

# **Synchronisieren von Capella Modellen mithilfe von Tripel-Graph-Grammatiken**

Bachelorarbeit von Adrian Zwenger  
Tag der Einreichung: 27. Juni 2022

1. Gutachten: Prof. Dr. rer. nat. Andy Schürr  
2. Gutachten: M. Sc. Lars Fritzsche

Darmstadt  
ES-B-0158



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Fachbereich Elektrotechnik  
und Informationstechnik  
Institut für Datentechnik  
Fachgebiet Echtzeitsysteme

---

## **Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt**

---

Hiermit versichere ich, Adrian Zwenger, die vorliegende Bachelorarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 27. Juni 2022

---

A. Zwenger

# Zusammenfassung

---

Im Model-Based Systems Engineering (MBSE) werden viele verschiedene Sichten auf ein System modelliert, um die Systemarchitektur zu entwerfen und zu dokumentieren. An diesem Planungs- und Dokumentationsprozess sind in der Regel eine Vielzahl an Personen beteiligt, welche aus verschiedenen Domänen stammen und unterschiedliche Interessen verfolgen. Dabei kann es vorkommen, dass sich Systemsichten im Informationsgehalt überlappen und diese simultan modifiziert werden. Durch das Überlappen im Informationsgehalt und dem simultanen Modifizieren der Sichten, können Konflikte und Unklarheiten in der Beschreibung des Systems, bestehend aus der Summe aller Sichten, entstehen. Um dies zu vermeiden, müssen die verschiedenen Sichten synchron zueinander gehalten werden. Das Capella MBSE-Werkzeug implementiert die ARCADIA-MBSE-Methode, und eignet sich besonders zum Planen und Modellieren von Systemarchitekturen. Allerdings existieren derzeit keine Möglichkeiten, Capella-Modelle werkzeuggestützt miteinander zu synchronisieren. Capella-Modelle müssen daher manuell abgeglichen und konsolidiert werden. Proprietäre Angebote, wie Team for Capella, ermöglichen das simultane kollaborative Bearbeiten von Capella-Modellen, aber dies wird durch das Teilen einer gemeinsamen Projekt-Instanz auf einem Server und dem sperren von Modellementen ermöglicht. In dieser Arbeit soll ermittelt werden, ob Tripel-Graph-Grammatiken (TGGen) sich eignen, um Capella-Modelle zu synchronisieren. Hierfür wird ein beispielhafter Regelsatz zum Synchronisieren zweier Capella-Systemmodelltypen definiert und mit eMoflon::IBeX ein Übersetzungs- und ein Synchronisationsprototyp implementiert. Zuletzt wird mittels einer Laufzeitanalyse die Skalierbarkeit des implementierten Übersetzers und Synchronisierers beurteilt.

# Inhaltsverzeichnis

---

<b>1. Einleitung</b>	<b>1</b>
<b>2. Grundlagen</b>	<b>3</b>
2.1. Laufendes Beispiel . . . . .	3
2.2. Systems Engineering . . . . .	4
2.3. Model-Based Systems Engineering (MBSE) . . . . .	8
2.4. Metamodelle . . . . .	10
2.5. Graphtransformation . . . . .	12
2.6. Modellsynchronisation mittels Tripel-Graph-Grammatiken . . . . .	15
2.6.1. Erweiterung des laufenden Beispiels um ein Korrespondenzmodell . . . . .	17
2.6.2. Tripel-Graph-Grammatik . . . . .	18
2.6.3. Inkrementelle Modellsynchronisierung . . . . .	24
2.7. Die ARCADIA MBSE-Methode . . . . .	26
2.7.1. Operational Analysis . . . . .	29
2.7.2. System Need Analysis . . . . .	31
2.7.3. Logical Architecture . . . . .	32
2.8. Das Capella MBSE-Werkzeug . . . . .	33
2.8.1. Die Capella Nutzeroberfläche . . . . .	34
2.8.2. Team for Capella . . . . .	38
<b>3. Implementierung</b>	<b>44</b>
3.1. Capella Metamodelle . . . . .	44
3.2. Erzeugen von Modellelementen . . . . .	51
3.2.1. Komponenten mit einer Realisierung . . . . .	53
3.2.2. Komponenten mit mehreren Realisierungen . . . . .	56
3.2.3. Erzeugen von Realisierungsreferenzen . . . . .	60
3.3. Erzeugen von Beziehungen zwischen Modellelementen . . . . .	61
<b>4. Evaluation</b>	<b>65</b>
4.1. Laufzeitverhalten von Batch-Übersetzungen . . . . .	66
4.2. Laufzeitverhalten der Synchronisierung . . . . .	68
<b>5. Verwandte Arbeiten</b>	<b>74</b>
<b>6. Zusammenfassung</b>	<b>77</b>
<b>A. Anhang: Vorgestellte TGG-Regeln zum Synchronisieren von SA- mit LA-Systemmodellen</b>	<b>79</b>
A.1. Erzeugen von Elementen mit einer Realisierung . . . . .	79
A.2. Erzeugen von zusätzlichen Realisierungen . . . . .	82

