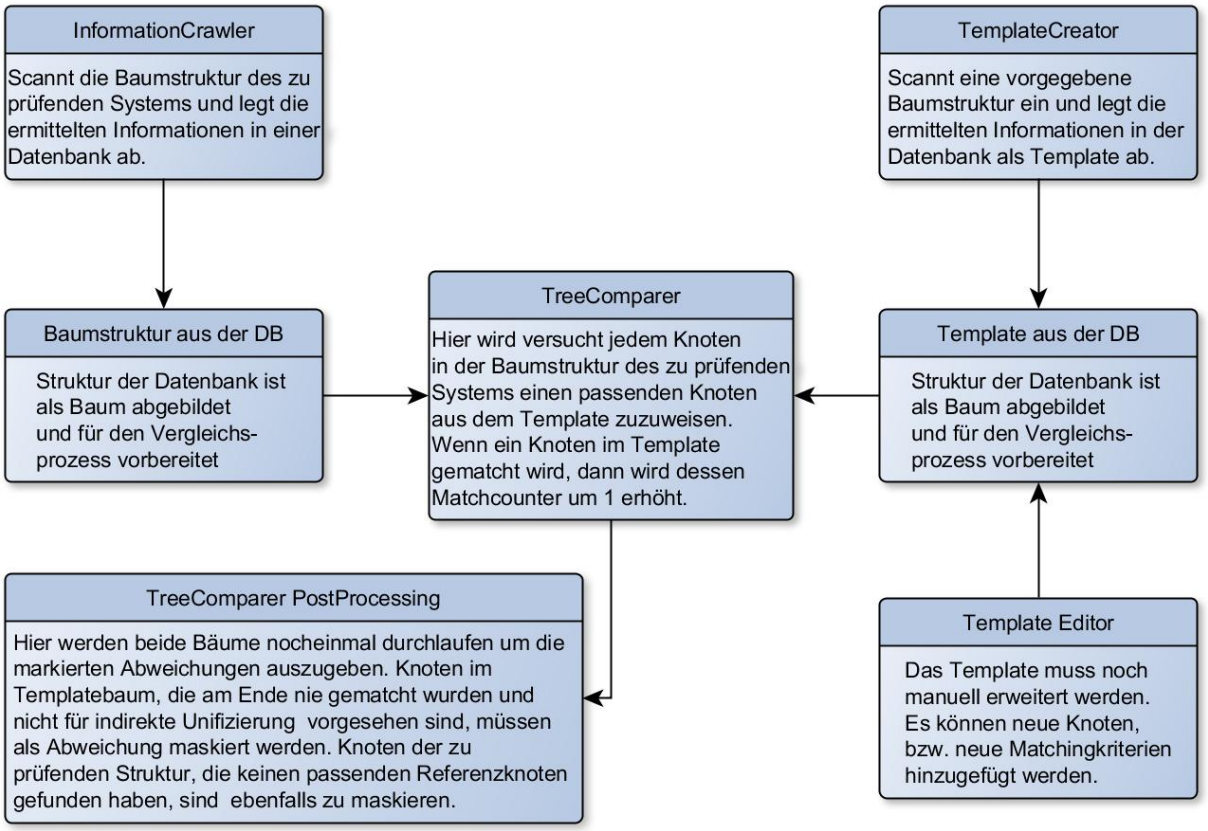


Abbildung 6 Architektur des Datenbankansatzes

Die beiden zu vergleichenden Bäume werden durch eine komplette Traversierung der jeweiligen Struktur erzeugt. Die Musterstruktur enthält noch zusätzliche Informationen, die für den Vergleichsprozess relevant sind. Man kann das Template aber nachträglich noch manuell bearbeiten, um relevante Informationen hinzuzufügen. Danach kann man die beiden Bäume miteinander vergleichen. Als Ergebnis erhält man die selektierten Unterschiede der Benutzer bzw. Berechtigungen auf den einzelnen Knoten der Bäume.

Vorteile: Man hat alle benötigten Informationen in der Datenbank und muss die aktuelle Baumstruktur, die man überprüfen möchte, nur einmal traversieren. Da man alle relevanten Informationen in der Datenbank hat, ist es später auch kein Problem

irgendwelche Erweiterungen am System vorzunehmen, die die zu überprüfende Struktur längere Zeit für sich beanspruchen.

Nachteile: Das größte Problem hierbei ist die Datenmenge, die gespeichert werden muss.

Zurzeit handelt es sich beispielsweise am Dateiserver um ca. 2 Millionen Dateien und Ordner, die jeweils diverse Benutzergruppen mit entsprechenden Berechtigungen beinhalten. Die Datenbanktabelle mit den verschiedenen Berechtigungen auf allen Dateien wird ungefähr 30 Millionen Einträge beinhalten. Hier kann es zu Laufzeitproblemen kommen, wenn man zu einer konkreten Datei die berechtigten Benutzer aus dieser Berechtigungstabelle ermitteln möchte: Wenn diese Abfrage beispielsweise ungefähr eine Sekunde benötigt, nimmt dieser Vorgang theoretisch über 100 Stunden in Anspruch, um alle Dateien mit den entsprechenden Benutzergruppen zu verbinden, was für den späteren Vergleichsprozess von bedeutender Relevanz ist. Falls es hier zu Zeitproblemen kommt, muss man die Datenbanklösung auf Ordner beschränken, was eine Reduktion der Inhalte am Dateiserver auf ungefähr 500.000 Einträge bewirken wird und somit auch die Berechtigungstabelle auf etwa 7 Millionen Einträge verringert.

Da man bei dieser Variante während der Systemtraversierung noch kein Template zur Verfügung hat, muss man sich aber jeden Knoten (im Dateiserverszenario die Ordner) anschauen, da er für den späteren Vergleichsprozess relevant sein kann. Dieses Problem umgeht man, indem das Template einfach vorher erzeugt wird, wodurch man aber den wesentlichen Vorteil der Datenbank verspielt: Alle Informationen sind in der Datenbank, auch die der Knoten, die zu diesem Zeitpunkt noch nicht überprüft werden sollen. Somit muss man nochmal das gesamte zu prüfende System traversieren, um die gewünschten Informationen zu sammeln. Diese wurden eventuell auf Grund der Definition des Templates zu einem früheren Zeitpunkt ignoriert.

3.1.2 „On-the-Fly"

Hier wird im Gegensatz zur Datenbankvariante, bei der es grundsätzlich egal ist, welchen Strukturbaum man zuerst erzeugt, die Musterstruktur als erstes angelegt. Im Anschluss daran wird der aktuelle Baum traversiert und nur wenn man einen passenden Referenzknoten in der Musterstruktur finden kann, wird der Knoten überprüft bzw. seine Kindknoten überhaupt betrachtet. Die Architektur dieses Ansatzes ist in Abbildung 7 veranschaulicht:

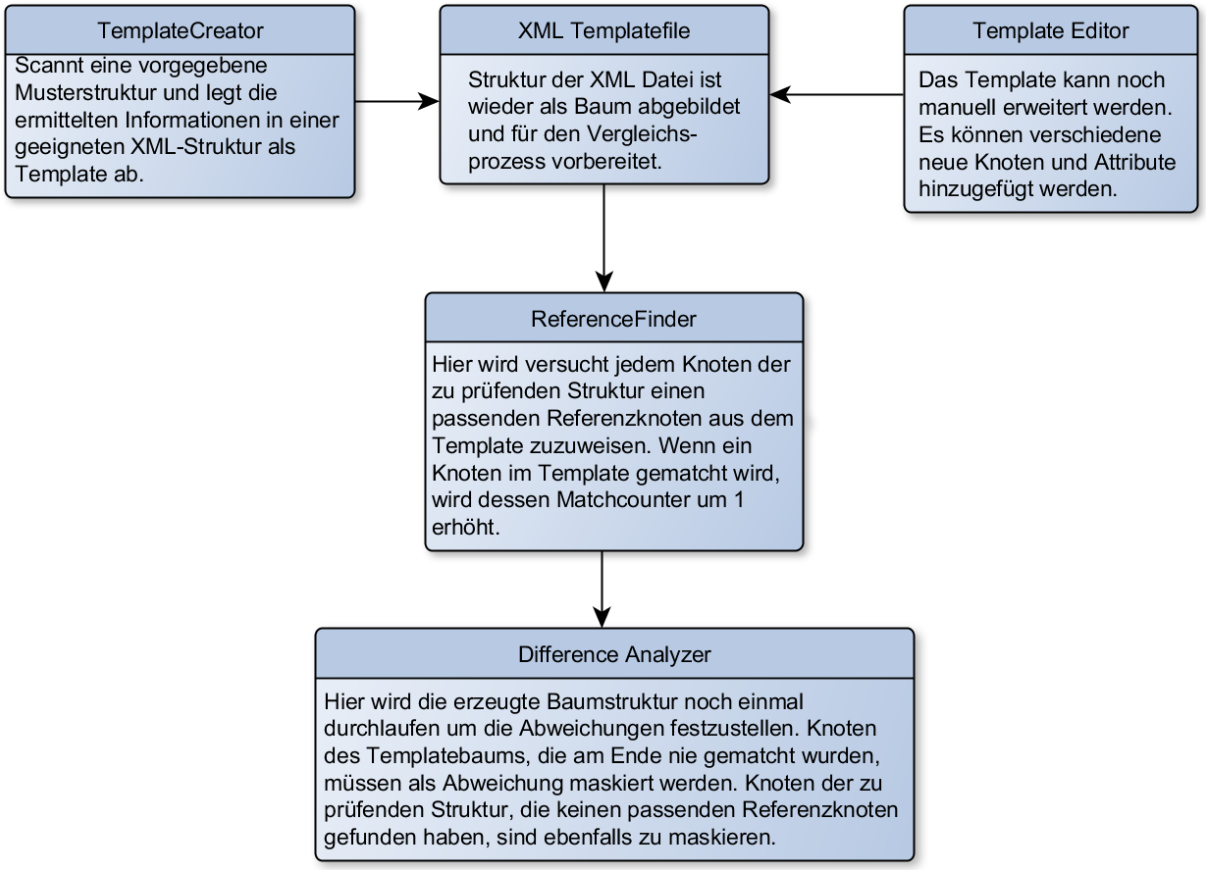


Abbildung 7 Architektur des „On-the-Fly" Ansatzes

Die Erzeugung des Templates basiert wie im Datenbankansatz auf einer Musterstruktur des zu prüfenden Systems. Problematisch hierbei ist aber die Serialisierung der Teilergebnisse, da man nicht immer den gesamten Prozess starten möchte, um die Unterschiede zu visualisieren. Die Datenmenge, die hierbei gespeichert werden muss, kann zu Problemen beim Einlesen dieser Daten führen. Hierfür muss man sich eine geschickte Klassenstruktur überlegen, um mit so wenigen Informationen wie möglich die Struktur wieder aufbauen zu können.

Vorteile: Man erzielt eine weit bessere Laufzeit vom gesamten Prozess, da Kindknoten nur dann betrachtet werden, wenn für den entsprechende Elternknoten ein passender Referenzknoten im Template definiert ist. Das bedeutet also, dass die 20. Ebene nicht mehr betrachtet wird, wenn der entsprechende Referenzknoten im Template nur bis zur dritten Ebene definiert ist. Das spart enorm viel Zeit, da große Teilbäume unten einfach abgeschnitten werden, da sie laut Definition des Templates nicht überprüft werden müssen.

Nachteile: Ein großer Nachteil ist allerdings, dass man nur mit Knoten im Template vergleicht und dadurch darunterliegende Knoten überhaupt nicht berücksichtigen kann. Man sammelt keinerlei Informationen über Strukturen die unter einem Knoten liegen, der

keinem Referenzknoten zugeteilt ist. Im Datenbankansatz hat man den Vorteil, dass diese Informationen schon zur Verfügung stehen, da sie durch einen kompletten Suchlauf über das zu prüfende System bereits in einer Tabelle gespeichert sind. Somit muss nur im Template ein neuer Knoten definiert werden, um einen neuen Vergleichsprozess bei der Datenbankvariante zu starten. Dies ist bei diesem Ansatz nicht möglich, da ein komplett neuer Durchlauf des zu prüfenden Systems gestartet werden muss, welcher relativ viel Zeit in Anspruch nehmen kann.

3.1.3 Angewandte Strategie

Ich habe mich für die zweite Variante entschieden, da sie in meinen Augen universeller eingesetzt werden kann und einen geringeren Overhead darstellt. Für die Datenbankvariante muss man eine Datenbank inklusive passender Tabellen designen und ansprechen können. Dementsprechend hat man in weiterer Folge einen höheren Verwaltungsaufwand, der nicht unbedingt notwendig ist. Die folgende Hybridlösung der beiden Varianten macht eventuell mehr Sinn, wurde aber nicht weiter verfolgt:
Im ersten Schritt sammelt man alle Informationen des zu prüfenden Systems und legt sie in der Datenbank ab. Im zweiten Schritt wendet man den „On the Fly" Ansatz auf die Informationen in der Datenbank, und nicht mehr direkt auf das Live-System an. Diese Lösung hätte den Vorteil, dass die zu prüfenden Systeme nicht so oft belastet werden müssen, um Informationen zu sammeln und hätte beim Abgleich den Vorteil, dass man direkt mit dem Template abgleichen kann. Ein kurzer Vergleich der beiden Varianten ist in Tabelle 4 zu sehen:

	Datenbank	On The Fly
Anforderungen	Man benötigt eine Datenbank und muss die Struktur der einzelnen Tabellen definieren. -	Keine weiteren Anforderungen, da die Informationen in Dateien abgelegt werden. +
Codekomplexität	Man muss sich um Kommunikation und die Fehlerbehandlung der Datenbankverbindung kümmern. -	Zu speichernde Informationen werden in Form von Dateien serialisiert und gespeichert. +
Belastung des zu prüfenden Systems	+	-
Gesamt	**-**	**+**

Tabelle 4 Gegenüberstellung der beiden Lösungsansätze

3.2 Templateerzeugung

Die Erstellung des Templates ist ein grundsätzliches Problem, das unabhängig von den beiden oben beschriebenen Lösungsansätzen betrachtet werden kann. Man vergleicht eine kompakte und schlanke Struktur (das Template) mit einem beliebig großen Baum. Hier wird zwischen direkter und indirekter Unifizierung laut Definition 8 und Definition 9 unterschieden. Bei einer direkten Übereinstimmung muss der Knoten, wie in Kapitel 2 erläutert, den exakt gleichen Namen wie der korrespondierende Knoten des Templates haben. Wenn ein Knoten des zu prüfenden Systems mit dem regulären Ausdruck eines Knotens der Musterstruktur unifiziert werden konnte, handelt es sich um eine indirekte Übereinstimmung. Abbildung 8 verdeutlicht die Unterschiede zwischen den beiden Unifizierungsprozessen noch einmal in einem größeren Szenario als in Kapitel 2 beschrieben.

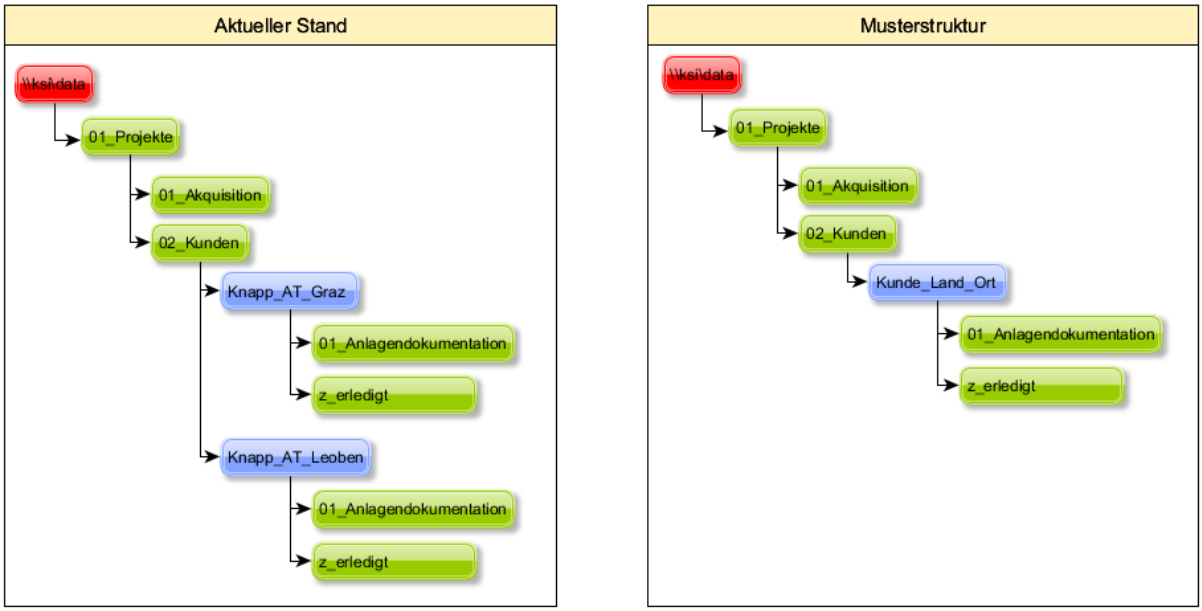


Abbildung 8 Unterschiedliche Unifizierungsmöglichkeiten

Die Wurzeln der beiden Strukturen sind rot markiert, da sie auf jeden Fall miteinander verbunden werden, egal ob sie übereinstimmen oder nicht. In diesem Beispiel sind direkte Übereinstimmungen grün und indirekte Übereinstimmungen blau hinterlegt. In Abbildung 8 ist es so, dass man in der Musterstruktur im „02_Kunden" Knoten einen Kindknoten definiert hat, der über einen regulären Ausdruck mit einem Knoten des aktuellen Zustands verglichen wird und dadurch als indirekte Unifizierung gehandhabt wird. Dieser Knoten trägt im Template den Namen „Kunde_Land_Ort" und enthält den regulären Ausdruck „^[a-zA-Z\d-&]+_[a-zA-Z]{2}([a-zA-Z])?(_[a-zA-Z\d-&]+)?$", der die beiden Knoten des aktuellen Standes „Knapp_AT_Graz" bzw. „Knapp_AT_Leoben" über einen indirekten Vergleich unifizieren kann. Die darunterliegende Struktur kann natürlich wieder jede beliebige Form annehmen. Auch weitere indirekte Musterknoten

sind als Kindknoten erlaubt. Hier stößt man aber auf ein neues Problem: Verschiedene Versionen von indirekten Unifizierungen sind erlaubt. Es kann also passieren, dass es beispielsweise eine Musterstruktur für alte und neue Kundenprojekte gibt, die sich in ihrer Namensgebung im aktuellen Strukturbaum nicht unterscheiden. Sie weisen nur in einer tieferen Ebene strukturelle Abweichungen voneinander auf. Unterschiede sind eventuell in der Anzahl oder in der Namensgebung dieser Unterknoten feststellbar. Deswegen ist auch nicht sofort entscheidbar, zu welchem Knoten im Template man eine Verbindung herstellen muss, um einen gültigen Referenzknoten zu erhalten. Das ist auch der Grund, wieso man alle möglichen Referenzknoten auf ihre darunterliegende Knotenstruktur überprüfen muss. Es wird dann der Basisknoten ausgewählt, der die ähnlichste darunterliegende Struktur aufweist. Bei diesem Prozess kann es auch passieren, dass man eine Struktur vorfindet, die aus mehreren versionsabhängigen Unterstrukturen besteht. Um diese etwas komplizierteren Rahmenbedingungen ein wenig zu veranschaulichen, sind die beiden nachfolgenden Abbildungen (Abbildung 9 und Abbildung 10) angedacht:

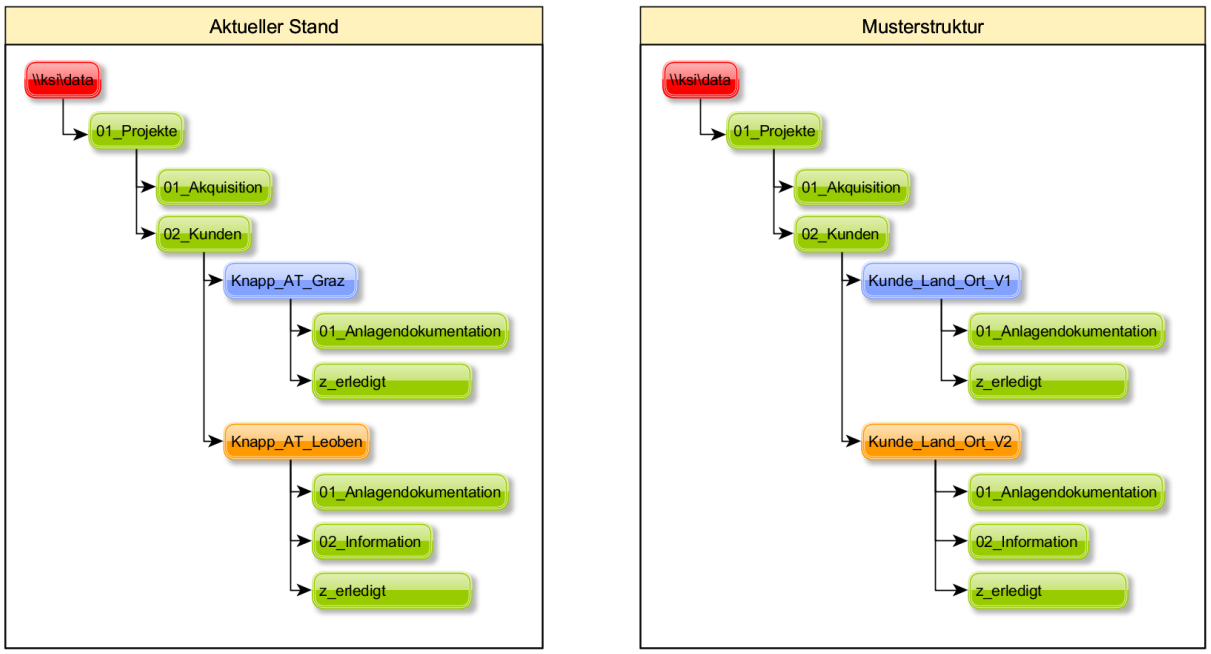


Abbildung 9 Zwei einfache, relativ ähnliche Strukturen

Abbildung 9 zeigt in der Musterstruktur zwei verschiedene Versionen der Kundenprojekte bzw. deren darunterliegende Struktur. Die alte Version befindet sich unter dem blauen Knoten und die neue liegt unter dem orangen Knoten auf der rechten Seite im Template. Beide Versionen unterliegen den gleichen Regeln der Namensgebung. Das bedeutet, dass sie beide vom gleichen Knoten im Template über eine indirekte Unifizierung gematcht werden können. Deshalb ist es auf der Ebene der verschiedenen Kundenprojekte noch nicht möglich zu bestimmen, ob es sich um eine neue oder eine alte Version dieses Knotens handelt.

Man muss zuerst die darunterliegende Struktur analysieren, um bestimmen zu können, welcher Knoten im Template ein Referenzknoten für einen Knoten des zu prüfenden Systems sein kann. Abbildung 10 ist ein wenig komplizierter aufgebaut und bildet die eigentliche Problematik dementsprechend besser ab:

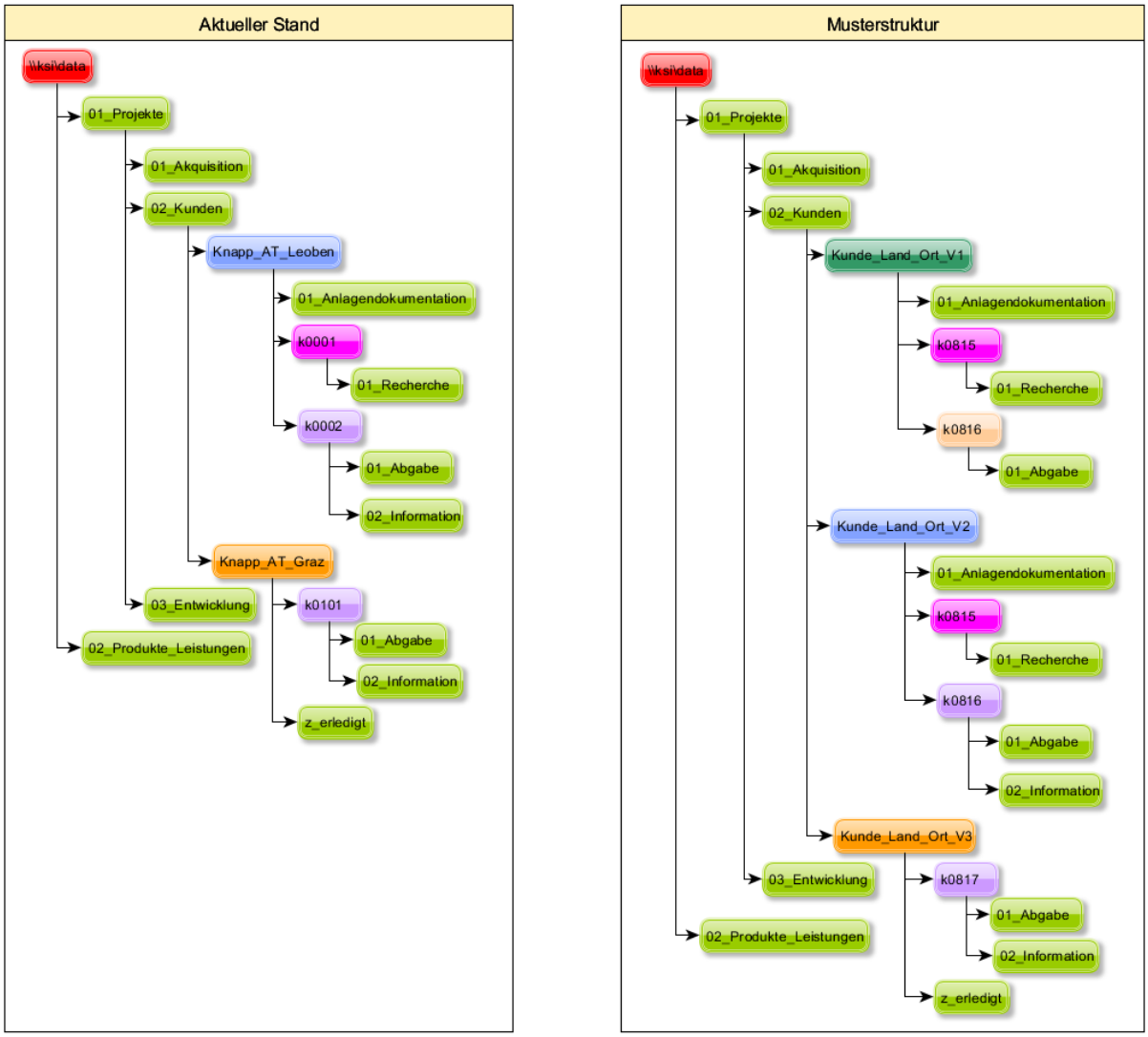


Abbildung 10 Zwei komplexere Strukturen

Das erste Entscheidungsproblem tritt in Abbildung 10 auf, wenn der Knoten „Knapp_AT_Leoben" überprüft wird. Dieser kann zu Version 1, Version 2 oder Version 3 gehören. Da man aber nicht sofort sagen kann, welcher Knoten im Template der passende Referenzknoten ist, muss man sich die darunterliegenden Ebenen ebenfalls anschauen. Man erkennt in weiterer Folge, dass es sich nicht um die dritte Version des Knotens handelt, da kein passender Knoten für „k0001" definiert ist, der die passende Unterstruktur hat. Somit handelt es sich wahrscheinlich um Version 1 oder Version 2. Hier ist auf der darunterliegenden Ebene aber auch noch kein Unterschied feststellbar, da die beiden Strukturen auf diesem Level ident sind. Erst eine weitere Ebene tiefer erkennt man, dass es sich definitiv um die zweite Version des Kundenordners handelt, da zu allen

Knoten ein passender Referenzknoten definiert ist. Dieses kleine Beispiel zeigt, dass die Suche nach der passenden Struktur im Template, zu einer gegebenen Struktur des aktuellen Zustandes, nicht immer ganz trivial ist. In manchen Fällen passt aber keine der vordefinierten Strukturen exakt zu einer vorhandenen Struktur des zu prüfenden Systems. An dieser Stelle muss man mit einer entsprechenden Gewichtung arbeiten, um den Referenzknoten zu ermitteln, der die ähnlichste darunter liegende Struktur aufweist.
Hier taucht dann auch eine weitere sehr zentrale Fragestellung auf: „Wie erzeugt man so ein Template und speichert darin die notwendigen Informationen für einen späteren Abgleich?"

3.2.1 Manuelle Erzeugung eines Templates

Da diese Anwendung in weiterer Folge eine enge Interaktion mit dem Microsoft SharePoint haben soll, liegt die Problemlösung durch eine SharePoint Liste relativ nahe. Dafür wird eine Liste am SharePoint entwickelt, die folgendermaßen aufgebaut ist (siehe Abbildung 11):

DirTitle	FileShareRoot	Parent	Recurrent	RegEx	onDirectory	Right_1	Right_2
TestDirBase	D:\TestDirBase						
01_Projekte		TestDirBase	No				
PMA NEW					01_Projekte	Yes	No

Abbildung 11 Templaterealisierung als Liste am SharePoint

Es gibt zwei verschiedene Knotentypen:

- Knoten, die eine Website, Liste, Bibliothek oder einen Ordner mit den folgenden Attributen repräsentieren: DirTitle, FileShareRoot, Parent, Recurrent, RegEx, …
- Gruppenknoten, die für einen bestimmten Benutzer oder eine Gruppe stehen und mit den entsprechenden Berechtigungen versehen sind. Die folgenden Attribute wurden hierfür festgelegt: onDirectory, Right_1,…, Right_X. Diese Berechtigungen halten im Hintergrund einen Wert, der sich bei der Kombination verschiedener Rechte wieder zu einer gesamten Berechtigung zusammenführen lässt.

Dieser Lösungsansatz ist sehr allgemein gehalten und somit für viele Szenarien einsetzbar. Da es sich, wie vorher erwähnt, um ungefähr eine halbe Million Ordner auf dem Dateiserver handelt, erscheint diese Lösung, trotz ihrer guten allgemeinen Anwendbarkeit, als wenig zweckmäßig. Problematisch ist natürlich auch die Wartung einer solchen Liste, da man Änderungen manuell durchführen muss. Im Grunde hat man in der Praxis im Umgang mit dem Dateiserver aber ein noch viel größeres Problem: Man kann aus der Struktur des Templates nicht einfach einen Teilbaum auf den Dateiserver kopieren. Dieser Vorgang muss aber möglich sein, da beispielsweise ein neues Kundenprojekt mit dementsprechender Struktur und Berechtigungen direkt aus der

Musterstruktur erzeugt werden muss. Hier müsste man sozusagen zwei Templatestrukturen synchron halten, da man nicht ohne große Änderungen der aktuellen Workflows neue Kundenordner auf dem Dateiserver anlegen kann. Man müsste die Workflows so anpassen, dass aus der Liste am SharePoint, die als Template dient, neue Ordnerstrukturen automatisiert mit entsprechenden Berechtigungen am Dateiserver erzeugt werden können. Das ist zurzeit aber ein relativ großer Aufwand und nur schwer realisierbar.

3.2.2 Semiautomatische Erzeugung eines Templates

Dieser Ansatz ist weitestgehend automatisiert und fügt sich sehr gut in die bestehenden Workflows ein. Dafür leidet die Abstraktion dieser Lösung und sie ist nicht mehr ohne geringe Änderungen für verschiedene Szenarien einsetzbar.
Eine Musterstruktur des Dateiservers wird direkt auf Dateisystemebene realisiert, indem man die in Dokumenten definierte Struktur direkt auf das Dateisystem überträgt. Der Vorteil bei dieser Variante ist, dass diese Musterstruktur schon teilweise vorhanden ist und nur mehr umgebaut werden muss. Informationen bezüglich Berechtigungen sind schon im Idealzustand auf den einzelnen Ordnern vorhanden. Man muss nur mehr geringfügige Änderungen durchführen, um sich weitere Informationen für indirekte Unifizierungen auf dieser Struktur merken zu können. An dieser Stelle tut sich aber eine neue Frage auf: „Wo kann man Informationen für indirekte Unifizierungen oder andere Parameter ablegen?"

Im Falle des Dateiservers, der bei dieser Entwurfsentscheidung die größte Rolle spielt, gibt es bei NTFS die Möglichkeit sich Metainformationen auf einzelnen Dateien zu merken. Diese Option steht bei Ordnern aber nicht zur Verfügung. Das bedeutet, dass man keine Möglichkeit hat sich eine Information, wie etwa einen regulären Ausdruck, direkt am Ordner zu vermerken. Somit muss das Problem über Umwege mittels einer Konfigurationsdatei gelöst werden. Diese befindet sich direkt in den Ordnern, die über eine indirekte Unifizierung abgeglichen werden sollen.

In dieser Konfigurationsdatei werden dann auch noch weitere Informationen abgelegt, wie es auch bei der Liste am SharePoint möglich ist. Beispielsweise kann man auch vermerken, dass dieser Knoten übersprungen werden soll oder dass ab diesem Knoten alle darunterliegenden Knoten die gleichen Berechtigungen vererbt bekommen müssen. Es ist auch möglich eine bestimmte Anzahl von indirekten Unifizierungen dieses Knotens zu definieren. Damit wird festgelegt, dass es nicht mehr bzw. weniger als X oder genau X Überstimmungen mit diesem Knoten geben darf. Der große Vorteil von indirekten Unifizierungen ist, dass man mehrere verschiedene Strukturen, die alle dem gleichen Muster entsprechen, mit nur einem Referenzknoten aus der Musterstruktur abgleichen

kann. Somit ist es nicht mehr erforderlich für jeden Kundenordner einen eigenen Knoten zu definieren. Das führt abhängig von der zu prüfenden Struktur zu einer erheblichen Reduktion der Knotenanzahl im Template. Wenn man eine Struktur überprüfen muss, die aus sich oft wiederholenden Unterstrukturen besteht, reduziert man mit diesem Ansatz die Komplexität des Templates um einen erheblichen Faktor.

3.2.3 Hybrider Lösungsweg zur Templateerzeugung

Für die Erzeugung des Templates am Dateiserver habe ich mich für die zweite Variante entschieden, da sie sich leichter in die aktuellen Prozesse integrieren lässt. In diesem Szenario wird die Musterstruktur auf dem Dateiserver in einer normalen Ordnerstruktur abgebildet und anschließend traversiert, um das Template als XML Dokument zu erzeugen. Um eine ungefähre Vorstellung von so einem Template zu bekommen, ist Abbildung 12 vorgesehen:

```
<Folder Name="\\ksi\data" xmlns:xsi="http://www.w3.org..." xsi:noNamespaceSchemaLocation="..\fsSchema.xsd">
  <Folder Name="01_Test" Fullname="..\01_Test">
  <Groups><Group Name="sed" Rights="FullControl" Permissionbits="11111000000111111111" /></Groups>
  <Folder Name="01_ABC" Fullname="..\01_ Test\01_ABC" MatchOnlyPermissionsFromHere="true">
    <Groups>
      <Group Name="LIH" Rights="ReadAndExecute, Synchronize" Permissionbits="10010000000010101001" />
      <Group Name="sed" Rights="FullControl" Permissionbits="11111000000111111111" />
    </Groups>
  </Folder>
  <Folder Name="Ku_La_Or" Fullname="..\01_ABC\Ku_La_Or" Regex="\w+_\w\w(\w)?(_\w+)?" Quantifier="*">
    <Groups><Group Name="sed" Rights="FullControl" Permissionbits="11111000000111111111" /></Groups>
    <Folder Name="01_XYZ" Fullname="..\01_ABC\Ku_La_Or\01_XYZ"></Folder>
    <Folder Name="k5678" Fullname="..\01_ABC\Ku_La_Or\k5678" Regex="k\d\d\d\d" Quantifier="2"></Folder>
  </Folder>
</Folder>
```

Abbildung 12 Kurzer Auszug aus dem Xml Template für den Dateiserver

Die Erzeugung des Templates für den SharePoint ist leider etwas komplizierter, da man sich keine Musterstruktur wie beim Dateiserver aufbauen kann, um daraus dann automatisiert ein XML Dokument zu erzeugen. Hier ist eine gänzlich manuelle Erzeugung des XML Dokumentes erforderlich, da man nicht ohne weiteres einen zusätzlichen SharePoint konfiguriert und betreibt, der dann nur die Musterstruktur repräsentiert. Zu diesem Zeitpunkt ist die Struktur des SharePoints aber noch relativ kompakt, die Anzahl der Knoten noch überschaubar und deswegen noch keine Automatisierung der Erzeugung des Templates erforderlich. In weiterer Folge muss man aber den ersten Ansatz mit der SharePoint Liste weiterverfolgen bzw. eventuell auch auf eine Bibliothek am SharePoint ausweichen, da die Liste am SharePoint nur eine begrenzte Anzahl an Einträgen verwalten kann. Generell ist der erste Lösungsansatz in kommenden Szenarien aber die bessere Variante, da die Strukturen alle zentral verwaltet werden können und durch eigene Datentypen spezifizierbar sind. Die Workflows dieser Szenarien müssen dann aber auch in enger Interaktion mit diesen definierten Strukturen arbeiten, damit es zu keinen redundanten Datenbeständen kommt.

3.3 XML Dokumente als Musterstruktur

Die Entscheidung, die Musterstruktur mittels XML zu realisieren, ist relativ früh gefallen. Sie basiert auf meinen gewonnen Erkenntnissen, die aus den folgenden Quellen stammen: [**6**], [**8**], [**9**], [**10**] und [**11**].
XML ist ein anerkannter offener Standard, der eine unkomplizierte Änderung der Vorlagenstruktur jederzeit ermöglicht. Falls die Hinterlegung neuer Informationen in den Knoten erforderlich ist, gelingt dies problemlos durch das Hinzufügen von zusätzlichen Kindknoten bzw. Attributen. Dadurch ist auch in Zukunft ein Abgleich mit anderen Strukturen, die zu diesem Zeitpunkt noch nicht bekannt sind, gewährleistet. Die einzige Voraussetzung ist eine erfolgreiche Portierung auf diesen Standard, was für simple Baumstrukturen in den meisten Fällen kein Problem darstellt. Jede XML Vorlage muss natürlich eine bestimmte Struktur aufweisen, um einen erfolgreichen Abgleich zu gewährleisten. Für diese Verifizierung standen die etwas in die Jahre gekommene „DTD"(= Document Type Definition) und deren Nachfolger „XML Schema" zur Auswahl. Hier fiel die Wahl nach Studium der oben genannten Quellen relativ schnell auf den neuen Standard. Ein „XML Schema" basiert auch auf der XML Syntax und somit entsteht kein zusätzlicher Einarbeitungsaufwand. Des Weiteren kann man den Attributen eigene Datentypen zuweisen, was eine leichtere Überprüfung der jeweiligen Attributwerte ermöglicht. Dieses Feature stellt einen großen Vorteil bei der inhaltlichen Überprüfung der Musterstruktur gegenüber dem älteren DTD Standard dar. In Kapitel 4 „Implementierung" werde ich auf den Aufbau solcher Musterstrukturen, deren Verifizierung und die Unterschiede zwischen einer DTD und einem „XML Schema" noch genauer eingehen. Dadurch wird auch sehr schnell klar, dass die eigens definierten Datentypen einen enormen Vorteil darstellen.

3.4 Finden des passenden Referenzknotens

Dieser Punkt ist die größte Herausforderung, da schon bei den Überlegungen zur Erzeugung des Templates klar wurde, dass es relativ zeitaufwendig sein wird, den gesamten Dateiserver bzw. Microsoft SharePoint zu traversieren. Ein weiteres zeitliches Problem stellen die verschiedenen Versionen von Knotenstrukturen dar, bei denen man nicht sofort feststellen kann, um welche Ausprägung es sich tatsächlich handelt. Wenn man hier nicht rechtzeitig erkennt, dass es sich wahrscheinlich nicht um den gesuchten Knoten handelt, benötigt man eventuell zu viel Zeit bei der Suche nach dem passenden Referenzknoten im Template. Das hat zur Folge, dass man gegebene zeitliche Rahmenbedingungen vielleicht nicht mehr einhalten kann. Hier ist es sinnvoll, wenn man entscheiden kann, ob schon bei der ersten Abweichung der aktuelle Knoten als nicht optimaler Referenzknoten betrachtet wird oder ob der Vorgang trotz auftretender Abweichungen fortgesetzt werden soll.

In den vorliegenden Szenarien des Dateiservers bzw. des SharePoints ist es grundsätzlich so angedacht, dass es immer eine ideale Knotenstruktur im Template geben muss, mit der man das zu prüfende System vergleichen kann. Dass das nicht der Fall sein wird, ist aber sehr naheliegend, da im Laufe der Zeit Mischungen der einzelnen Versionen entstanden sind. Es sollte aber trotzdem dann der Referenzknoten als Ergebnis gewählt werden, der die höchste strukturelle Ähnlichkeit mit dem aktuellen Stand aufweist, obwohl die darunterliegende Struktur dadurch nicht exakt abgebildet werden kann. Es sind also, wie in Kapitel 2 erklärt, keine automatischen Korrekturmaßnahmen der Struktur erlaubt, um sie dem Template anzupassen. Das Gleiche gilt für die Berechtigungsstruktur auf einem Knoten. Hier ist auch jegliche automatische Korrektur unerwünscht, da die Auswirkungen dieser Änderungen nicht abschätzbar sind. Es ist implementierungstechnisch aber kein Problem die erkannten Abweichungen automatisiert zu beheben bzw. sie über einen Empfehlungsdienst auszugeben.
Die Systemarchitekturen der beiden Ansätze aus Kapitel 3.1 unterscheiden sich sehr stark und dadurch auch die entsprechenden Vorgangsweisen der oben erläuterten Varianten. Weil ich mich gegen die Variante mit einer Datenbank entschieden habe, wird auch nur der zweite Ansatz aus Kapitel 3.1.2 näher beschrieben, da er vollständig umgesetzt wurde und nicht nur auf einem frühen Prototyp basiert. Prinzipiell wird dafür folgendes Lösungsschema angewendet:

Zuerst wird die Musterstruktur erzeugt, damit man überhaupt weiß, welche Knotenmenge für den Vergleich zur Verfügung steht. Dann traversiert man die zu prüfende Struktur und versucht im Template den passenden Knoten für den Unifizierungsprozess zu finden. Der „on-the-Fly" Ansatz arbeitet sich über die Hauptreihenfolge (=Tiefensuche), wie in Abbildung 13 zu sehen ist, vor und versucht rekursiv jeweils die Kindknoten der Wurzel der zu prüfenden Struktur mit den Kindknoten der Wurzel der Musterstruktur abzugleichen. Dieser Abgleich wird ausschließlich über die Namensgebung durchgeführt und ist hinsichtlich anderer Kriterien zwar erweiterbar, aber in den relevanten Szenarien nicht erforderlich bzw. zielführend. Es kommt hier ausschließlich auf eine korrekte Namensgebung und Struktur an und nicht auf semantische Korrektheit der darunterliegenden Informationen. Wenn man einen passenden Referenzknoten im Template gefunden hat, werden die Kindknoten dieses Knotens mit den Kindknoten des Knotens im zu prüfenden System verglichen. Dieser Vorgang wird solange wiederholt, bis man zu einem Referenzknoten im Template gelangt, der keine Kindknoten besitzt. An dieser Stelle wird die Rekursion abgebrochen und arbeitet die Kindknoten eine Ebene höher weiter ab. Es kann auch, wie vorher erwähnt, passieren, dass man bei einer indirekten Unifizierung mehrere mögliche Strukturen als potentiellen Referenzknoten identifiziert. In Abbildung 13 kann, wie vorher in Abbildung 10 erläutert, nicht sofort entschieden werden, ob Knoten 6 oder Knoten 9 im Template zum Knoten 6 der zu prüfenden Struktur passt. Mittels einer geeigneten Gewichtung wird die darunterliegende

Struktur analysiert und ermittelt, welcher Referenzknoten zur gegebenen Struktur passt. Die Wurzel der Struktur, die eine höhere Gesamtgewichtung erzielt, wird als bester Referenzknoten betrachtet und als solcher dann festgelegt. Problematisch ist allerdings, dass es darunterliegend auch noch die Möglichkeit gibt, mehrere potentielle Referenzknoten zu identifizieren. Hierfür wird dieser Algorithmus zur Gewichtung wieder auf die entsprechenden Unterstrukturen angewendet. Die Wurzel der Unterstruktur mit der besten Gewichtung wird als Referenzknoten in der übergeordneten Unterstruktur betrachtet. Falls mehrere Knoten den gleichen Gewichtskoeffizienten liefern, wird nach Definition 10 der bestmögliche gewählt. In Abbildung 13 bekommt der Knoten 6 im Template eine Gewichtung von einem Punkt, da zwei Knoten der geforderten Struktur entsprechen und für einen Knoten kein passender Referenzknoten ermittelt werden konnte. Der Knoten 9 im Template hingegen erreicht drei Punkte als Gewichtung, da die darunterliegende Struktur ident ist. Er erhält für die drei richtigen Kindknoten drei Punkte und da keiner seiner Kindknoten nicht unifiziert werden konnte, auch keine Abzüge. Daraus folgt, dass der Knoten 6 der zu prüfenden Struktur mit dem Knoten 9 der Musterstruktur unifiziert und als Referenzknoten vermerkt wird.