A.3. Erzeugen von Beziehungen zwischen Modellelementen . . . . .	89
<b>B. Weitere Messergebnisse des Laufzeitverhaltens einer Synchronisierung</b>	<b>90</b>
<b>Literaturverzeichnis</b>	<b>95</b>

# 1. Einleitung

---

Die Entwicklung und Wartung komplexer Systeme gestaltet sich oft als einen schwierigen Prozess, da es meist verschiedene Interessengruppen gibt, welche Anforderungen und Ziele für das System haben und des Weiteren unterschiedliche Kompetenzen aufweisen. Dementsprechend ergibt es Sinn, für die Interessengruppen angepasste *Sichten* auf das System bereitzustellen, welche jene Informationen abstrahieren, die für die Interessengruppe nicht von Relevanz sind. Das *Model-Based Systems Engineering (MBSE)* bietet Methoden an, mit welchen das Bereitstellen von angepassten Sichten ermöglicht werden kann, indem ein Gesamtsystem durch eine Reihe von Modellen beschrieben wird. Diese Modelle stellen bestimmte Teilespekte des Systems dar und lassen sich als Sichten auf das Gesamtsystem sehen.

*Architecture Analysis and Design Integrated Approach (ARCADIA)* ist eine bekannte MBSE-Methode, welche vom französischen Konzern Thales entwickelt wurde. Diese Methode besteht aus fünf verschiedenen Schritten. In jedem dieser Schritte wird das System ausgehend von jeweils anderen Aspekten betrachtet und mehrere Sichten werden erstellt. Die Sichten jedes nachfolgenden Schrittes stellen das System weniger abstrakt dar und verfeinern die Sichten des vorherigen ARCADIA-Schrittes. Dementsprechend muss jedes Modellement einer Sicht eines ARCADIA-Schrittes in mindestens einer Sicht des folgenden ARCADIA-Schrittes repräsentiert werden. Des Weiteren kann diese Repräsentation mit weiteren Informationen angereichert werden. Da die Sichten dasselbe System mit variierendem Abstraktionsniveau darstellen und in der Summe dasselbe Gesamtsystem beschreiben, sind die Sichten stark miteinander verwandt. Beispielsweise werden die Anforderungen der System-Nutzer im ersten Schritt in ein Modell übertragen. Basierend auf diesem Anforderungsmodell lässt sich im folgenden ARCADIA-Schritt ein Modell der System-Funktionen anlegen, welche die Anforderungen implementieren.

Das Open-Source-Werkzeug *Capella* implementiert die ARCADIA-Methode und bietet ein Modellier-Rahmenwerk zum Erstellen und Verwalten der Sichten an. Eine vollständige mit Capella angelegte Systembeschreibung beinhaltet jeweils mindestens eine Sicht für jeden der fünf ARCADIA-Schritte. Da die Sichten alle dasselbe System beschreiben und die Sichten eines ARCADIA-Schrittes jeweils eine spezifischere Darstellung der Sichten des vorherigen Schrittes sind, überlappt sich teilweise der Informationsgehalt der Sichten. Somit müssen vorgenommene Änderungen an einer Sicht möglicherweise auch in andere Sichten propagierte werden, damit das System konsistent dargestellt wird. Dementsprechend kann das Arbeiten mit verschiedenen Sichten Probleme beim Umsetzen und Einpflegen von Änderungen mit sich bringen. Es handelt sich hierbei um ein *Synchronisierungsproblem*.