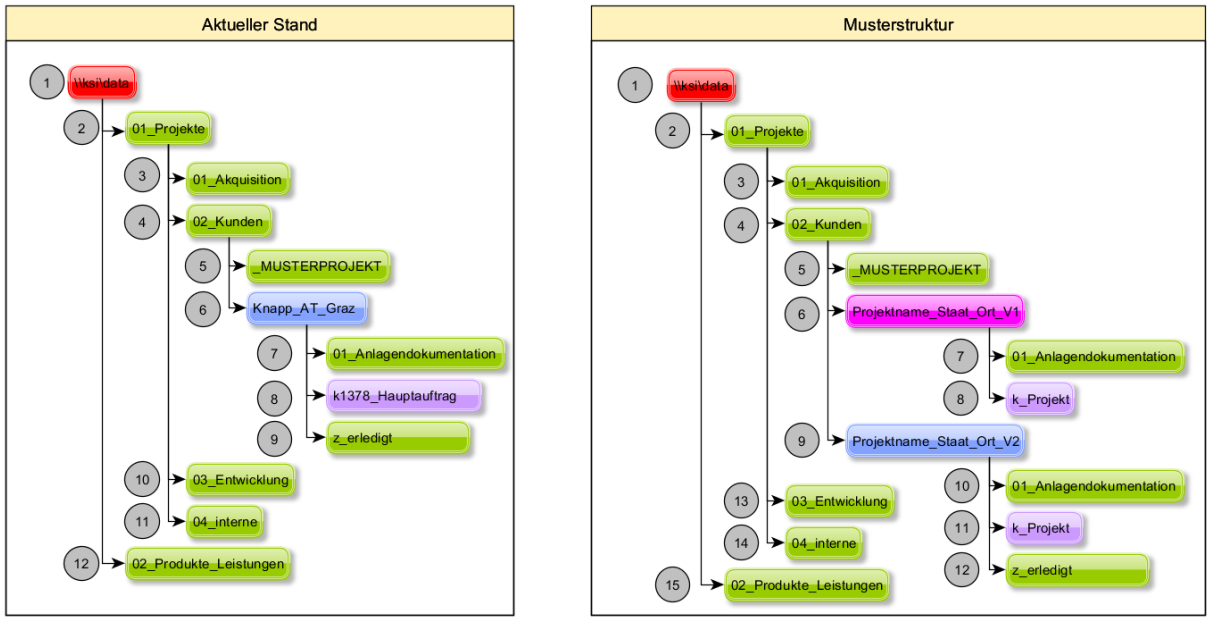


Abbildung 13 Traversierung der beiden Strukturen

Im Idealfall wurde jedem Knoten der zu prüfenden Struktur ein Referenzknoten im Template zugewiesen. Was passiert aber mit den Knoten, die nicht auf korrekte Berechtigungen geprüft werden können, weil sie keinem Referenzknoten zugeordnet sind? Diese müssen dann natürlich als strukturelle Abweichung in einer geeigneten Form maskiert werden, um eine Korrektur des Templates oder des zu prüfende Systems dementsprechend zu ermöglichen.

Die andere Betrachtungsweise seitens der Musterstruktur, ist ein wenig komplizierter aufgebaut. Da besteht die Möglichkeit, dass Knoten im Template mit einer bestimmten Häufigkeit vorkommen müssen. In diesem Fall werden dann zusätzliche Knoten in die Ergebnisstruktur eingefügt, damit man sieht, dass es sich eigentlich um diese bestimmte Struktur handeln müsste, obwohl eine geringfügige Abweichung detektiert wurde. Grundsätzlich ist die Vorgangsweise aus beiden Richtungen betrachtet aber dieselbe:
Falls ein Referenzknoten nicht seiner Häufigkeit entsprechend oft in der zu prüfenden Struktur enthalten ist, muss er als strukturelle Abweichung behandelt werden. Die strukturelle Problematik bzw. die Verifizierung des Auftretens eines Referenzknotens wird in Kapitel 4 „Implementierung" noch genauer erläutert.

3.5 Ermittlung der Berechtigungsunterschiede

Nachdem die passenden Referenzknoten ermittelt wurden, können auch die Berechtigungen problemlos miteinander verglichen werden. Diese beiden Schritte werden getrennt durchgeführt, damit beim Hinzufügen von Struktur- oder Berechtigungsausnahmen nicht mehr das zu prüfende System traversiert werden muss. Es sind nur mehr die Berechtigungsunterschiede bzw. die Struktur zu visualisieren. Das .NET Framework ermöglicht es direkt über die Programmierschnittstelle auf die „Active Directory" Gruppen, Benutzer, NTFS und SharePoint Berechtigungen zuzugreifen und so die Berechtigungsdifferenzen zu ermitteln [**12**] und [**13**]. Hier stellt sich die Frage, wie die vererbten Berechtigungsinformationen mit den direkten Berechtigungen zusammen auf den einzelnen Knoten gespeichert sind bzw. wie man die Differenzen effizient ermitteln kann. Über das .NET Framework ist es möglich die Unterschiede durch Bitvergleiche zu ermitteln, da man diese zusätzlichen Informationen bei den einzelnen Berechtigungen zur Verfügung gestellt bekommt. Abbildung 14 zeigt einen einzelnen Knoten auf dem mehrere Gruppen inklusive deren Berechtigungen liegen. Dafür wurde folgender Lösungsansatz gewählt:

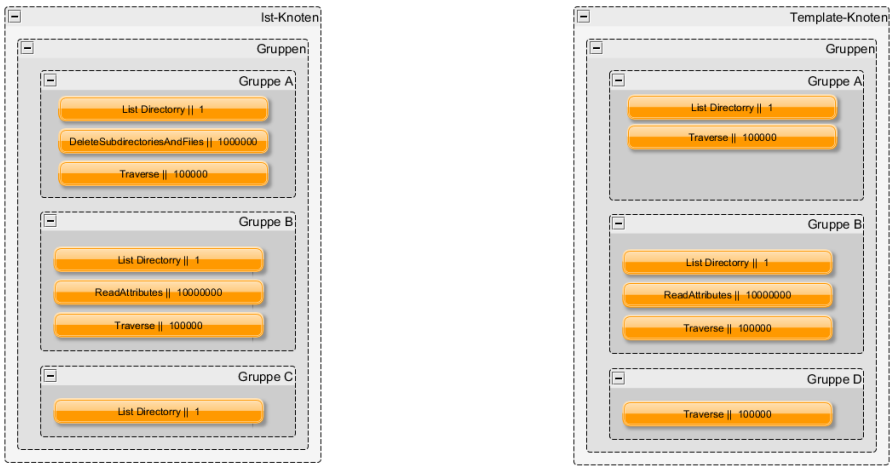


Abbildung 14 Aufbau der Berechtigungen innerhalb eines Knotens

Abbildung 14 zeigt den theoretischen Aufbau eines solchen Knotens. Er besteht aus Benutzergruppen bzw. Benutzern und deren Berechtigungen auf diesem Knoten. Falls zu den direkten Berechtigungen noch vererbte von der gleichen Gruppe hinzukommen, dann werden diese einfach zu den bestehenden Berechtigungen mit logisch ODER hinzugefügt.

Dies ist beim Dateiserver relevant, da vererbte Berechtigungen separat im zu prüfenden Knoten hinterlegt werden, was dazu führt, dass eine Benutzergruppe mehrfach auf einem Knoten gelistet werden kann. Beim SharePoint liegt diese Problematik nicht vor, da die vererbten Berechtigungen mit den knotenspezifischen Berechtigungen schon vorher vom System vereinheitlicht werden.

Mit dieser Methodik sollte es effizient möglich sein, die beiden binären Berechtigungsstrings der jeweiligen Gruppen zu vergleichen. In Abbildung 14 ist bei Gruppe A eine Abweichung festzustellen. Gruppe B ist ident und stellt somit kein Problem dar. Gruppe C ist nur im zu prüfenden System vorhanden, im Template aber nicht definiert und stellt somit eine Differenz dar. Gruppe D ist wie Gruppe C zu behandeln.

3.6 Ausnahmenbehandlung

Es wird natürlich nicht immer möglich sein, die dokumentierte Berechtigungsstruktur einzuhalten. Aus diesem Grund muss eine Ausnahmenbehandlung definiert werden, um eventuell genehmigte Abweichungen nicht als Fehler zu maskieren. Des Weiteren ist es in Zukunft durchaus vorstellbar, diese Abweichungen an eigene Workflows zu binden.

Bei einem Benutzer bestehen beispielsweise Differenzen zwischen den Berechtigungen auf der Musterstruktur und der ermittelten Berechtigungen. Jetzt kann bei der entsprechenden Abteilung die Notwendigkeit dieser Berechtigungen überprüft und dadurch ermittelt werden, ob diese überhaupt benötigt werden. Abhängig vom Ergebnis kann dementsprechend reagiert werden.

Aus diesem Grund wird zwischen den folgenden drei Ausnahmen unterschieden:

- Strukturelle Ausnahmen
- Ausnahmen auf Benutzergruppenebene
- Namensspezifische Ausnahmen (=Wildcard)

Bei strukturellen Ausnahmen handelt es sich um Knoten, die für die Ermittlung der Abweichungen irrelevant sind und somit auch keine Differenz darstellen sollen. Diese Knoten werden einfach übersprungen und deren Berechtigungsstruktur wird nicht ermittelt bzw. überprüft.

Die Ausnahmen auf Benutzergruppenebene betreffen Knoten im zu prüfenden System, die einem passenden Referenzknoten in der Musterstruktur zugeordnet sind. Wenn Berechtigungsdifferenzen ermittelt wurden, die in zukünftigen Ergebnissen nicht mehr als Abweichung gelistet werden sollen, sondern einfach ignoriert werden können, dann ist es sinnvoll so eine Ausnahme zu definieren. Des Weiteren kann man hier dann auch explizite Workflows aktivieren, die in den einzelnen Abteilungen nachfragen, ob diese Berechtigung noch benötigt wird oder zurückgezogen werden kann. Dieser Aspekt wird in Kapitel 6 „Fragen, Schlussfolgerungen und Ausblicke" noch genauer beschrieben.
Bei diesen beiden Typen von Ausnahmen wurde zuerst der Fehler gemacht, dass die Struktur- und Berechtigungsausnahmen auf den entsprechenden Knoten im Template hinterlegt wurden und nicht global in einer Konfigurationsdatei. Diese Vorgangsweise funktioniert problemlos und sehr zeiteffizient bei Referenzknoten, die mittels einer direkten Unifizierung ermittelt werden. Ein großes Problem entsteht aber bei Referenzknoten, die für mehrere Knoten in der zu prüfenden Struktur verantwortlich sind. Falls so ein Referenzknoten eine Ausnahme für diese beiden Arten enthält, dann gilt diese Ausnahme für alle Knoten der zu prüfenden Struktur, die mit diesem Referenzknoten verbunden sind. Somit ist dieser Ansatz für indirekte Unifizierungen gänzlich ungeeignet. Hier darf man nicht den Fehler machen und die Ausnahmendefinition ins Template verlagern, da es dort durch die regulären Ausdrücke auch zu indirekten Unifizierungen kommen kann. Dadurch ist ein Referenzknoten mit mehreren Knoten im zu prüfenden System verbunden. Hier ist es dann nicht mehr möglich für jeden Knoten im aktuellen Stand eine eigene Behandlung zu definieren, da sie alle vom selben Referenzknoten mit Informationen versorgt werden. Abbildung 15 dient dazu, dieses Szenario noch genauer zu beleuchten:

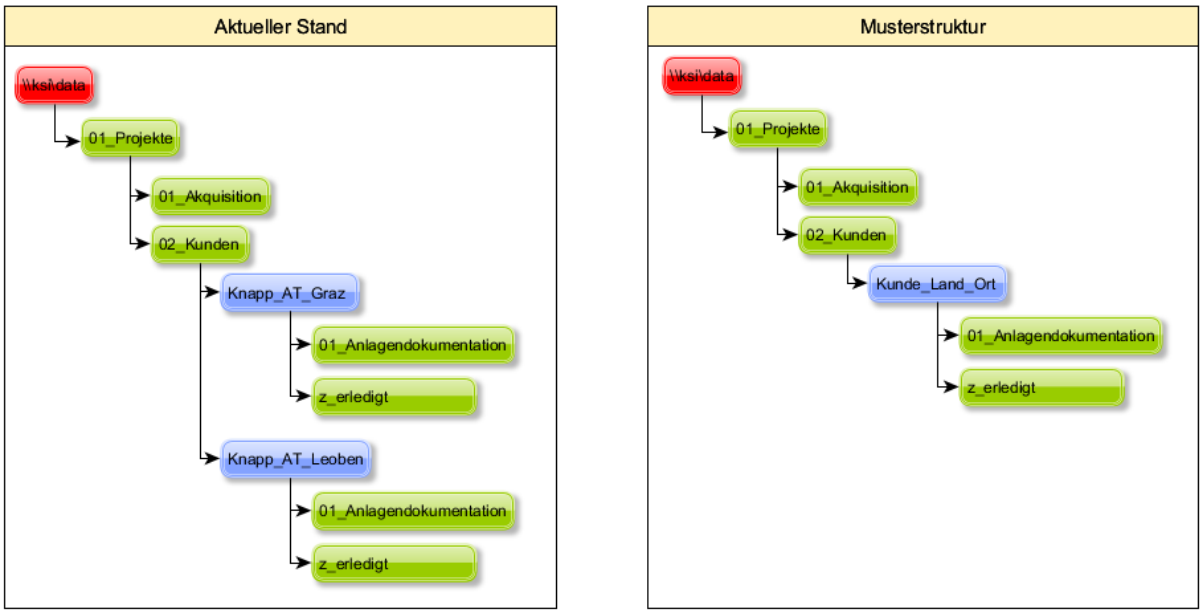


Abbildung 15 Behandlung von Ausnahmen

Wenn man auf der rechten Seite im Template den Knoten „Kunde_Land_Ort" als Ausnahme definiert, dann kann man keinen einzigen Knoten im aktuellen Stand mehr kontrollieren, der diesem Muster entspricht. Wenn der Referenzknoten als Ausnahme festgelegt wird, werden alle Knoten, die diesem Muster entsprechen, ignoriert. Aus diesem Grund wurde in weiterer Folge auch eine globale Lösung angestrebt, die für jeden der drei Berechtigungstypen eine eigene Konfigurationsdatei vorsieht, die diese Ausnahmen enthält.

Die namensspezifischen Ausnahmen sind beim SharePoint besonders interessant, da man dort relativ viele Knoten zur Prüfung erhält, die zum System gehören und nicht von den Administratoren oder Benutzern angelegt werden. Wenn man diese Knoten nicht im Template definiert, dann werden alle ohne diese Ausnahmenregelung als Abweichungen gewertet. Sie kommen im zu prüfenden System vor, sind aber im Template nicht definiert. Diese Vorgangsweise ist grundsätzlich auch erforderlich, nur in diesem Fall macht es keinen Sinn diese Systemknoten explizit definieren zu müssen, da man keinen Einfluss auf ihre Namen oder Berechtigungen im zu prüfenden System hat. Diese müssen dann nicht mit vielen strukturellen Ausnahmen von einer weiteren Prüfung ausgeschlossen werden, sondern können einfach anhand ihres Namens global von allen Überprüfungen ausgenommen werden.

Bei der Definition dieser drei Ausnahmetypen habe ich mich wieder, wie bei der Erstellung des Templates, für eine XML Struktur entschieden. Es stellt eine einheitliche Lösung für alle möglichen Ausnahmeregelungen dar. Durch die Kapselung der einzelnen Ausnahmen, lässt sich dieses System auch bei späteren Änderungen gut erweitern.

3.7 Kurze Beschreibung des Algorithmus

1. Erstellung des Templates. Entweder manuell oder semiautomatisch.
2. Die zu prüfende Baumstruktur wird rekursiv mittels Tiefensuche traversiert und es werden passende Referenzknoten ermittelt.
    a. Bei indirekter Unifizierung kann es vorkommen, dass mehrere Knoten als Referenzknoten in Frage kommen. Hier wird der Knoten gewählt, der die höchste Gewichtung der darunterliegenden Struktur hat.
3. Nach Schritt 2 hat jeder Knoten des zu prüfenden Systems, der einen Referenzknoten zugeteilt bekommen hat, Zugriff auf dessen definierte Berechtigungen.
4. Vergleich der Berechtigungen auf den einzelnen Knoten mit ihren Musterberechtigungen.
5. Ausgabe der Resultate. Strukturelle Abweichungen müssen klar ersichtlich sein. Unterschiede zwischen den tatsächlichen und den definierten Berechtigungen müssen in geeigneter Form repräsentiert werden. Ausnahmen sind für eine spätere Revidierung noch leicht erkennbar.

3.8 Visualisierung der Ergebnisse

Es ist sinnvoll die Resultate so zu visualisieren, wie sie in der zu prüfenden Struktur ursprünglich dargestellt wurden. In den meisten Fällen wird das, wie auch in den beiden hier vorgestellten Szenarien, in Form eines Treeviews erfolgt sein. Das bedeutet, dass man wieder eine Baumstruktur sowie verschiedene Felder zur Visualisierung der Eigenschaften eines gewählten Knotens benötigt. Eine passende Visualisierung der Berechtigungsdifferenzen auf den einzelnen Knoten ist auch erforderlich. Wenn große Strukturen zu prüfen sind, die sich beispielweise aus mehreren tausend Knoten zusammensetzen und nur ein relativ schlechtes Template zur Verfügung steht, dann wird das Ergebnis sehr viele strukturelle Fehler beinhalten. Dadurch fällt es relativ schwer, sich einen Überblick zu verschaffen. Hier muss es in irgendeiner Form möglich sein, alle Fehler systematisch durchzugehen, um die Fehlerursache zu ermitteln.
Für diese Problematik werden alle Fehler auf einer Liste gespeichert. Auf dieser Liste befinden sich dann strukturelle Abweichungen, Berechtigungsdifferenzen sowie die definierten Ausnahmen. Diese verschiedenen Fehlertypen werden dann getrennt über ein Dropdownmenü angesteuert. Dadurch kann man zwischen den einzelnen Fehlertypen klar trennen und nur die zu diesem Zeitpunkt relevanten Fehler in einem weiteren Dropdownmenü auflisten und dadurch leichter überprüfen.
Interessante Aspekte sind die Komplexität des Speicherbedarfs zur Visualisierung der Ergebnisstruktur in Form eines Baumes und die zeitliche Komplexität beim Laden dieser Struktur und der Fehlerliste.
Für die Visualisierung von großen zu überprüfenden Strukturen wird eine besonders schlanke Objektstruktur benötigt. Sonst kann man die einzelnen Knoten nur bei Bedarf nachladen und muss dafür die Ergebnisstruktur schon vorbereiten, da sonst auf einer oberen Ebene nicht feststellbar ist, ob in einem darunterliegenden Level verschiedene Fehler aufgetreten sind. Da es sich hier um eine Visualisierung der Ergebnisse handelt, darf die Ladezeit natürlich auch nicht zu groß sein und außerdem muss sie durch irgendeine Form von Fortschritt immer visualisiert werden. Die graphische Oberfläche darf beispielsweise nicht einfach für zwei Minuten einfrieren und dem Benutzer einen Applikationsabsturz suggerieren. Die zeitliche Komplexität ist aus Multithreading Sicht aber ein durchaus lösbares Problem.

Ein Prototyp zur besseren Vorstellung der Oberfläche ist in Abbildung 16 zu sehen.

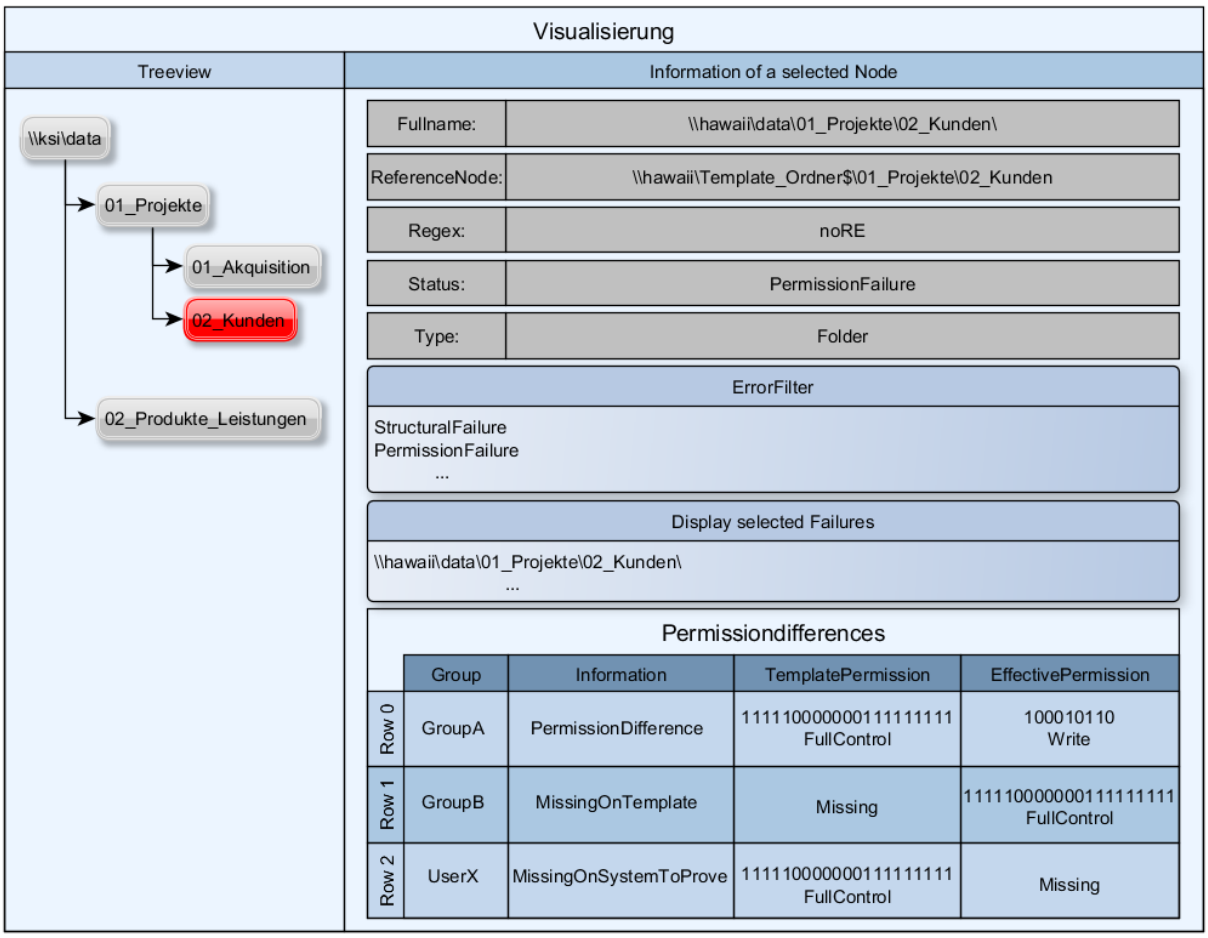


Abbildung 16 Prototyp der Visualisierung

Für den Vergleich der einzelnen Resultate gibt es einen entsprechenden Report, der folgende Kennzahlen beinhaltet:

- Anzahl der überprüften Knoten
- Korrekte Knoten
- Anzahl der fehlerhaften Knoten
  - Knoten mit einer Wildcard
  - Ignorierte Knoten
  - Strukturelle Fehler
  - Fehlerhafte Berechtigungen
  - Fehlerhafter „Quantifier"

Reports

Abteilungen on 24.11.2011 at 08:35

Path: \\ksi\data\05_Abteilungen
Template: C:\Users\sed___DA\xml\fsTemplate_05_Abteilungen.xml

Type	Amount	Percentage
Amount of nodes	56	
Correct nodes	49	87,50%
Failed nodes	7	12,50%

Failed nodes	Amount	Percentage
Wildcarded nodes	0	0,00%
Ignored nodes	0	0,00%
Structural failures	7	12,50%
Permission failures	0	0,00%
Quantifier failures	0	0,00%

KSI-Data on 23.11.2011 at 15:20

Path: \\ksi\data
Template: C:\Users\sed___DA\xml\fsTemplate.xml

Type	Amount	Percentage
Amount of nodes	104369	
Correct nodes	70470	67,52%
Failed nodes	33899	32,48%

Failed nodes	Amount	Percentage
Wildcarded nodes	0	0,00%
Ignored nodes	2	0,00%
Structural failures	4515	4,33%
Permission failures	29268	28,04%
Quantifier failures	114	0,11%

Abbildung 17 Resultate zweier verschiedener Durchläufe

Abbildung 17 zeigt beispielsweise zwei solche Ergebnisse. Dadurch ist es möglich die einzelnen Reports in Relation zu einander zu betrachten. Wenn man das gleiche System zu zwei verschiedenen Zeitpunkten überprüft, lässt sich die Entwicklung des Systems leichter verfolgen.

4 Implementierung

4.1 Definition der Musterstruktur mittels XML

Abbildung 18 zeigt eine graphische Repräsentation des Schemas für den Dateiserver:

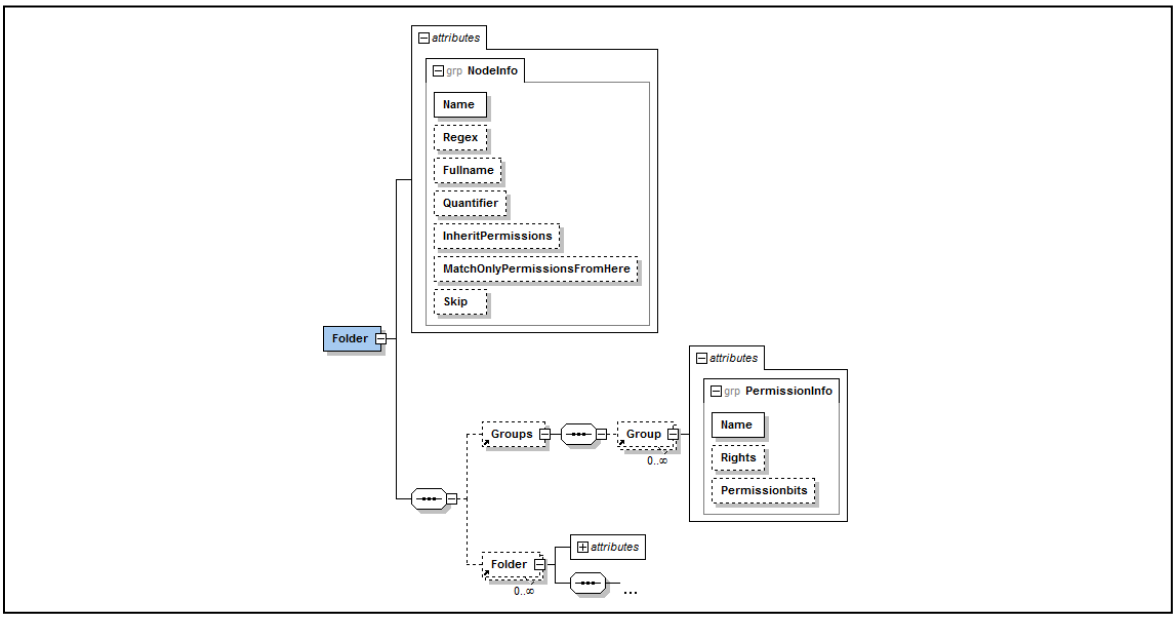


Abbildung 18 XML Schema für die Verifizierung des Dateiservertemplates

Folder dient als Wurzelknoten und hat die folgenden Attribute:
Name, Regex, Fullname, Quantifier, InheritPermissions, MatchOnlyPermissionsFromHere und Skip

4.1.1 Beschreibung der Attribute

Name: Repräsentiert den Namen des Knotens.
Regex: Hier kann man einen regulären Ausdruck angeben, der für eine indirekte Übereinstimmung verwendet werden soll.
Fullname: Dieses Attribut ist optional und gibt den absoluten Pfad zu diesem Knoten an.
Quantifier: Mit dem Quantifier kann man angeben, wie oft dieser Knoten als Referenzknoten verwendet werden darf. Standardmäßig ist bei direkten Übereinstimmungen der Wert 1 festgelegt und für indirekte Vergleiche dient der * als Standardwert. Folgende Einstellungen sind möglich:

- 0…n (Integer): Gibt an, dass der Knoten 0 bis n mal als Referenzknoten fungieren darf.
- + bzw. *: siehe Kapitel 2.5 Reguläre Ausdrücke.

InheritPermissions: Durch dieses Attribut ist es einem Kindknoten möglich die Berechtigungen des übergeordneten Elternknotens zu übernehmen.

MatchOnlyPermissionsFromHere: Mit diesem Attribut hat man die Möglichkeit, dass auf einen strukturellen Vergleich verzichtet wird. Die Berechtigungen auf den folgenden Kindknoten werden aber weiterhin überprüft und mit dem letzten, im Template definierten Knoten, verglichen.
Skip: Ermöglicht es diesen Knoten und dessen untergeordnete Kindknoten von der Überprüfung auszuschließen.

Ein Folder darf dementsprechend beliebig viele weitere Folderknoten und einen Knoten Groups enthalten, der wiederum eine beliebige Anzahl vom Typ Group enthalten kann. Ein solcher Knoten Group ist aus folgenden drei Attributen aufgebaut:

- Name: Repräsentiert den Gruppennamen.
- Rights: Gibt an, welche Berechtigungen dieser Gruppe zugeordnet sind.
- Permissionbits: Ist eine Kombination mit logisch ODER der einzelnen binären Berechtigungsstrings, die der Gruppe zugeordnet sind.

Abbildung 19 zeigt eine einfache textbasierte Definition des Schemas für den Dateiserver, das aber einzelne Erweiterungen für andere Musterstrukturen nur schwer zulässt:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema elementFormDefault="qualified">
  <!-- Group und Groups -->
  <xsd:element name="Group">
    <xsd:complexType> <xsd:attributeGroup ref="PermissionInfo"/> </xsd:complexType>
  </xsd:element>
  <xsd:element name="Groups"> <xsd:complexType> <xsd:sequence> <xsd:element ref="Group" minOccurs="0"
maxOccurs="unbounded"/> </xsd:sequence> </xsd:complexType> </xsd:element>
  <!-- Types -->
  <xsd:simpleType name="MyBinary">
    <xsd:restriction base="xsd:string"> <xsd:pattern value="(0|1)+"/> <xsd:maxLength value="63"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="MyQuantifier">
    <xsd:restriction base="xsd:string"> <xsd:pattern value="((\d+)|\*|\+)"/> </xsd:restriction>
  </xsd:simpleType>
  <!-- Attributes-->
  <xsd:attributeGroup name="NodeInfo">
    <xsd:attribute name="Name" type="xsd:string" use="required"/>
    <xsd:attribute name="Regex" type="xsd:string" use="optional"/>
    <xsd:attribute name="Fullname" type="xsd:string" use="optional"/>
    <xsd:attribute name="Quantifier" type="MyQuantifier" use="optional"/>
    <xsd:attribute name="InheritPermissions" type="xsd:boolean" use="optional"/>
    <xsd:attribute name="MatchOnlyPermissionsFromHere" type="xsd:boolean" use="optional"/>
    <xsd:attribute name="Skip" type="xsd:boolean" use="optional"/>
  </xsd:attributeGroup>
  <xsd:attributeGroup name="PermissionInfo">
    <xsd:attribute name="Name" type="xsd:string" use="required"/>
    <xsd:attribute name="Rights" type="xsd:string" use="optional"/>
    <xsd:attribute name="Permissionbits" type="MyBinary" use="optional"/>
  </xsd:attributeGroup>
  <!--Structure-->
  <xsd:element name="Folder">
    <xsd:complexType>
      <xsd:sequence> <xsd:element ref="Groups" minOccurs="0" maxOccurs="1"/>
        <xsd:element ref="Folder" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attributeGroup ref="NodeInfo"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Abbildung 19 Textbasierte Definition des XML Schemas für den Dateiserver

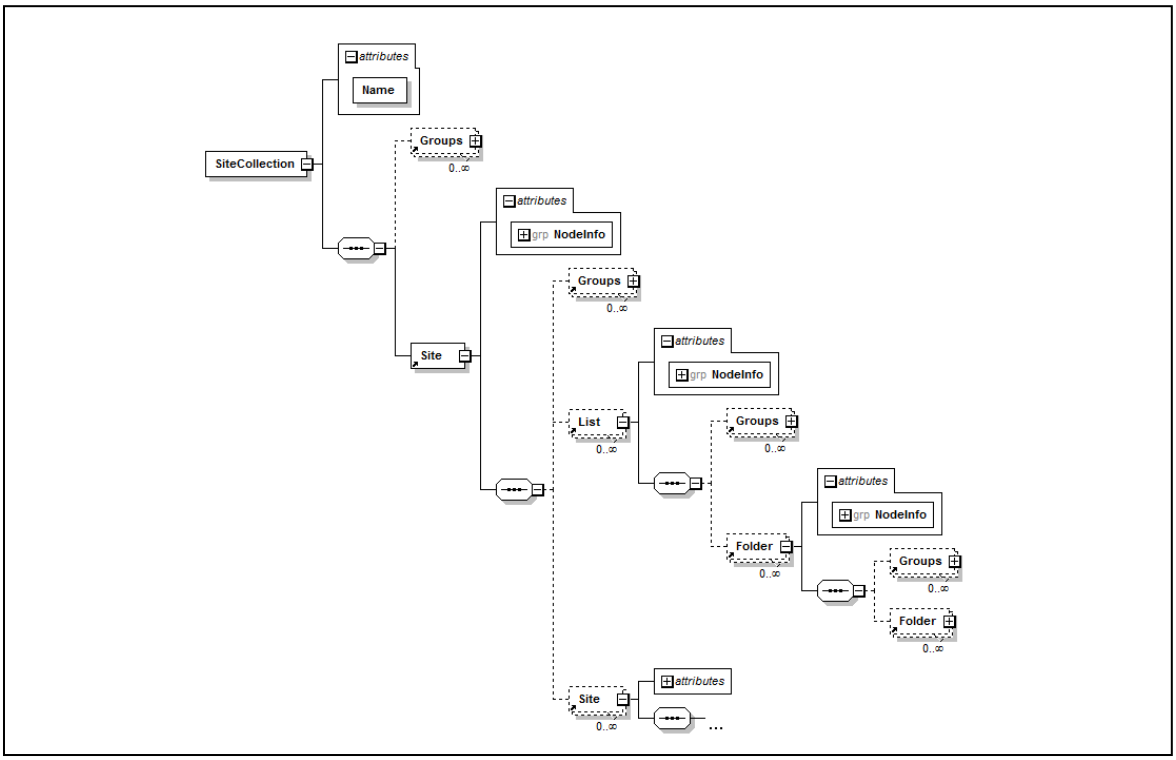


Abbildung 20 XML Schema für die Verifizierung des SharePoints

Abbildung 20 zeigt eine Definition des Schemas für den Microsoft SharePoint. Der oben beschriebene Knoten „Folder" ist erst in einer sehr tiefen Ebene vorhanden, da er nur in Listen enthalten sein kann. Bei einem Folder am SharePoint kann es sich um einen Ordner handeln, der sich in einer Dokumentbibliothek oder Liste befindet. Diese sind einer konkreten Webseite untergeordnet, die wiederum einer anderen Webseite oder direkt der Sitecollection untergeordnet ist. Die Attribute sind die gleichen wie beim Schema des Dateiservers. Wie man am Beispiel des Dateiservers gut erkennen kann, ist die Definition eines solchen Schemas relativ intuitiv und kann durch XML beschrieben werden. Wenn man dieses Schema mit einer DTD beschreibt, sieht es folgendermaßen aus (siehe Abbildung 21):

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Group EMPTY>
<!ATTLIST Group
   Name CDATA #REQUIRED
   Rights CDATA #IMPLIED
   Permissionbits CDATA #IMPLIED>

<!ELEMENT Groups (Group*)>

<!ELEMENT Folder (Groups?, Folder*)>
<!ATTLIST Folder
   Name CDATA #REQUIRED
   Regex CDATA #IMPLIED
   Fullname CDATA #IMPLIED
   Quantifier CDATA #IMPLIED
   InheritPermissions NMTOKEN #IMPLIED
   MatchOnlyPermissionsFromHere NMTOKEN #IMPLIED
   Skip NMTOKEN #IMPLIED
>
```

Abbildung 21 DTD für das Dateiservertemplate

Durch diese Gegenüberstellung kann man schon sehr leicht erkennen, dass es sich um zwei grundlegend verschiedene Techniken zur Schemaverifizierung handelt. Das XML Schema ist sehr intuitiv und da das zu verifizierende Dokument auch mit XML beschrieben wird, ist es sehr leicht nachzuvollziehen. DTD beruht auf SGML, wird auch damit beschrieben und ist deshalb zu Beginn schwieriger zu verstehen. Vor allem ist die Bedeutung der einzelnen Bezeichnungen nicht so klar ausgeprägt wie bei einem XML Schema. Die Mächtigkeit von DTD ist aber weiterhin unbestritten, da früher alle Webseiten Instanzen einer einzigen DTD Klasse für HTML waren. Des Weiteren ist es in der XML 1.0 Spezifikation vorgesehen, dass XML Dokumente mit einem entsprechenden DTD überprüft werden müssen. Somit ist eine Verifizierung der XML Dokumente generell vorgesehen, um die Spezifikation zu erfüllen. Im weiteren Verlauf wurde DTD aber vom XML Schema abgelöst. Es ermöglicht unter anderem die genaue Angabe von Datentypen, die für Elemente und Attribute erlaubt sind. Man kann sich auch, wie im oben abgebildeten Schema, selbst eigene Datentypen definieren, die gewisse Einschränkungen der Eingaben zur Verfügung stellen können. Das ist ein enormer Vorteil für die spätere Weiterverarbeitung dieser XML Dokumente, da die Verifizierung der einzelnen Attributswerte schon vom Schema übernommen wird und nicht mühsam beim Einlesen der Daten durchgeführt werden muss. Bei einem DTD hat man nicht die Möglichkeit zwischen Datentypen zu unterscheiden. Es kann also nicht im Datenmodell spezifiziert werden, ob es sich um ein Datum, einen Text oder eine Uhrzeit handelt. Des Weiteren hat man beim XML Schema auch die Möglichkeit einen erlaubten Wertebereich für die gewünschten Eingaben zu definieren. Der größte Vorteil ist aus meiner Sicht aber noch immer, dass man XML Schema mittels XML definieren und über das DOM (=Document Object Model) ansprechen kann. Dieser und der folgende Teil sind in Anlehnung an [**6**], [**8**], [**9**], [**10**], [**14**] und [**15**] aufgebaut.

Grundsätzlich ist die Variante aus Abbildung 19 ausreichend. Da aber im weiteren Verlauf noch die Möglichkeit besteht zusätzliche Templates einzubinden, die auch verifiziert werden müssen, ist es auch hier erforderlich Redundanzen zu vermeiden. Diverse Knoten wie etwa: Folder, Groups, Group und PermissionInfo werden in wahrscheinlich vielen weiteren Vorlagen auch verwendet werden. Das ist am vorherigen Beispiel des Microsoft SharePoints ebenfalls schon ersichtlich und lässt mich zu dem Schluss kommen, dass ein solcher Aufbau der Schemata für diese Zwecke nicht ideal ist.

Bei der Definition des Schemas standen folgende drei Modelle zur Auswahl:

- Matrjoschka-Design („Russion Doll")
- Stufenmodell („Salami Slice")
- Venezianischer Spiegel („Venetian Blind")

Beim „Matrjoschka-Design" werden alle Elemente wie bei einer russischen Matrjoschka ineinander weiter aufgebaut. Es ist eine sehr einfache Designvariante, die keine gute Wiederverwendbarkeit der einzelnen Blöcke bereitstellt. Die eigenen Datentypen müssen teilweise neu definiert werden, da sie nicht global zugänglich sind. Dafür entstehen bei Änderungen keine Seiteneffekte, da keine Wiederverwendung von anderen Elementen stattfindet.

Beim Stufenmodell, das ich zuerst verwendet habe, werden die verwendeten Elemente und Attribute global definiert. Diese Elemente können über Referenzen dann auch aufeinander zugreifen. Der Vorteil liegt aus meiner Sicht darin, dass die Struktur des Schemas klarer erhalten bleibt und nicht so stark durch sich öffnende und schließende Tags beeinträchtigt wird. Problematisch ist aber, dass man durch die hohe Anzahl an globalen Elementen sehr viele potentielle Wurzelelemente hat.

Deswegen habe ich in weiterer Folge mit dem „Venezianischen Spiegel" als Pattern gearbeitet. In Kombination mit Inklusionen, wie in modernen Sprachen üblich, ist es ein Ansatz, der maximale Wiederverwendbarkeit der einzelnen Elemente verspricht. Dabei werden die einzelnen Komponenten auf verschiedene Datentypen in einzelnen Dateien aufgeteilt, die dann von den verschiedenen Schemata der Module mittels Inklusion referenziert werden können. Das macht die verschiedenen Schemata noch leserlicher und verhindert redundante Datendefinitionen in den jeweiligen Vorlagen. Das daraus entstandene endgültige Modell basiert auf dem Pattern des „Venezianischen Spiegels" und ist in Abbildung 23 zu sehen. Die verschiedenen Knotentypen (Folder, Site, List,…) sind jetzt jeweils in eigenen Datentypen gekapselt. Dadurch ist das Hinzufügen neuer Knotentypen leicht lösbar und innerhalb der einzelnen Knotentypen kann immer auf das gleiche Element „Groups" verwiesen werden. Hier kommt auch wieder ein Vorteil von XML gegenüber DTD zum Tragen, da eigene Datentypen definiert werden können. Für die Überprüfung des „Quantifier" Attributs wird ein eigener Datentyp definiert, der in Abbildung 22 abgebildet ist. Damit reglementiert man die erlaubten Zeichen in der Zeichenkette und gestattet nur Ganzzahlen, „*" oder „+".

```
<xsd:simpleType name="MyQuantifier">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="((\d+)|\*|\+)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 22 „Quantifier" Datentyp

Analog zu Abbildung 22 ist auch der Datentyp „MyBinary" aus Abbildung 19 für die Überprüfung der binären Berechtigungsstrings aufgebaut.