Sukzessive Änderungen eines Capella-Nutzers an einem Capella-Projekt können in der Regel durch Capella selbst propagierte werden. Beim Entfernen eines Modellementes werden für den Nutzer alle korrespondierenden Elemente aus spezifischeren Sichten von Capella automatisch entfernt. Sollte der Capella-Nutzer ein Element zu einer Sicht hinzufügen, so kann Capella den Nutzer dabei unterstützen, eine spezifischere Repräsentation des Elementes für den nächsten ARCADIA Schritt anzulegen. Während wie beschrieben einzelne Änderungen nacheinander eingepflegt werden können, ist es mittels Capella nicht möglich, simultan mit mehreren Nutzern an einem Capella-Projekt zu arbeiten und Änderungen konsistent einzupflegen. Wenn

Capella-Nutzer mehrere Modelle gleichzeitig bearbeiten, können ihre Änderungen sich widersprechen und somit in Konflikt zueinander stehen. Diese Konflikte müssen erkannt und aufgelöst werden. Die kommerzielle Lösung *Team for Capella*<sup>1</sup> wirbt mit dem Angebot Capella zu erweitern und das simultane Bearbeiten eines Capella-Projekts möglich zu machen. Jedoch basiert diese Lösung auf einem pessimistischen Sperren von Modellelementen und zugehörigen Elementen in anderen Modellen, damit Änderungen nicht in Konflikt zueinander stehen können. Editiert ein Capella-Nutzer also ein Modellelement, so wird das jeweilige Element und Elemente aus anderen Sichten mit gemeinsamem Informationsgehalt gesperrt. Diese Sperre bleibt so lange erhalten, bis sie manuell wieder aufgehoben wird. Des Weiteren funktioniert das Bearbeiten nur, wenn der Capella-Nutzer mit einem Server verbunden ist, auf welchem Team for Capella installiert ist. Allerdings kann das Sperren von Modellelementen und der Zwang zu einer konstant bestehenden Serververbindung einschränkend sein, da für die Capella-Nutzung ein stabiler Internetzugang erforderlich ist. Ein alternativer Ansatz wäre das lokale Editieren und Zusammenführen von Modellinstanzen, das anschließende Auflösen von Konflikten und Propagieren von Änderungen, wie schon von anderen Versionsverwaltungslösungen wie *Git* oder *SVN* bekannt ist. Ein solcher Ansatz ist aber noch nicht für Capella implementiert worden.

Diese Thesis schlägt das Verwenden von *Tripel-Graph-Grammatiken (TGG)* [1] zur Konsistenzerhaltung von Capella-Modellen vor. Dabei werden Capella-Modelle als Graphen interpretiert, während ein Satz von TGG-Regeln die Sprache aller zueinander konsistenten Modelle beschreibt. Aus dieser Sprache können anschließend Modell-Synchronisierer und -Übersetzer abgeleitet werden. Zuvor war es nur möglich, Änderungen an zwei Modellen mittels Synchronisierer sequenziell einzupflegen. Beim sequenziellen Einfügen wurden zuvor nur Änderungen aus einem Ursprungsmodell betrachtet und diese in das Zielmodell propagiert, ohne Änderungen am Ursprungsmodell vorzunehmen. Erst anschließend sind Änderungen am Zielmodell auch zurück in das Ursprungsmodell propagiert worden. Dabei wurde das Zielmodell ebenfalls nicht verändert. Allerdings ist es seit Kurzem mithilfe des Graphtransformationswerkzeugs *eMoflon::IBeX* möglich, zwei sich gleichzeitig ändernde Modelle synchron zueinanderzuhalten [2]. Es können nun Änderungen an beiden Modellen simultan in beide Richtungen propagiert und dabei entstehende Konflikte erkannt und aufgelöst werden.