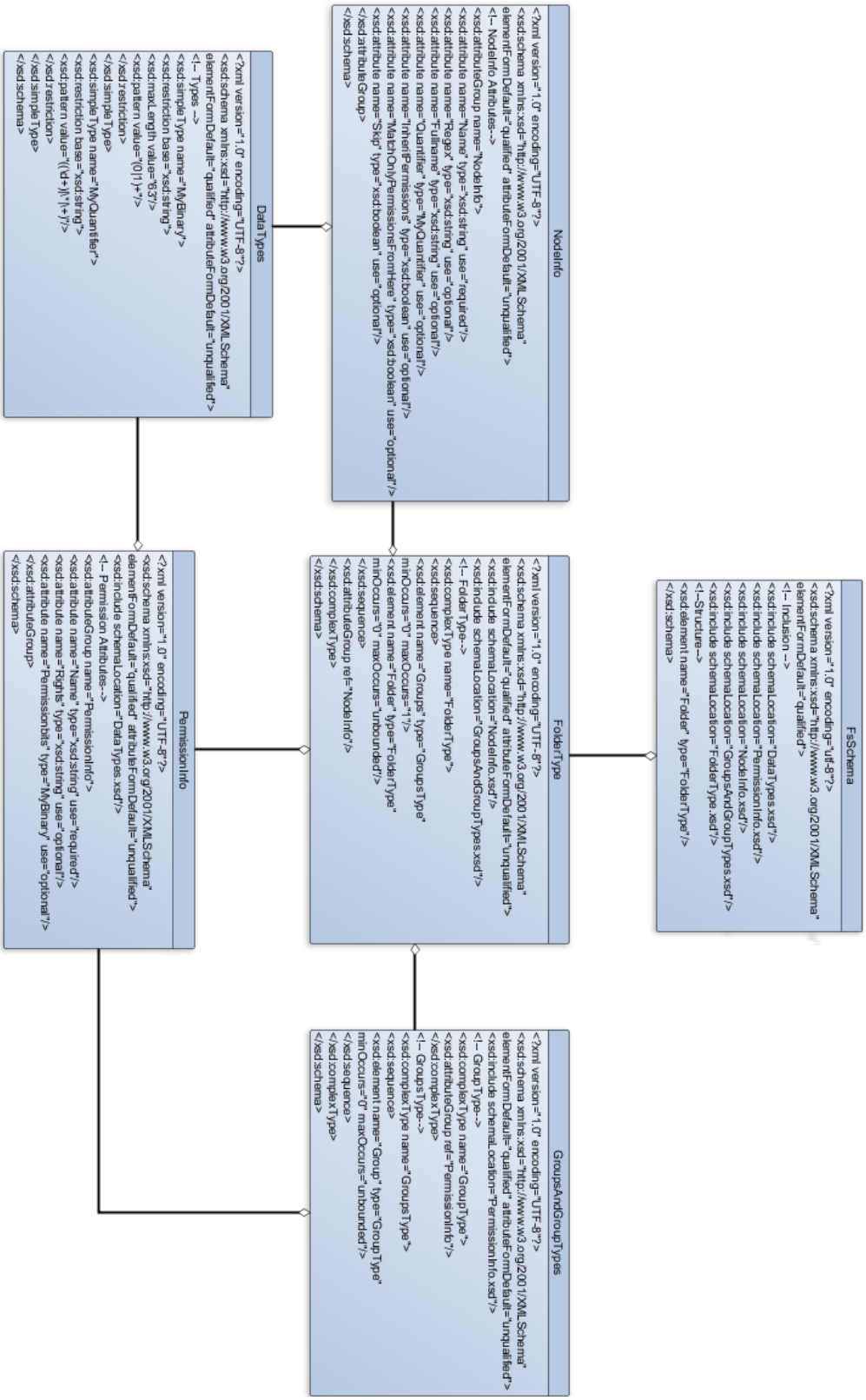


Abbildung 23 Endgültiges XML Schema für das Dateiservertemplate

4.2 Ermittlung der passenden Referenzknoten und Verifizierung

Zu Beginn wird der Wurzelknoten der Musterstruktur als Referenzknoten der zu prüfenden Struktur hinzugefügt. Dadurch ist es möglich, die einzelnen Kindknoten der beiden verknüpften Knoten miteinander zu vergleichen. Dann wird eine Rekursion gestartet, die das zu prüfende System mittels Tiefensuche durchläuft. Eine stark vereinfachte Systemübersicht ist in Abbildung 24 zu sehen.

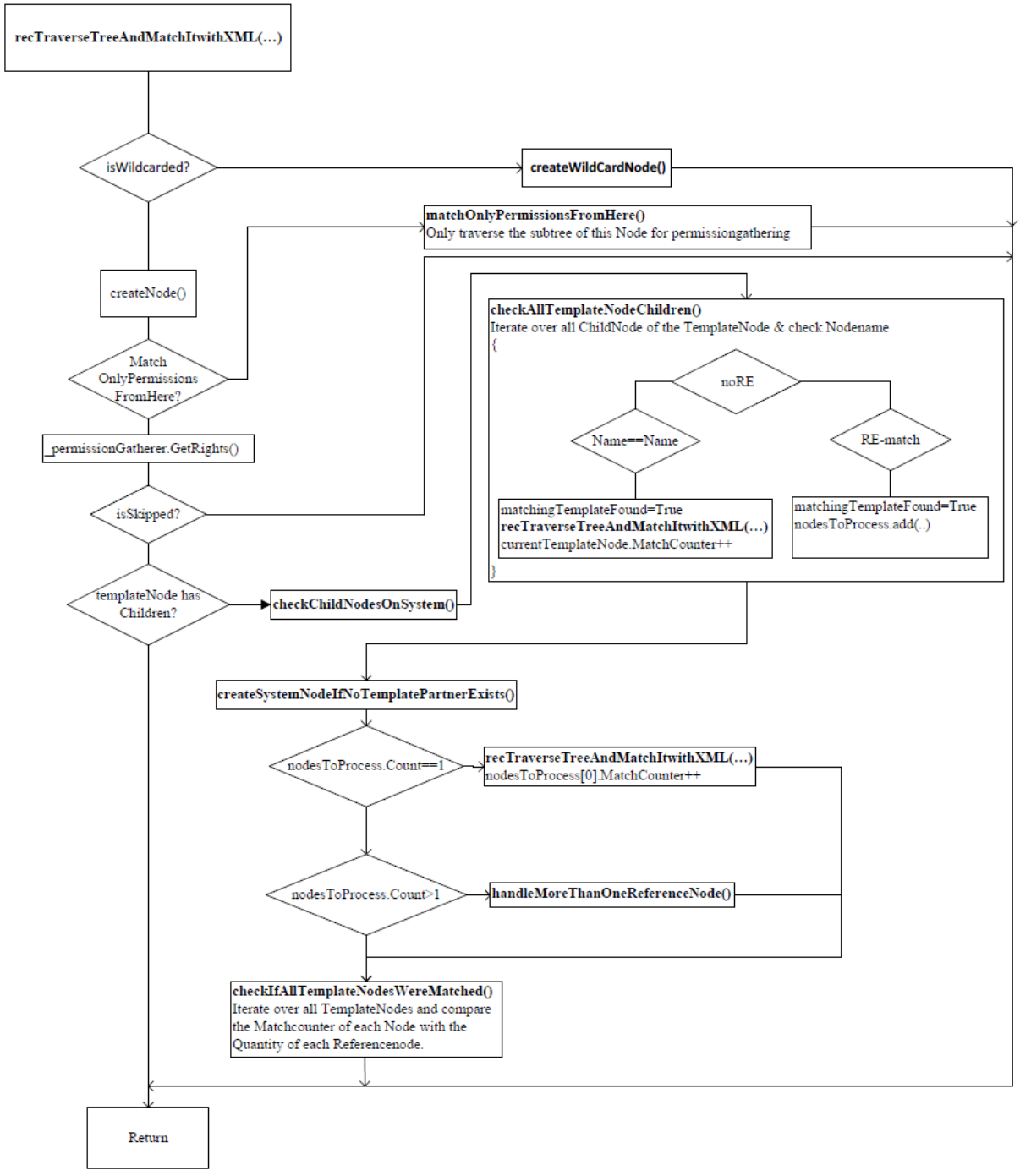


Abbildung 24 Vereinfachtes Flussdiagramm zur Ermittlung des Referenzknotens

Zuerst wird überprüft, ob sich der Knoten auf der Wildcardliste befindet. Falls das der Fall ist, wird er ohne überprüft zu werden direkt in die Ergebnisstruktur übernommen. Sonst wird ein neuer Knoten erzeugt, da man zu Beginn mit einem gültigen Knoten (der Wurzel) gestartet ist. Dadurch wird die Wurzel der Ergebnisstruktur definiert.

Danach wird geprüft, ob das Attribut „MatchOnlyPermissionsFromHere" gesetzt ist. Wenn dieses Flag aktiv ist, werden allen darunterliegenden Knoten, inklusive deren Berechtigungen, in die Ergebnisstruktur eingefügt. Als Referenzknoten wird der Knoten genommen, der diesen Vorgang ausgelöst hat, also der mit dem gesetzten „MatchOnlyPermissionsFromHere" Attribut.
Sonst wird fortgefahren und die Berechtigungen dieses Systemknotens werden noch in die Ergebnisstruktur für einen späteren Vergleich hinzugefügt.

Mittels des „Skip" Attributs ist es möglich eine darunterliegende Struktur zu ignorieren und dadurch die Rekursion abzubrechen.
Falls dieses Attribut nicht gesetzt ist und der Musterknoten weitere Kindknoten enthält, müssen diese natürlich weiter überprüft werden. Dabei iteriert man verschachtelt über alle Kindknoten des zu prüfenden Systems und die Kindknoten des Musterknotens. Diese Vorgehensweise ist erforderlich, um alle potentiellen Abweichungen zu detektieren. An dieser Stelle wird zwischen direkten und indirekten Unifizierungen unterschieden. Bei einer direkten Übereinstimmung wird die Rekursion sofort eine Ebene tiefer fortgesetzt. Bei indirekten Unifizierungen wird hingegen der mögliche Referenzknoten auf einer Liste als möglicher Kandidat vorgemerkt. Falls zu einem Systemknoten überhaupt kein potentieller Referenzknoten gefunden wurde, muss ein Platzhalter erzeugt werden, der den Knoten als strukturelle Abweichung in der Ergebnisliste vermerkt. Sonst erfolgt im Falle einer indirekten Unifizierung eine Abfrage, ob sich einer bzw. mehrere mögliche Kandidaten auf dieser Liste befinden. Wenn nur ein möglicher Kandidat gefunden wurde, dann wird dieser automatisch als bester identifiziert und die Rekursion wird mit diesem Knoten fortgesetzt. Falls sich mehrere Knoten auf dieser Liste befinden, dann wird die entsprechende Methode „handleMoreThanOneReferenceNode()" aufgerufen, die diesen Fall behandeln soll. Wie diese Situation zu behandeln ist, werde ich aber erst ein wenig später in Kapitel 4.2.1 „Methodik für mehrere mögliche Referenzknoten" erläutern, da es dafür eine relativ umfangreiche Methodik gibt.

Am Ende eines Rekursionsschritts, wenn alle Kindknoten des Rekursionsschritts abgearbeitet wurden, muss noch überprüft werden, ob alle Knoten der Musterstruktur ihrem „Quantifier" entsprechend oft als Referenzknoten fungieren. Dabei können auch mögliche Abweichungen gefunden werden. Es wird über alle Kindknoten des aktuellen Musterknotens iteriert und überprüft, ob der „Matchcounter" bei direkten Knoten der Zahl eins entspricht (siehe Abbildung 25).

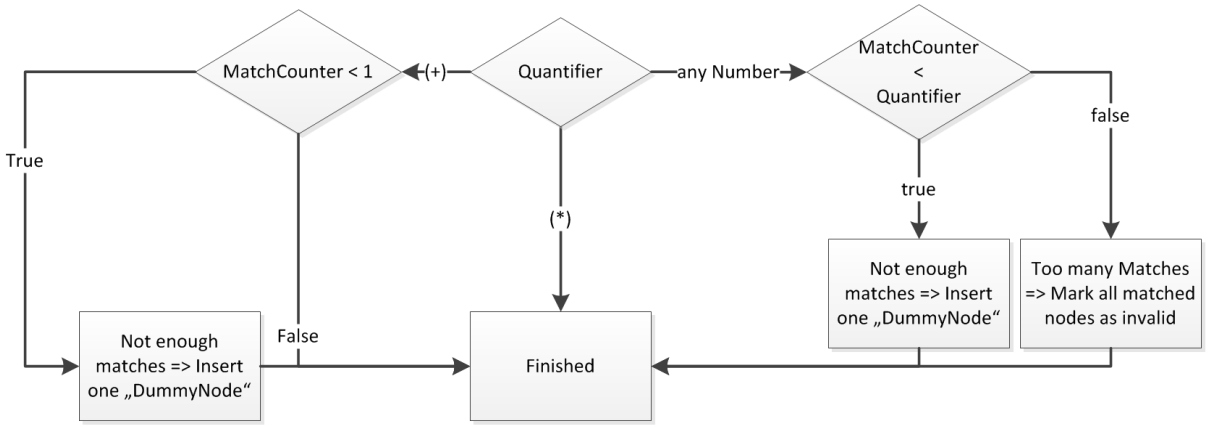


Abbildung 25 Auswertung des „Quantifiers“

Bei Knoten, die über einen regulären Ausdruck unifiziert werden müssen, wird das Attribut „Quantifier“ ausgewertet. Wenn der „Quantifier“ einem „*“ entspricht, ist es irrelevant ob der Musterknoten überhaupt als Referenzknoten fungiert oder nicht. Bei einem „+“ muss nur überprüft werden, ob der Matchcounter < 1 ist, denn dann muss ein zusätzlicher Knoten in die Ergebnisstruktur eingefügt werden, da der Knoten nicht oft genug als Referenzknoten fungieren konnte. Falls der „Quantifier“ einer Zahl entspricht, wird unterschieden, ob der Knoten zu häufig oder nicht ausreichend oft gematcht wurde. Bei zu geringem Matchcounter muss ein Knoten in die Ergebnisstruktur eingefügt werden. Dieser symbolisiert, dass der Referenzknoten nicht seiner Anzahl entsprechend oft gematcht wurde. Ein relativ großes Problem stellt ein Knoten dar, der einen zu hohen Matchcounter aufweist. Da alle Ergebnisknoten auf einer Liste gespeichert werden und nicht direkt in einer Baumstruktur, muss man in der gesamten Ergebnisliste nachprüfen, welche Knoten mit diesem regulären Ausdruck indirekt unifiziert wurden.

Zu Beginn der Arbeit habe ich die Ergebnisknoten nicht in einer Liste gespeichert, sondern mit ihnen direkt einen Baum aufgebaut. Dieser Ansatz ist sehr naheliegend, da man in weiterer Folge sowieso einen Baum für die Visualisierung benötigt. Problematisch war aber die Speicherung dieses Ergebnisbaumes, da jeder Knoten weitere Kindknoten enthält und diese Kindknoten wieder Referenzen auf ihre entsprechenden Kinder enthalten. Das führt bei der Serialisierung zu einem Problem mit zirkulären Referenzen. Jedes Objekt wird serialisiert und mit allen Referenzen in der Datei abgebildet. Das benötigt relativ viel Speicherplatz, da jeder Knoten seine Kindknoten mitspeichern muss. Deswegen habe ich mich hier auch für eine möglichst schlanke Klassenstruktur entschieden, um dieser Problematik entgegen zu wirken. (In Kapitel 6 „Fragen, Schlussfolgerungen und Ausblicke“ wird eine effizientere Lösung vorgeschlagen, bei der direkt auf der Baumstruktur gearbeitet werden kann.)

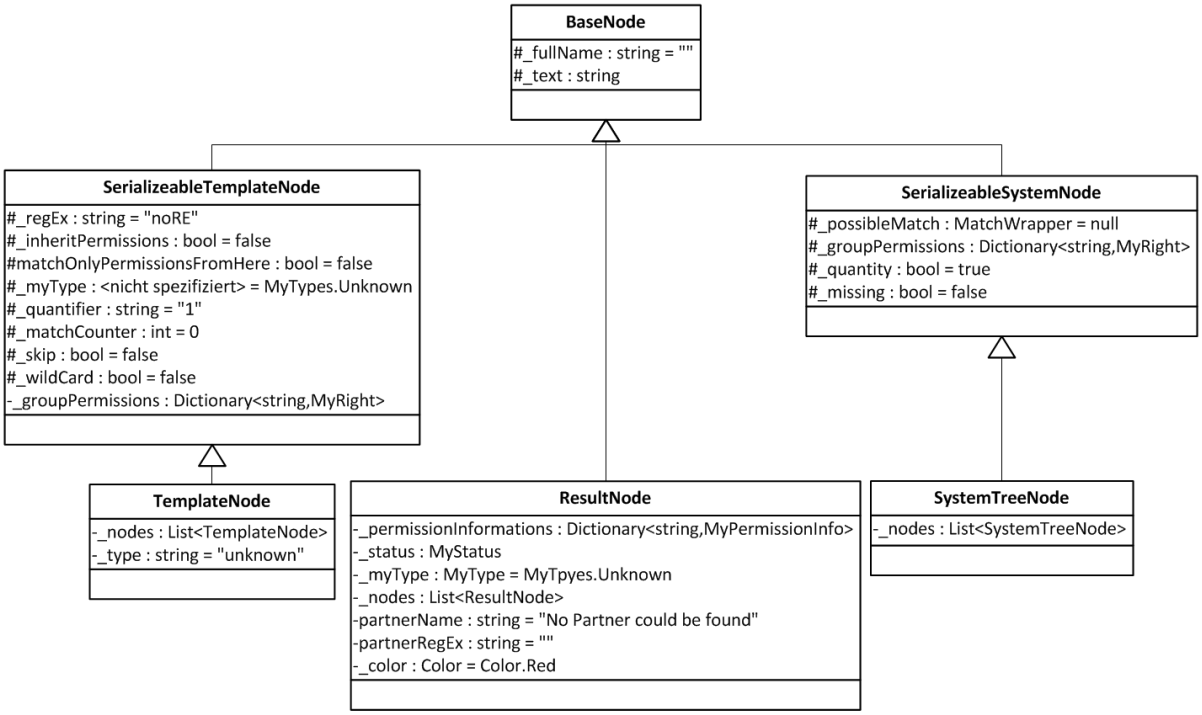


Abbildung 26 Klassenstruktur der Knoten

Um das XML Template in einer geeigneten Objektstruktur zu halten, wird der Knotentyp TemplateNode benötigt. Bei der Suche nach dem idealen Referenzknoten werden alle Knoten der „_nodes" Liste eines TemplateNodes durchsucht. Falls ein passender Knoten gefunden wurde, wird nicht direkt der TemplateNode als Referenzknoten hinterlegt, sondern nur der SerializeableTemplateNode, da dieser keine Referenzen auf weitere Kindknoten beinhaltet.

Als Ergebnis eines Abgleichs wird zuerst eine Liste von SerializeableSystemNodes erzeugt. Jeder dieser Knoten, der einen Referenzknoten ermitteln konnte, hat diesen in Form eines SerializeableTemplateNodes abgelegt. Diese Liste lässt sich sehr leicht automatisiert serialisieren, da keine zirkulären Referenzen enthalten sind.

Um einen Vergleich der Berechtigungen zu initiieren, muss diese Liste wieder geladen werden. Dafür wird aus ihr erneut eine Baumstruktur erzeugt, die sich aus SystemTreeNodes zusammensetzt.

Als Ergebnis eines solchen Abgleiches erhält man einen Baum, der aus ResultNodes aufgebaut ist. Diese enthalten nur mehr die wichtigsten Informationen des Referenzknotens und die selektierten Unterschiede. Der Ergebnisbaum eines Abgleichs kann dadurch separat gespeichert werden und ist zu jedem Zeitpunkt wieder als Ergebnis abrufbar.

Für die graphische Visualisierung der Resultate wird ein TreeView verwendet, da die Ursprungsstrukturen damit auch visualisiert wurden. In Abbildung 27 wird zwischen den folgenden beiden Knotentypen unterschieden:

TemplateTreeNode wird bei der graphischen Ausgabe des Templates verwendet. Dadurch kann man das erstellte Template besser überprüfen, als in Form einer XML Datei.
Der VisualNode ist für die eigentliche Ausgabe der Ergebnisse verantwortlich. Beide Knoten sind vom TreeNode des .Net Frameworks abgeleitet. Diese Klasse ist relativ umfangreich und dadurch für eine Serialisierung nicht geeignet.

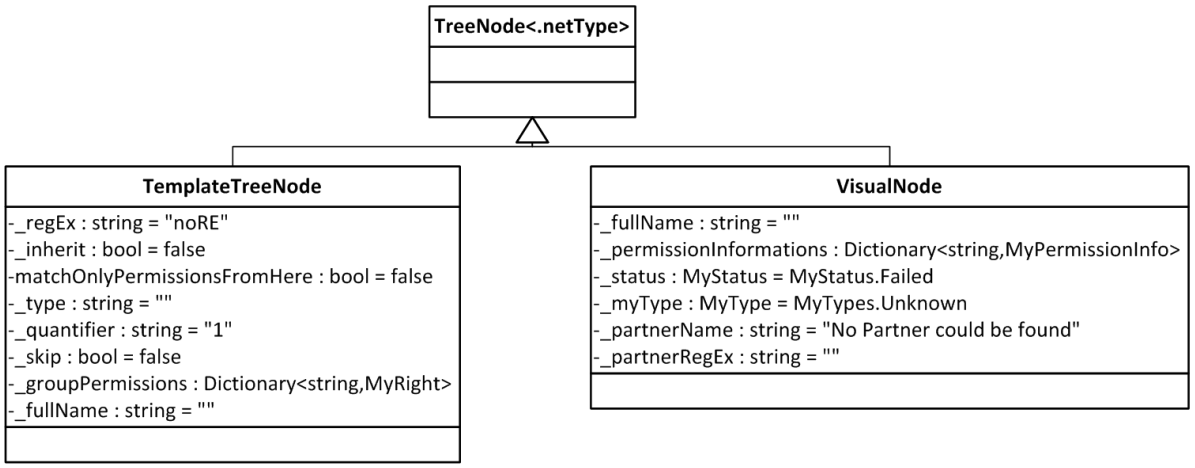


Abbildung 27 Klassenstruktur der Knoten für die Visualisierung

4.2.1 Methodik für mehrere mögliche Referenzknoten

Wenn sich mehr als ein Knoten auf der Liste „NodesToProcess" befindet, dann muss die in Abbildung 28 gezeigte „handleMoreThanOneReferenceNode()" Methode aufgerufen werden, da man mehr als einen möglichen Knoten im Template als Referenzknoten identifiziert hat.

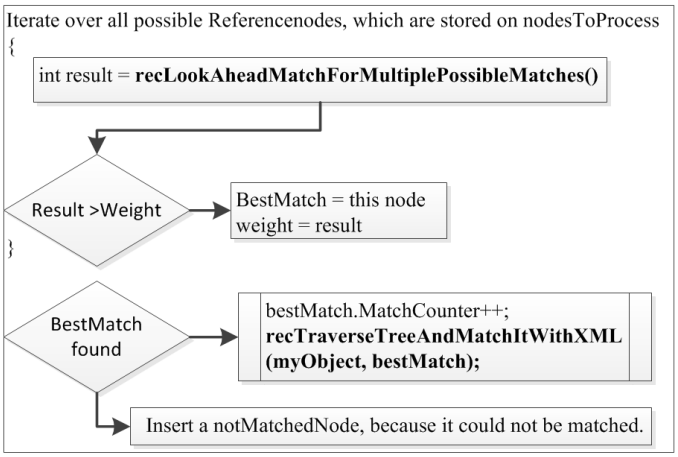


Abbildung 28 HandleMoreThanOneReferenceNode()

Hier wird dann die Methode „recLookAheadMatchForMultiplePossibleMatches()" für jeden dieser möglichen Referenzknoten aufgerufen. Diese Funktion liefert dann eine

Gewichtung laut Definition 6, die den Trefferkoeffizienten repräsentiert. Der größte gelieferte Wert gehört zu dem am besten geeignetsten Referenzknoten. Mit dem erhaltenen Knoten wird dann wieder die von oben bekannte Methode „recTraverseTreeAndMatchWithXML()" aufgerufen. Dieses Szenario tritt nur dann ein, wenn in der darunterliegenden Struktur keine verschiedenen Versionen von Knoten enthalten sind. Falls überhaupt kein passender Knoten ermittelt werden kann, muss ein Platzhalterknoten eingefügt werden, der diesen Umstand wiederspiegelt.

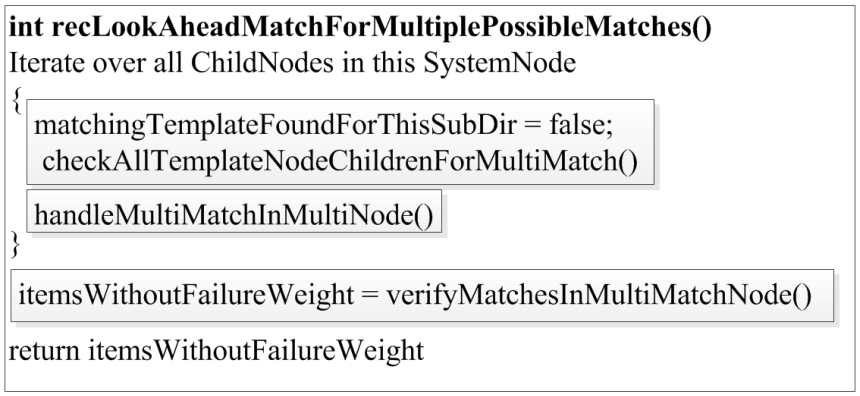


Abbildung 29 recLookAheadMatchForMultiplePossibleMatches()

In den darunterliegenden Strukturen sind aber auch verschiedene Versionen möglich, weswegen in der Methode „recLookAheadMatchForMultiplePossibleMatches()" (siehe Abbildung 29) auch eine Gewichtung benötigt wird. Eventuell ist ein passendes Abbruchkriterium ebenfalls von Vorteil.
Hier ist eine Traversierung des restlichen Baumes mittels Tiefensuche natürlich auch möglich. Jedoch hat dieser Ansatz einen Nachteil, da bei Abweichungen der letzten Kindknoten auf der ersten Ebene trotzdem alle tieferen Ebenen der vorderen Knoten kontrolliert werden müssen. Aus diesem Grund habe ich mich für einen Ansatz entschieden, der auf level-order (=Breitensuche) basiert und dadurch einen eventuell erwünschten Abbruch im Fehlerfall früher und auch auf einer höheren Ebene ermöglicht.
Abbildung 30 zeigt, wie die direkten Kindknoten der zu prüfenden Struktur jeweils mit den Kindknoten des korrespondierenden Musterknoten verglichen werden. Wenn zu einem Systemknoten ein möglicher Referenzknoten gefunden wurde, wird bei einer direkten Unifizierung ein Objekt, das System- und Templateknoten beinhaltet, auf die Liste „bfsList" hinzugefügt, das Flag „matchingTemplateFoundForThisSubDir" gesetzt, der Matchcounter und das Gewicht erhöht und im Grunde mit dem nächsten Systemknoten fortgefahren, da in diesem Fall keine Behandlung der indirekten Unifizierung erforderlich ist. Bei einer indirekten Unifizierung kann es passieren, dass in weiterer Folge mehrere passende Referenzknoten ermittelt werden, weswegen nicht direkt mit dem nächsten Systemknoten fortgefahren werden kann. Hier wird nur vermerkt, dass es eine Übereinstimmung gegeben hat. Das Gewicht wird dementsprechend erhöht, falls es noch

nicht erhöht wurde und der Knoten wird auf die Liste „multimatchKnotenList" hinzugefügt. Hier gilt es zu beachten, dass eine Fortsetzung mit dem nächsten Systemknoten nicht sofort möglich ist. Die Iteration wird normal fortgesetzt, da eventuell noch weitere potentielle Referenzknoten ermittelt werden. Falls kein passender Knoten ermittelt wurde, bleibt der Wert der Variable „matchingTemplateFoundForThisSubDir" auf false.

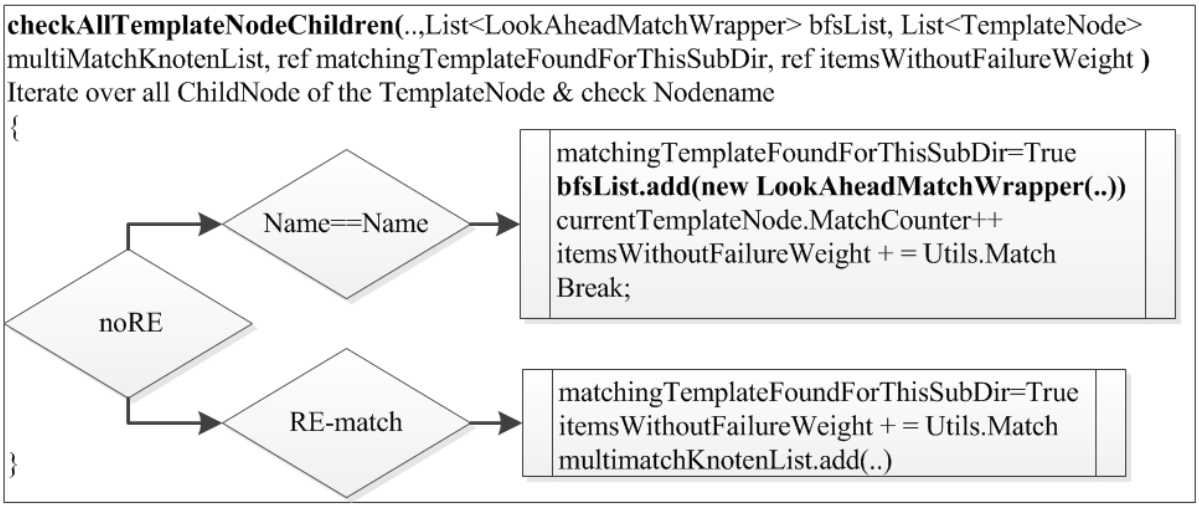


Abbildung 30 Abgleich mittels Breitensuche

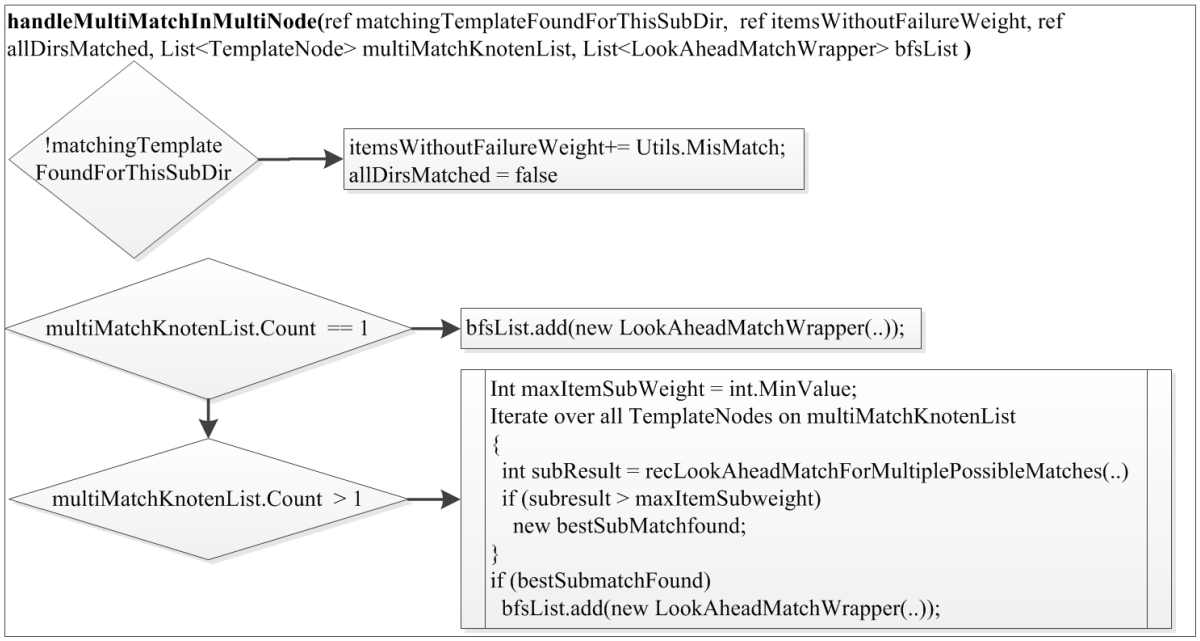


Abbildung 31 handleMultiMatchInMultiNode()

Im Anschluss daran wird in der Methode aus Abbildung 31 noch überprüft, ob dieser Wert gesetzt ist. Falls nicht, wird die Variable „allDirsMatched" auf false gesetzt und das Gewicht verringert. An dieser Stelle werden auch noch die indirekten Unifizierungen im

Grunde analog zur oberen Erklärung behandelt. Falls nur ein passender Referenzknoten gefunden werden kann, wird dieser, wie bei einer direkten Unifizierung, auf die Liste „bfsList" gesetzt. Wenn alle Knoten auf dieser Ebene geprüft wurden und bei indirekten Unifizierungen die Kontrollen der multiplen potentiellen Referenzknoten in den tieferen Ebenen abgeschlossen sind, müssen die Ergebnisse dieser Ebene noch verifiziert werden.

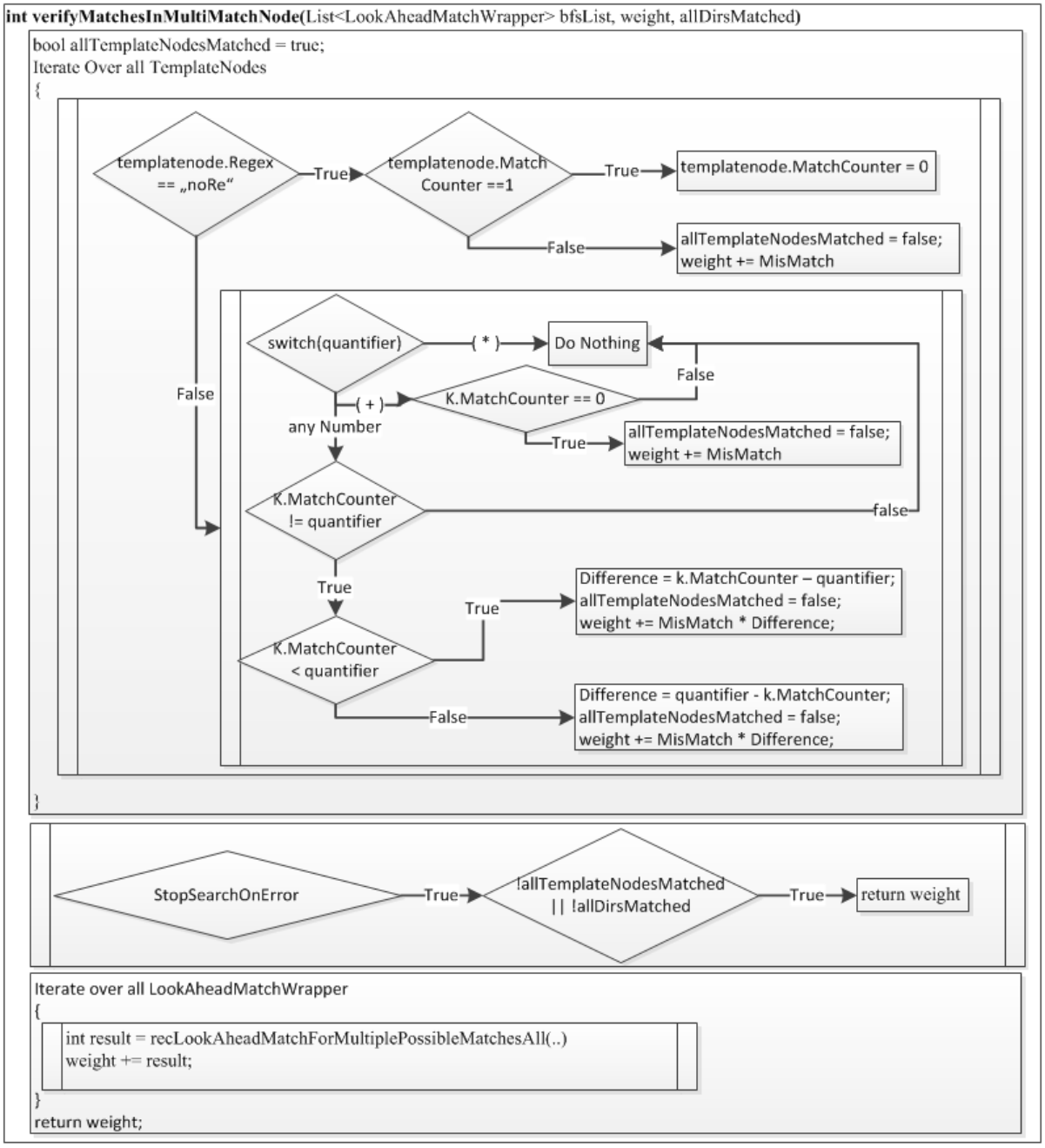


Abbildung 32 verifyMatchesInMultiMatchNode()

Dafür wird die Methode aus Abbildung 32 aufgerufen, die eine Gewichtung für die übergeordnete Funktion bereitstellt. Dabei wird zuerst wieder über alle Kindknoten des entsprechenden Knotens der Musterstruktur iteriert und die Gewichtung angepasst.

Danach werden bei aktivem „StopSearchOnError" Flag die beiden Variablen „allTemplateNodesMatched" bzw. „allDirsMatched" ausgewertet und dementsprechend abgebrochen. Das hat den Vorteil, dass man bei einer einzigen Abweichung sofort zu der Erkenntnis kommt, dass es sich nicht um den entsprechenden Referenzknoten handeln kann. Dieses Abbruchkriterium sollte aber nur dann verwendet werden, wenn man garantieren kann, dass es immer einen fehlerfreien Referenzknoten zu allen Systemknoten gibt. Andernfalls wird nicht immer der Referenzknoten mit der besten Gewichtung gewählt, da man auf einer zu hohen Ebene aufgrund eines Fehlers abbricht. Das hat zur Folge, dass vermeintlich Knoten aus tieferen Ebenen nicht mehr bewertet werden können.
Wenn keine Fehler gefunden wurden bzw. nicht vorher schon abgebrochen wurde, wird über die Knoten der Liste „bfsList" iteriert und entsprechend oft wieder die Funktion „recLookAheadMatchForMultiplePossibleMatches()" mit den gefundenen Systemknoten und dem jeweils korrespondierenden Referenzknoten aufgerufen. Das Ergebnis jedes Aufrufes ist die Gewichtung der darunterliegenden Struktur, die dann entsprechend aufsummiert von der „verifyMatchesInMultiMatchNode()" Methode zurückgeliefert wird.

4.3 Grafische UserInterfaces

Der in Abbildung 16 vorgestellte Prototyp hat sich als relativ zweckdienlich herausgestellt und kann alles bis auf die entsprechende Gewichtung wiedergeben. Diese Information ist dann von Vorteil, wenn es mehrere mögliche Referenzknoten gegeben hat. Auf den ersten Blick erscheint diese Information immens wertvoll, doch wenn man bedenkt, dass im Template meistens eine geringere Knotenanzahl als in der zu prüfenden Struktur definiert ist, relativiert sich die Wertigkeit dieser Kennzahl. Sobald viele indirekte Unifizierungen in einer Unterstruktur vorgenommen werden müssen, ist eine Schätzung dieses Gewichts relativ schwer. Wenn unter einem indirekt unifizierbaren Knoten aber nur direkt unifizierbare Knoten definiert sind, dann spiegelt diese Kennzahl das exakte Gewicht der darunterliegenden Struktur wieder und kann somit zu einer wertvollen Kennzahl der visuellen Verifizierung des Resultats werden.

Icon	Bedeutung
	In der Unterstruktur dieses Knotens sind keinerlei Fehler aufgetreten.
	In der Unterstruktur dieses Knotens ist ein struktureller Fehler.
	In der Unterstruktur dieses Knotens ist ein Berechtigungsfehler.
	Der Knoten konnte nicht erkannt werden.

Tabelle 5 Bedeutung der Icons in der GUI

In Tabelle 5 sind die verschiedenen Icontypen und ihre Bedeutungen abgebildet. Mit dieser einfachen Notation, die auch in Abbildung 33 und Abbildung 34 zum Einsatz

kommt, kann man auf einem Knoten höheren Levels sofort erkennen, ob darunterliegende Knoten fehlerhaft sind.

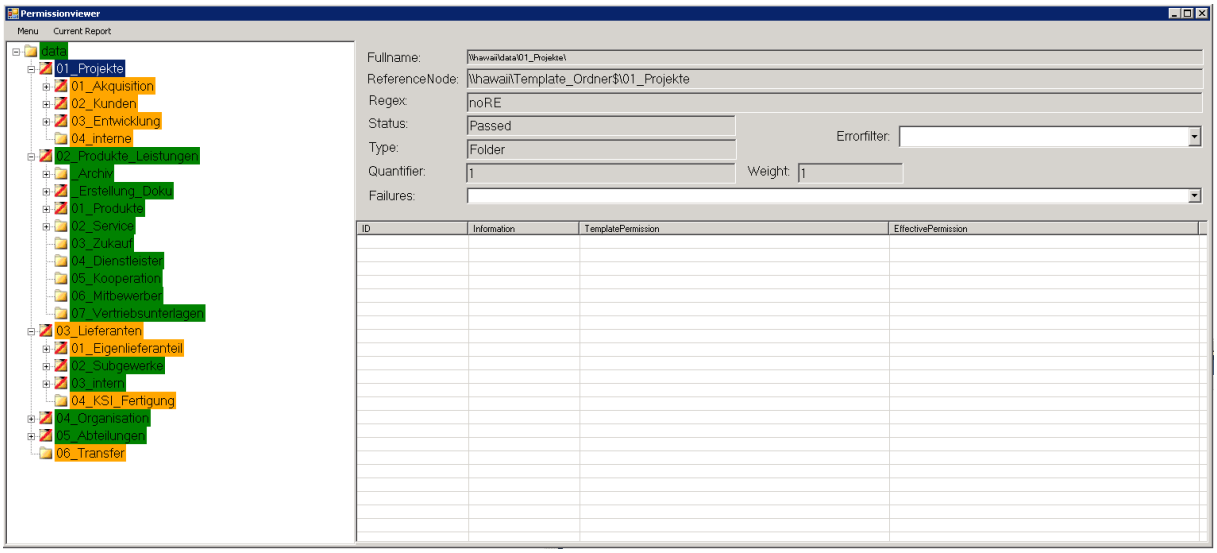


Abbildung 33 Visualisierung der Ergebnisse des Dateiservers

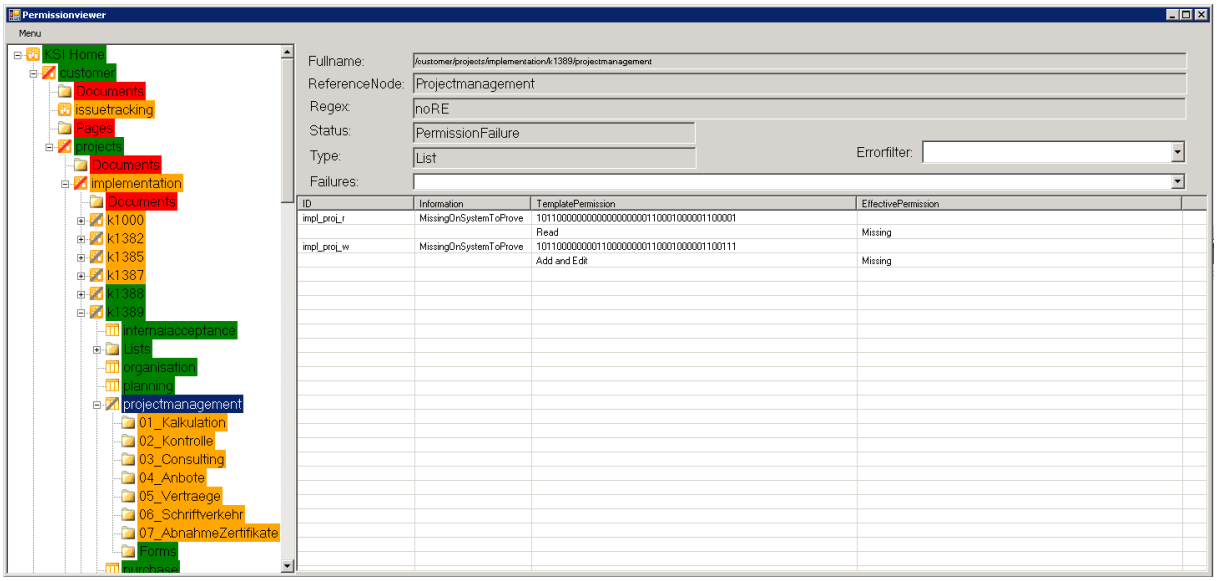


Abbildung 34 Visualisierung der Ergebnisse des SharePoints

Auf der rechten Seite ist ein Dropdownmenü namens „Errorfilter" zu sehen. Damit können die Ergebnisse der verschiedenen Fehler getrennt angesteuert werden. Problematisch ist nur das Hinzufügen der Einträge in das Dropdownmenü „Failures", da abhängig von den gefundenen Fehlern die Eintragsanzahl dieser Liste sehr groß werden kann. In den vorgestellten Szenarien betrug die Anzahl der Berechtigungsabweichungen teilweise über 40.000. Wenn diese Liste durch den Thread, der die graphische Oberfläche steuert, befüllt wird, dann friert die Oberfläche für diesen Zeitraum ein. In .NET ist für diesen Zweck die Klasse „BackgroundWorker" (=Thread, der im Hintergrund abgearbeitet

wird) vorgesehen [**16**]. Mit dieser kann dann aber leider nicht direkt auf die einzelnen Elemente der Benutzeroberfläche zugegriffen werden, da es sonst zu einem sogenannten „Cross-threading" kommt. Denn dafür ist ein Delegate zuständig, das am Methodenende ausgeführt wird und die ermittelten Daten an die Oberfläche weiterreicht. Das Binden von bestimmten Daten an ein Steuerelement kann aber auch nur in diesem Delegate durchgeführt werden und genau das ist das Problem, da diese Operation teilweise sehr viel Zeit in Anspruch nimmt.

Für diese Problematik wurde der in Abbildung 35 angedeutete Mechanismus entwickelt. Zuerst wird die gewünschte Liste ausgesucht und ein „Backgroundworker" gestartet, der diese Liste in kleinere Teile zerlegt. Diese Teillisten werden dann immer nur an einen einzigen weiteren Thread, der als BackgroundWorker arbeitet, delegiert. Da immer nur ein Thread für diesen Arbeitsprozess vorgesehen ist, können auch nicht mehrere Threads zum gleichen Zeitpunkt auf die graphische Oberfläche zugreifen.

Wenn diese Aufgabe nur von einem Thread durchgeführt wird, dann friert die Benutzeroberfläche nach kurzer Zeit ein, da keine Priorisierung des UI-Threads möglich ist. Ein weiteres Problem ist die angemessene Anzahl an hinzuzufügenden Einträgen, da es bei größeren Teillisten zum selben Problem kommt. Eine Zahl zwischen 100 und 200 hat sich in Tests als angemessen herausgestellt. Wenn diese Konstante zu groß gewählt wird, kommt es dementsprechend zu kurzen Aussetzern auf der Oberfläche. Des Weiteren ist dieser Wert natürlich von der Performance des Testsystems abhängig.

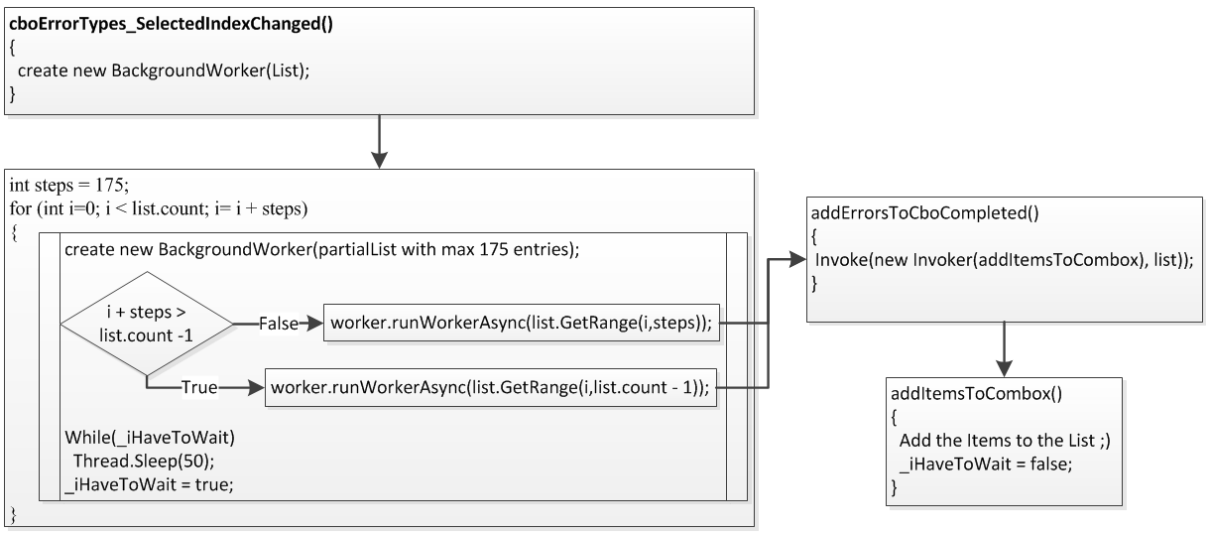


Abbildung 35 Partielle Listenbefüllung über BackgroundWorker

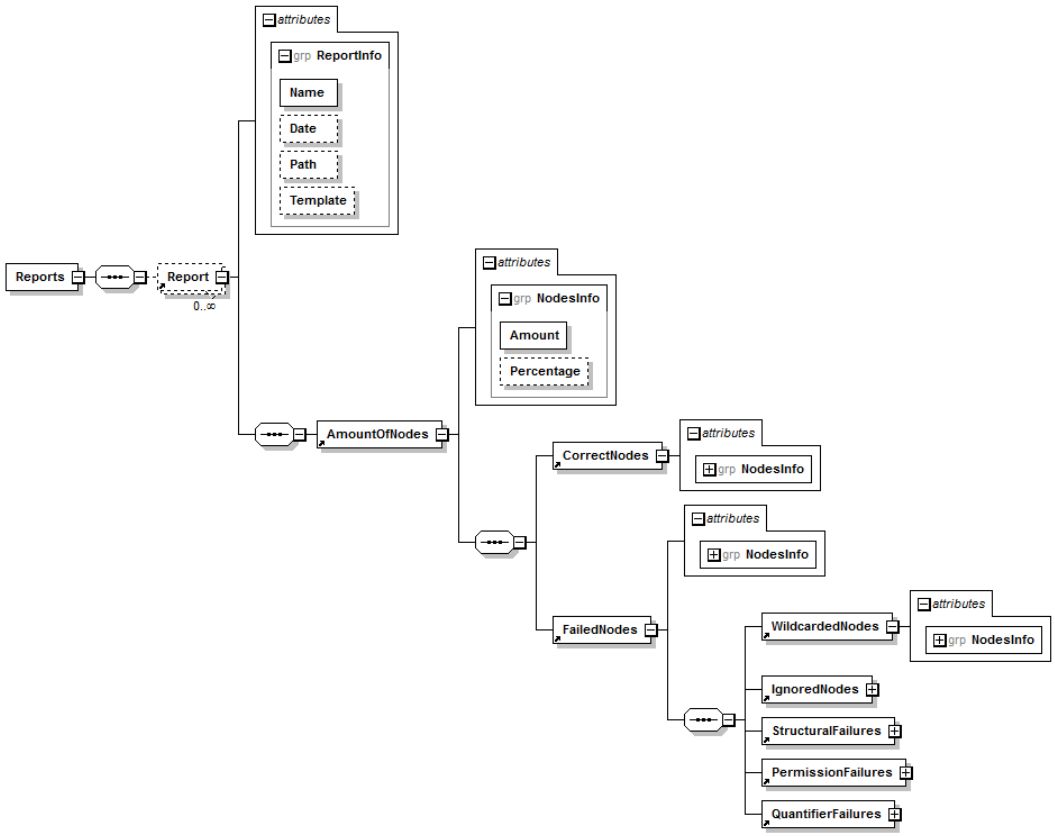


Abbildung 36 Schema der Reports

Die Reports aus Abbildung 17 sind wieder mittels XML realisiert. Dafür wird die Struktur selbst in einer XML Datei abgelegt (adäquates Schema: siehe Abbildung 36). Die Visualisierung wird dann mittels XSLT (=Extensible Stylesheet Language Transformations) in ein passendes Ausgabeformat transformiert.

5 Leistungsbewertung

5.1 Gewichtung nur auf Strukturebene

Da bei der Ermittlung der entsprechenden Referenzknoten in den meisten Fällen nur auf strukturelle und syntaktische Korrektheit geprüft wird, ist es natürlich nicht möglich eine hundertprozentige Genauigkeit beim Abgleich zu erreichen. Problematisch wird es dann, wenn man auf zwei potentielle Referenzknoten stößt, die sich strukturell und syntaktisch von einander in ihrer Unterstruktur nicht unterscheiden. Hier wird der Knoten als Referenzknoten gewählt, der zuerst gefunden wurde.

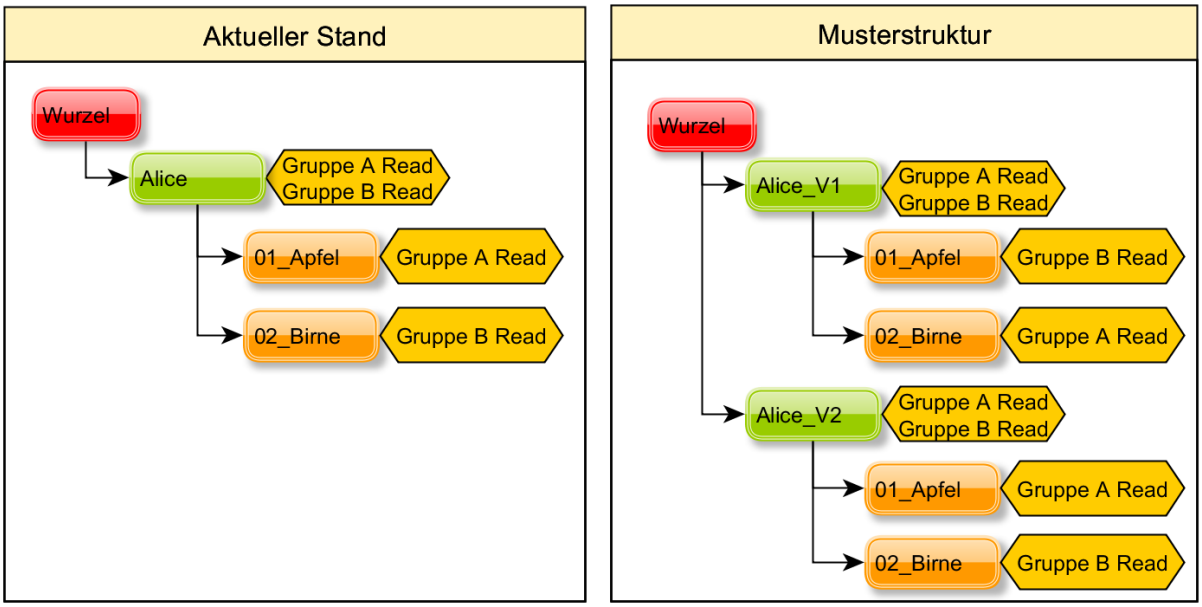


Abbildung 37 Falscher Referenzknoten

In Abbildung 37 wird der Systemknoten „Alice" betrachtet. In diesem Fall werden zwei potentielle Referenzknoten „Alice_V1" und „Alice_V2" ermittelt. Sie werden durch einen regulären Ausdruck indirekt unifiziert. Wie man sieht, sind die Unterstrukturen und die entsprechende Syntax dieser beiden Knoten ident. Dadurch erzielen beide den gleichen Gewichtungskoeffizienten. Jetzt kann es sein, dass sich diese beiden Knoten „Alice_V1" und „Alice_V2" noch in der Anzahl an berechtigten Gruppen auf diesem Knoten vom Systemknoten „Alice" unterscheiden. Wenn diese Zahl jedoch, wie in diesem Fall, ident ist und die Berechtigungen der entsprechenden Gruppen auch, müssen die einzelnen Berechtigungen der beiden Knoten noch mit dem Knoten „Alice" verglichen werden. Kommt es hier auch, wie in Abbildung 37, zu keiner Abweichung, dann wird Knoten „Alice_V1" als Referenzknoten gewählt, da er früher ermittelt wurde. In Wirklichkeit ist es aber so, dass die Knoten in der Unterstruktur von „Alice_V1" berechtigungstechnisch überhaupt nicht mit den Knoten der Unterstruktur von „Alice" übereinstimmen und es dadurch zu vielen Abweichungen bei den Berechtigungen kommt. Die Berechtigungen auf

der Struktur unter dem Knoten „Alice_V2" passen hingegen ideal zur Struktur unter dem Knoten „Alice". Dieses Problem lässt sich zu Lasten der Performance natürlich beheben. Das System ist in diesem Punkt auch erweiterbar, indem die Gewichtung einfach auch noch auf dem Abstraktionslevel der Berechtigungen durchgeführt wird. Diese Umstellung bringt natürlich eine enorme Genauigkeit, da man dadurch praktisch alle Strukturen richtig erkennen kann und keine Fehler mehr begeht. Die Bestimmung der einzelnen Berechtigungen ist allerdings ein sehr zeitaufwendiger Vorgang innerhalb dieses Prozesses. Deswegen ist es bei entsprechender Knotenanzahl von über 100.000 zeitlich nicht mehr zielführend diesen Lösungsweg zu gehen, auch wenn dadurch eine hundertprozentige Genauigkeit erzielt werden kann.

5.2 Genauigkeit vs. Geschwindigkeit

Durch die Variable „StopSearchOnError" ist es in indirekt unifizierten Knotenstrukturen möglich, vorzeitig festzulegen, dass es sich nicht um die gesuchte Referenzstruktur handeln kann. Das bringt auf der einen Seite zeitliche Vorteile, aber auf der anderen Seite wird dabei die strukturelle Erkennungsrate vermutlich schlechter. Dadurch kommt es vermutlich auch zu einer Verringerung der Berechtigungsabweichungen, da weniger Konten einen Referenzknoten ermitteln konnten. Die Knotenstruktur, die sich gar nicht bzw. erst in der tiefest möglichen Ebene von der zu prüfenden Struktur unterscheidet, wird als beste Struktur gewählt. Um diese Vermutungen zu belegen, wurden zwei Testläufe über den kompletten Dateiserver durchgeführt. Im ersten Szenario war „StopSearchOnError" aktiv und im zweiten Szenario wurde diese Option deaktiviert.

<table>
<tr><th>Szenario A
StopSearchOnError = Aktiv</th><th>Szenario B
StopSearchOnError = Inaktiv</th></tr>
<tr><td>
KSI-Data on 04.12.2011 at 20:45
Path: \\ksi\data
Template: C:\Users\sed___DA\xml\fsTemplate.xml
<table>
<tr><th>Type</th><th>Amount</th><th>Percentage</th></tr>
<tr><td>Amount of nodes</td><td>104457</td><td></td></tr>
<tr><td>Correct nodes</td><td>70551</td><td>67,54%</td></tr>
<tr><td>Failed nodes</td><td>33906</td><td>32,46%</td></tr>
</table>
<table>
<tr><th>Failed nodes</th><th>Amount</th><th>Percentage</th></tr>
<tr><td>Wildcarded nodes</td><td>0</td><td>0,00%</td></tr>
<tr><td>Ignored nodes</td><td>2</td><td>0,00%</td></tr>
<tr><td>Structural failures</td><td>4517</td><td>4,32%</td></tr>
<tr><td>Permission failures</td><td>29273</td><td>28,02%</td></tr>
<tr><td>Quantifier failures</td><td>114</td><td>0,11%</td></tr>
</table>
</td><td>
KSI-Data on 04.12.2011 at 21:44
Path: \\ksi\data
Template: C:\Users\sed___DA\xml\fsTemplate.xml
<table>
<tr><th>Type</th><th>Amount</th><th>Percentage</th></tr>
<tr><td>Amount of nodes</td><td>104498</td><td></td></tr>
<tr><td>Correct nodes</td><td>70438</td><td>67,41%</td></tr>
<tr><td>Failed nodes</td><td>34060</td><td>32,59%</td></tr>
</table>
<table>
<tr><th>Failed nodes</th><th>Amount</th><th>Percentage</th></tr>
<tr><td>Wildcarded nodes</td><td>0</td><td>0,00%</td></tr>
<tr><td>Ignored nodes</td><td>2</td><td>0,00%</td></tr>
<tr><td>Structural failures</td><td>4518</td><td>4,32%</td></tr>
<tr><td>Permission failures</td><td>29426</td><td>28,16%</td></tr>
<tr><td>Quantifier failures</td><td>114</td><td>0,11%</td></tr>
</table>
</td></tr>
<tr><td>Laufzeit: 54 Minuten</td><td>Laufzeit: 56 Minuten</td></tr>
</table>

Tabelle 6 Detektions- und Laufzeitergebnisse

In Tabelle 6 sieht man, dass sich die Vermutungen nur teilweise belegen lassen. Das „Szenario A", mit vorzeitigem Abbruch, wurde bei einer überprüften Knotenanzahl von

104.457 um nur zwei Minuten schneller abgearbeitet, als „Szenario B" mit 104.498 Knoten. Das ist leider nur eine marginale Performanceverbesserung, aber trotzdem eine Verbesserung. Folgende Vermutungen gilt es für eine aktivierte „StopSearchOnError" Variable noch zu belegen:

1.) Eine geringere Gesamtknotenanzahl wird überprüft.
2.) Geringere Abweichungen bei den Berechtigungen.
3.) Dadurch auch eine geringere Anzahl an fehlerhaften Knoten.

	Szenario A	Szenario B	Differenz (Szenario A – Szenario B)
Amount of nodes	104.457	104.498	-41
Correct nodes	70.551	70.438	113
Failed nodes	33.906	34.060	-154
Structural failures	4.517	4.518	-1
Permission failures	29.273	29.426	-153

Tabelle 7 Gegenüberstellung der beiden Szenarien

Laut Tabelle 7 trifft die erste Vermutung in diesem Testlauf zu, da in der Tat 41 Knoten weniger überprüft wurden. Die Verringerung der Berechtigungsdifferenzen ist auf die kleinere überprüfte Knotenmenge zurückzuführen. Im Schnitt sind ungefähr 3,55 Berechtigungen je Knoten fehlerhaft. Das ergibt für 41 Knoten eine zusätzliche Berechtigungsabweichung von 145,55. Die 153 „permission failures" liegen sehr nahe an diesem Wert und machen das Ergebnis dadurch auch sehr plausibel. Somit ist auch die zweite Vermutung praktisch belegt. Die überprüfte Knotenmenge ist um 41 gestiegen, dadurch steigt auch die Anzahl an fehlerhaften Berechtigungen durchschnittlich um den Faktor 3,55 pro erkanntem Knoten. Die unter einem Knoten liegende Struktur, die dadurch nicht mehr erkannt wird, wird nicht als zusätzlicher Fehler gewertet. Durch diese Tatsache lässt sich auch die dritte Vermutung belegen, da die Fehlerrate pro nicht erkanntem Knoten nur um den Faktor 1 steigen kann. Die Ergebnisse dieses Testlaufs sind nicht repräsentativ und sehr vom Template und der zu prüfenden Struktur abhängig, da bei geringen Ähnlichkeiten der versionsabhängigen Unterstrukturen früher abgebrochen werden kann, als bei nahezu identen.

5.3 Verschiedene Versionen eines Referenzknotens

Wenn innerhalb einer indirekt unifizierten Struktur, von der es mehr als eine Version gibt, weitere indirekte Unifizierungen vorgenommen werden müssen, dann wird dieser Vorgang mehrmals zu verschiedenen Zeitpunkten wiederholt. Das folgende Beispiel in Abbildung 38 verdeutlicht diese Problematik:

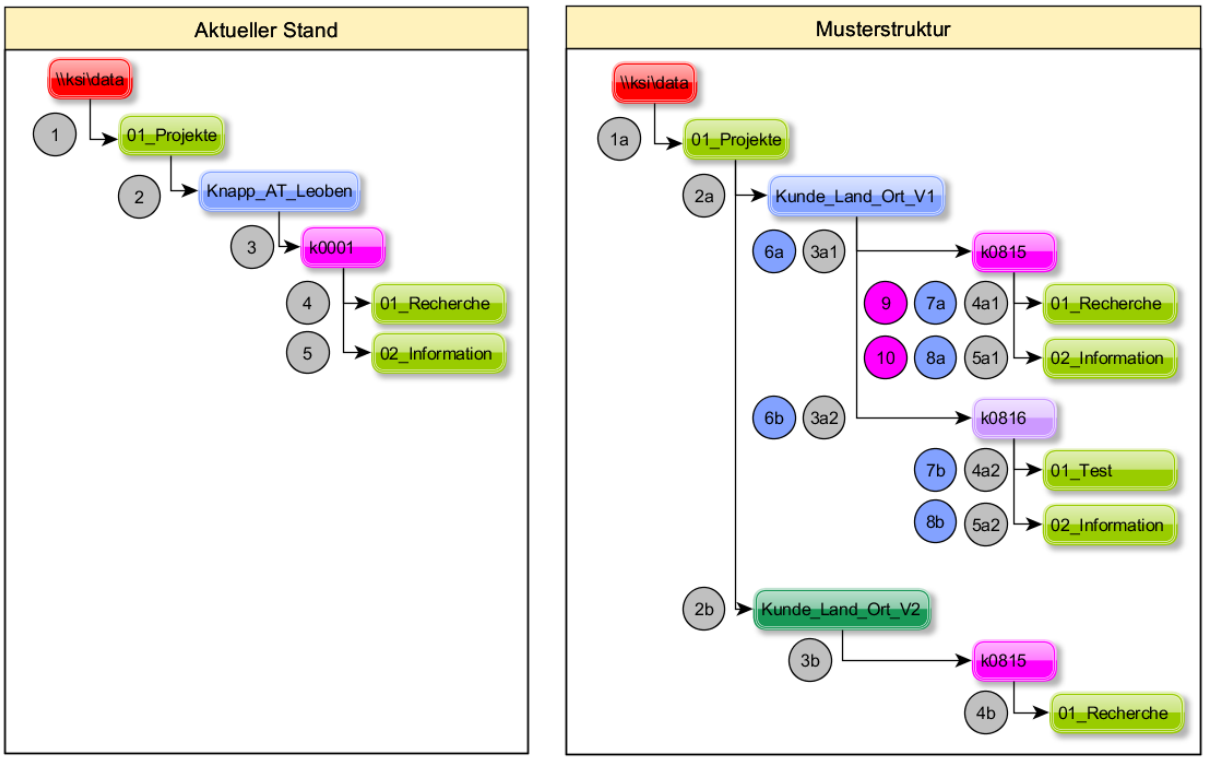


Abbildung 38 Erneutes Unifizieren verschiedener Knoten

1.) Knoten „01_Projekte" wird direkt unifiziert.

2.) Es wird ermittelt, ob es sich um „V1" bzw. „V2" handelt. Dafür werden die Knoten der Struktur unterhalb „V1" in folgender Reihenfolge traversiert: 3a1, 4a1, 5a1, 3a2, 4a2 und 5a2. Als Ergebnis erhält man die Gewichtung dieser Struktur. Danach muss die Struktur unterhalb „V2" in folgender Reihenfolge traversiert werden: 3b und 4b. Die Gewichtung dieser Struktur ist geringer als die der Struktur unter dem Knoten „V1". Deswegen wird der Knoten „Knapp_AT_Leoben" mit dem Knoten „Kunde_Land_Ort_V1" unifiziert.

3.) Man steht wieder vor dem gleichen Problem wie bei Punkt 2. Für den Knoten „k0001" gibt es zwei verschiedene Versionen. Die darunterliegende Struktur wird dann folgendermaßen traversiert: 6a, 7a und 8a. Die zweite mögliche Struktur wird in der Reihenfolge: 6b, 7b und 8b traversiert. Beide Versionen erhalten ein Gewicht und „k0815" wird als besserer Referenzknoten auf Grund des höheren Gewichtungskoeffizienten unifiziert.

4.) Die Knoten „01_Recherche" und „02_Information" werden mit ihren Referenzknoten neun und zehn direkt unifiziert.

Diese Lösung ist offensichtlich nicht optimal und kann unter anderem durch die in Kapitel 6.2.1 „Ergebnisliste als Baumstruktur" vorgestellte Variante viel effizienter realisiert werden. Die Verifizierung der „Quantifier" Attribute muss dann dementsprechend zu einem früheren Zeitpunkt und dadurch eventuell auch öfters als

jetzt durchgeführt werden. Dadurch ist auch die in Kapitel 5.1 vorgestellte Problematik besser zu kontrollieren, da die gleichen Knotenstrukturen nicht unnötig oft traversiert werden müssen.

6 Fragen, Schlussfolgerungen und Ausblicke

6.1 Fragen und Schlussfolgerungen

Bei der in Kapitel 4.2.1 „Methodik für mehrere mögliche Referenzknoten" vorgestellten Erkennung von mehreren potentieller Referenzknoten ist folgende Problematik zu bedenken, die in Abbildung 39 skizziert wird:

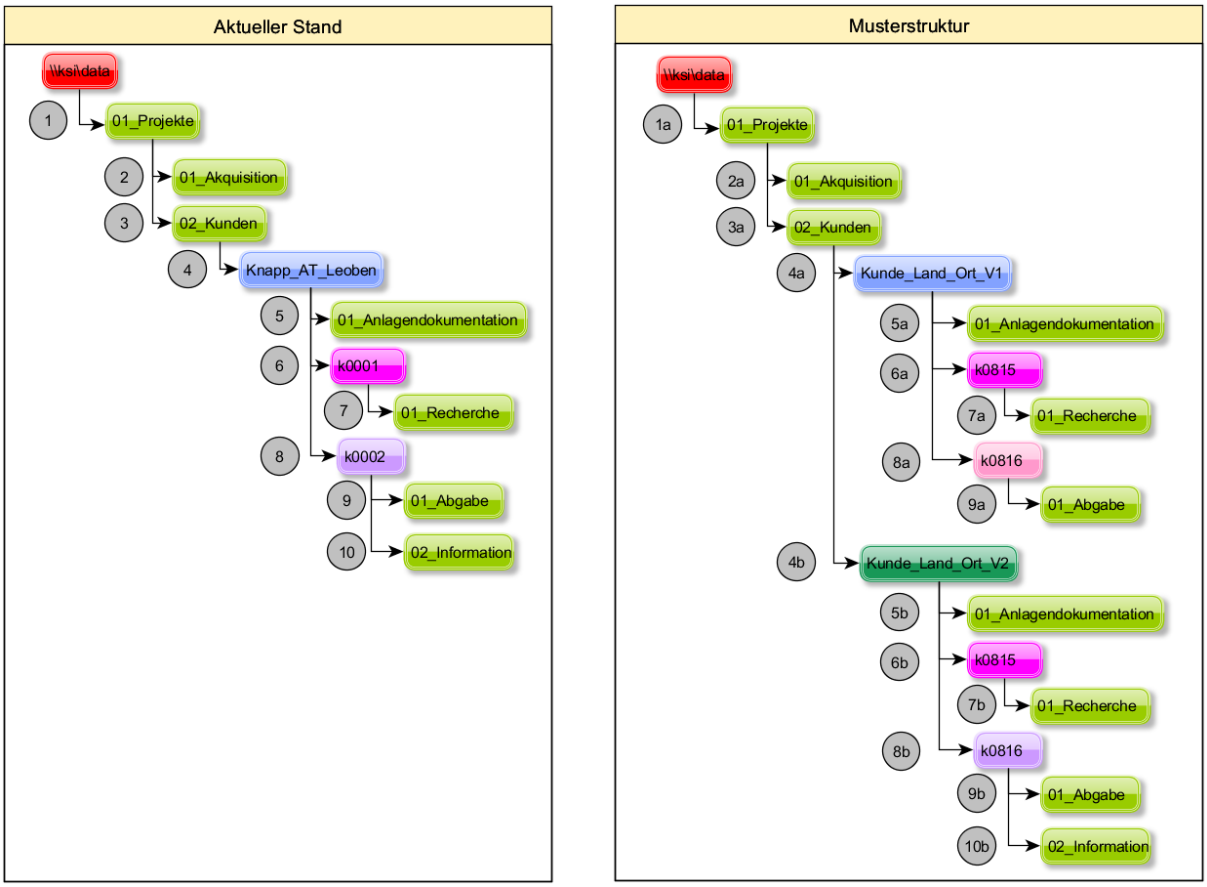


Abbildung 39 Ineffizienz bei mehreren möglichen Referenzknoten

Bei Schritt Nummer 4 handelt es sich um eine indirekte Unifizierung. Nach Traversierung der beiden möglichen Strukturen der Musterstruktur (Knoten 4a bis 9a bzw. 4b bis 10b) ist klar, dass „Kunde_Land_Ort_V2" der bessere Referenzknoten ist. Da die Verifizierung aber nur nach einem normalen Rekursionsschritt erfolgt, der keine multiplen Referenzknoten unterstützt, wird die Traversierung normal mit dem gefundenen Referenzknoten fortgesetzt. Das bedeutet aber auch, dass im weiteren Verlauf eine erneute indirekte Unifizierung bei den Knoten 6 und 7 vorgenommen wird. Dieser Problematik ist ohne des in Kapitel 0 vorgestellten Ansatzes schwer entgegenzuwirken, da die Verifizierung auf einer einzigen Ergebnisliste basiert. Diese enthält nur Knoten, die zu erfolgreich abgeglichenen Referenzknoten gehören. Deswegen dürfen die Knoten unter einem potentiellen Referenzknoten (Bei Knoten 4a „Kunde_Land_Ort_V1" sind das die Knoten 5a bis 9a) nicht zu dieser Liste hinzugefügt und somit in den

Verifizierungsprozess integriert werden. Bei den beiden hier vorgestellten Szenarien ist das allerdings kein Problem, da die zeitlichen Rahmenbedingungen trotz dieser Problematik eingehalten werden können. Bei einer exzessiven Verwendung von verschiedenen verschachtelten Versionsknoten kann es allerdings zu erheblichen zeitlichen Problemen kommen. Das in Abbildung 40 abgebildete Szenario wirkt auf den ersten Blick sehr konstruiert, ist aber theoretisch möglich. Hier muss zuerst festgestellt werden, um welchen Knoten es sich handelt: „Kunde_Land_Ort_V1" bzw. „Kunde_Land_Ort_V2". Wenn der richtige Referenzknoten ermittelt wurde, muss danach wieder die darunterliegende Struktur traversiert werden. Dieser Vorgang wurde aber schon beim oberen Referenzknoten durchgeführt. Dieses Szenario setzt sich bis in die unterste Ebene fort.

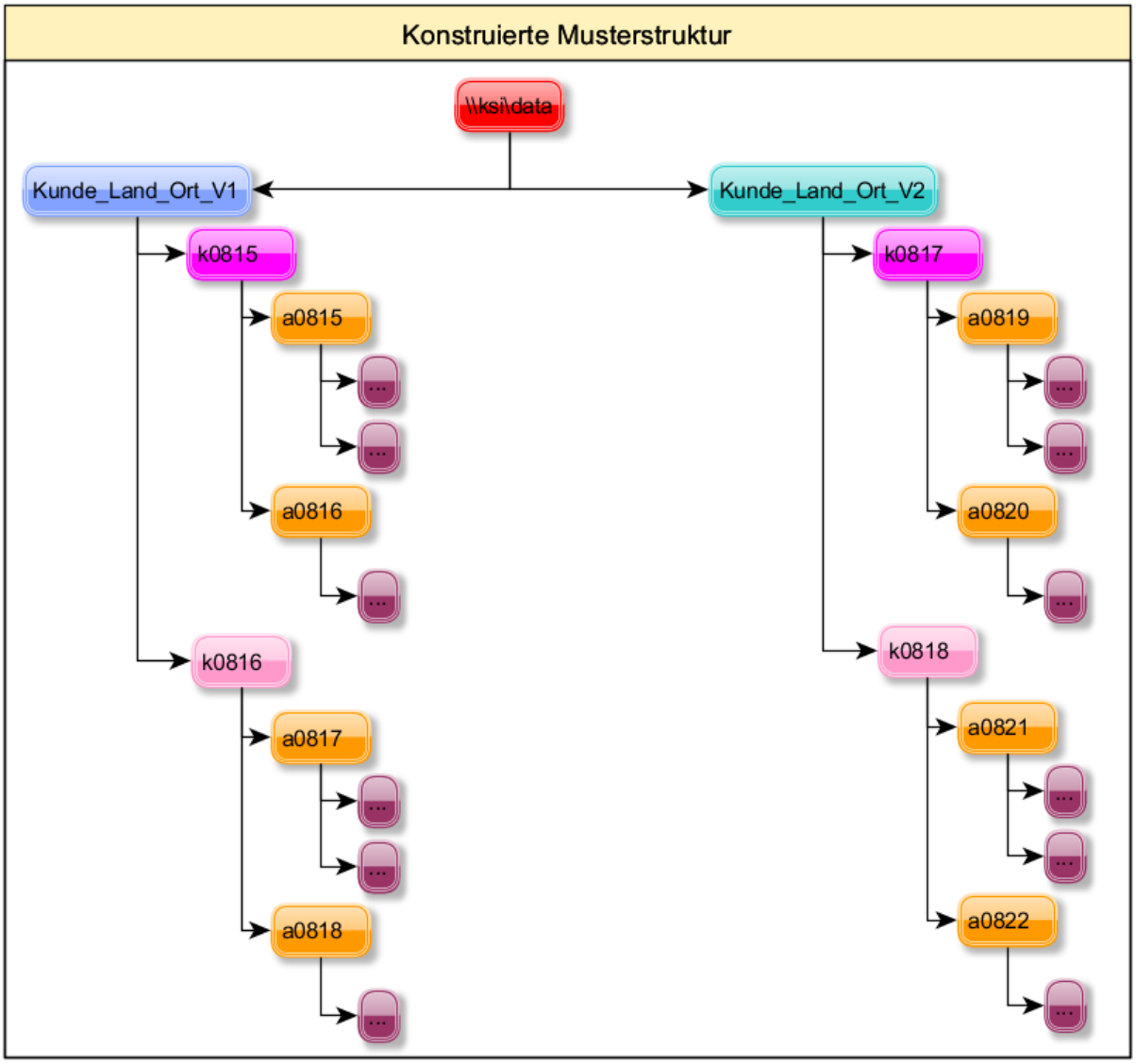


Abbildung 40 Konstruierte Musterstruktur

Abgesehen vom Abgleich von Berechtigungen kann die Strukturerkennung auch noch für andere Zwecke eingesetzt werden. Für das folgende Szenario am Dateiserver kann die automatische Strukturerkennung beispielsweise auch verwendet werden:

In jedem Projektordner befindet sich ein Ordner mit dem Namen „z_erledigt". Falls dieser Ordner weitere Kindknoten hat, dann ist dieses Projekt abgeschlossen und kann archiviert werden. In diesem Fall kann im Template unter dem Knoten „z_erledigt" ein weiterer Kindknoten eingefügt werden, der mit jedem beliebigen Knotennamen unifizierbar ist. Der „Quantifier" dieses Knotens wird auf den Wert 0 festgelegt. Dadurch werden alle Projekte auf die Liste der falsch quantifizierten Knoten hinzugefügt. Dadurch sind sie leicht zu identifizieren und können in zukünftigen Entwicklungen eventuell auch automatisiert archiviert werden.

6.2 Ausblick

6.2.1 Ergebnisliste als Baumstruktur

Die in Kapitel 4.2 „Ermittlung der passenden Referenzknoten und Verifizierung" vorgestellte Variante zur Speicherung der Systemknoten mit ihren Referenzknoten ist nicht sehr effizient bei der Verifizierung der Resultate. Für die Verifizierung muss teilweise die komplette Ergebnisliste durchsucht werden, was entweder zeit- oder speicherintensiv (=Dictionary; Map) ist. In diesem Fall ist es sehr speicherintensiv, da bei zu hohem Matchcounter ein Dictionary nach passenden Knoten durchsucht werden muss. Hier ist es sinnvoller die Ergebnisse direkt in einer Baumstruktur zu speichern. Das bringt große Vorteile bei der Verifizierung der Resultate. Ein Baum ist jedoch bei der automatischen Serialisierung auf Grund seiner Referenzen problematisch, weswegen zuerst die Speicherung der Ergebnisse in einer Baumstruktur erforderlich ist. Diese wird dann in eine Liste von Objekten, die keine Kind oder Elternknoten beinhalten, umgeformt. Diese Liste lässt sich dann automatisiert serialisieren. Beim Deserialisieren wird aus dem „Fullname" Attribut der einzelnen Objekte dieser Liste wieder wie vorher der Baum rekonstruiert.

6.2.2 Tolerantes Matching

Bei den direkten Unifizierungen ist es aktuell so, dass Knotennamen exakt übereinstimmen müssen, um einem Referenzknoten zugeordnet zu werden. Hier ist in anderen Szenarien eventuell eine tolerantere Unifizierung erwünscht, die vermeintliche Tippfehler toleriert. Hierfür kann beispielsweise ein weiteres Attribut für die Knoten im Template eingeführt werden. Dadurch ist ein tolerantes Matching für einzelne Knoten realisierbar. Hier könnte man in weiterer Folge mit Levenshtein Distanz, Hamming Distanz (bei gleich langen Strings), Jaro-Winkler Distanz oder ähnlichen Algorithmen

arbeiten. Das hat den Vorteil, dass bei kleineren Fehlern der Strukturvergleich in tieferen Ebenen trotz geringer Abweichungen fortgesetzt werden kann.

6.2.3 Integration diverser Workflows

Der in Kapitel 3.6 „Ausnahmenbehandlung" beschriebene Aspekt der verschiedenen Ausnahmen kann in verschiedene Workflows integriert werden.
Bei Berechtigungsabweichungen zwischen der Musterstruktur und dem zu prüfenden System kann man die Abteilungszugehörigkeit eines Benutzers, der vom Template abweicht, eruieren. Dadurch sind Workflows aktivierbar, die bei den jeweiligen Abteilungen nachfragen, ob diese Berechtigung erforderlich ist bzw. sie von entsprechenden Berechtigungsänderungen in Kenntnis setzen, da diese nicht den internen Richtlinien entsprechen.

6.2.4 Verbesserte Ausnahmenbehandlung

Bei den Ausnahmen weiß man nicht, ob sie überhaupt benötigt werden. Es kann nicht ermittelt werden, ob die Ausnahme überflüssig ist. Da die Ausnahmenbehandlung vor dem Abgleich ist, wird somit jeglicher Berechtigungs- bzw. Strukturvergleich verhindert. Hier muss die Reihenfolge vertauscht und eventuell ein neuer Fehlertyp definiert werden, der die unnötigen Ausnahmen beinhaltet.
Wobei in diesem Fall sogar beide Varianten ihre Daseinsberechtigung haben, da der gewählte Ansatz auf Grund der kleineren Vergleichsanzahl eine bessere Performance liefert.

7 Zusammenfassung

Jeder Mitarbeiter einer Firma hat in seinem Unternehmen verschiedene Berechtigungen. Diese sind in diversen Dokumentationen bzw. definierten Standards festgelegt. Die Kontrolle dieser Berechtigungen fällt in den meisten Fällen schwer, da die Wartung und Verifizierung diverser Berechtigungen nicht trivial lösbar sind. Für dieses Szenario wurde eine automatische Strukturerkennung entwickelt, die in weiterer Folge auch die Berechtigungen in diesen Systemen mit einer vordefinierten Vorlage abgleicht. Am Beispiel eines Microsoft Dateiservers und des Microsoft SharePoints wird eine automatisierte Strukturerkennung durchgeführt. Wenn dieser Vorgang abgeschlossen ist, können die jeweiligen Berechtigungen auf den ermittelten Knoten mit einem Template abgeglichen werden. Der Unifizierungsprozess zwischen einem System- und einem Referenzknoten aus dem Template erfolgt ausschließlich auf syntaktischer Ebene. Bei dieser Unifizierung wird zwischen direkten und indirekten Unifizierungen unterschieden. Bei indirekten Unifizierungen muss der Knotenname einem bestimmten Muster entsprechen, das in Form eines regulären Ausdrucks in den Musterknoten hinterlegt ist. In diesem Template können auch weitere Attribute auf den einzelnen Knoten hinterlegt werden. Dadurch können verschiedenste Kriterien, wie beispielsweise Auftrittshäufigkeit, sehr leicht umgesetzt werden. Die Definition des Templates und einer Ausnahmenstruktur wird mittels XML realisiert und durch ein Schema verifiziert. Dementsprechend werden verschiedene Gegenüberstellungen einzelner Ansätze diskutiert. Die hier vorgestellte Methodik einer automatischen Strukturerkennung mit Berechtigungsverifizierung lässt sich auf verschiedenste System übertragen, die durch eine Baumstruktur beschrieben werden.

8 Literaturverzeichnis

[1] www.NTFS.com. (2011) NTFS.com. [Online]. http://www.ntfs.com/ntfs-permissions.htm

[2] Prof. Dr. phil. Karl Heinz Wagner. (2011, Juni) Mathematische und logische Grundlagen der Linguistik. [Online]. http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel7.aspx

[3] Kanghong et al., "XML Document Correction Based on Update Conflict," in *XML Document Correction Based on Update Conflict*. Seoul: IEEE, 2010, pp. 273-278.

[4] Chen Chu and Guoqiang Li, "An Efficient Algorithm for Automatic Equational," in *An Efficient Algorithm for Automatic Equational*. Shanghai: IEEE, 2010, pp. 1088-1092.

[5] Prof. Dr. Bernhard Beckert. (2006) http://www.uni-koblenz.de | 10PraedikatenlogikSubstitutionenUnifikation_Teil2. [Online]. http://userpages.uni-koblenz.de/~beckert/Lehre/Logik/10PraedikatenlogikSubstitutionenUnifikation_Teil2.pdf

[6] Helmut Vonhoegen, *Einstieg in XML*.: Galileo Computing, 2011, pp. 21-181; 227-332.

[7] Radsoftware. (2011) Radsoftware. [Online]. http://www.radsoftware.com.au/articles/regexlearnsyntax.aspx

[8] Marcus Wiederstein, Sarah Winterstone Marco Skulschus, *XML Schema*.: Comelio Medien, 2011, pp. 25-202; 243-294; 345-384.

[9] w3schools.com. (2011) Introduction to XML Schema. [Online]. http://www.w3schools.com/schema/schema_intro.asp

[10] Christian Jänicke. (2010, Januar) Bitworld. [Online]. http://www.bitworld.de/index.php/grundlagen/dtd

[11] David Beech, Murray Maloney, Noah Medelsohn Henry S. Thompson. (2004, October) W3C. [Online]. http://www.w3.org/TR/xmlschema-1/#declare-attribute

[12] Microsoft. (2011) MSDN Library. [Online]. http://msdn.microsoft.com/en-us/library

[13] thund3rstruck. (2008, May) Howto: (Almost) Everything In Active Directory via C#. [Online]. http://www.codeproject.com/KB/system/everythingInAD.aspx

[14] Erik Wilde. www.dret.net. [Online]. http://dret.net/netdret/docs/wilde-xmlschema-1.pdf

[15] Ayub Khan and Marina Sum. (2006, November) sun.com. [Online]. http://developers.sun.com/jsenterprise/archive/nb_enterprise_pack/reference/techart/design_patterns.html

[16] Microsoft. MSDN BackgroundWorker. [Online]. http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker%28v=VS.90%29.aspx