Das Ziel dieser Thesis ist es, herauszufinden, ob mittels TGG Capella-Modelle synchron gehalten werden können. Falls dies nicht möglich ist, sollen die fehlenden Ausdrucksmöglichkeiten von TGG zum Synchronisieren von Capella-Modellen dokumentiert werden. Für die Capella-Modelle, für welche TGG infrage kommen, soll im Anschluss jeweils ein TGG-Regelsatz erstellt werden. Mit diesem Regelsatz und eMoflon::IBeX sollen anschließend Synchronisationsprototypen entstehen, an denen unter anderem das Laufzeitverhalten in verschiedenen Szenarien evaluiert wird.

---

<sup>1</sup><https://www.obeosoft.com/en/team-for-capella>

## 2. Grundlagen

---

Die Komplexität von Systemen steigt stetig [3], womit sich die erfolgreiche Umsetzung und Wartung jener Systeme immer fehleranfälliger und teurer gestaltet [4]. Das *Systems Engineering (SE)* bietet eine Methodik, wie ein System überlegt geplant und umgesetzt werden kann, um somit der steigenden Komplexität und Anforderungen moderner Systeme gerecht zu werden. Unterstützend zur Umsetzung und Planung von Systemen können Werkzeuge verwendet werden, welche eine SE-Methode implementieren und somit SE-Vorgehensweisen mit Entwicklungs- und Planungshilfen kombinieren. Capella<sup>1</sup> ist ein solches Werkzeug zum Entwerfen und Planen von Systemen, welches die ARCADIA-Methode implementiert. Mit Capella werden Modelle des Systems ausgehend von verschiedenen Sichten erstellt, um die Zusammensetzung und Umsetzung zu planen. Dabei hat Capella das Defizit, dass das Bearbeiten von unterschiedlichen Modellen eines Systems nicht von verschiedenen Nutzern gleichzeitig ohne Weiteres möglich ist. Ein momentaner Ansatz zum gleichzeitigen Bearbeiten von Capella-Modellen basiert auf dem Teilen einer zentralen Capella-Instanz, mit welcher sich Nutzer über das Internet verbinden können und wird von der proprietären Capella-Erweiterung Team for Capella<sup>2</sup> implementiert. Editiert ein Nutzer auf dieser Instanz ein Modellelement, so wird dieses und dessen nähere Umgebung im Modell für alle anderen Nutzer gesperrt. Durch dieses Sperren wird verhindert, dass zwei Nutzer in Konflikt zueinander stehende Änderungen an einem Modell vornehmen. In dieser Arbeit soll untersucht werden, ob der alternative Ansatz, dass alle Nutzer ihre Änderungen dezentral an lokalen Modell-Kopien vornehmen und diese anschließend mit Tripel-Graph-Grammatiken (TGGGen) synchronisieren, möglich ist. Zueinander in Konflikt stehende Änderungen an den Modell-Kopien könnten durch die Mechanismen, welche TGGGen anbieten, erkannt und aufgelöst werden. Angesichts dessen werden im Folgenden Systeme anhand eines laufenden Beispiels eingeführt. Anschließend wird das Systems Engineering, Modelle, Metamodelle, das Model-Based Systems Engineering (MBSE), Graphtransformation, Tripel-Graph-Grammatik, ARCADIA und Capella vorgestellt, bevor weiter auf das eigentliche Thema dieser Arbeit, das Synchronisieren von Capella-Modellen, eingegangen wird. Um diese Themengebiete beispielhaft zu illustrieren, wird zunächst auf das laufende Beispiel eingegangen.

---

### 2.1. Laufendes Beispiel

---

Streamingdienste sind in der heutigen Zeit die bevorzugte Dienstleistung, um zeit- und ortsunabhängig Medien wie Musik, Film oder Radio auf Abruf konsumieren zu können. In der Regel sind diese Dienste „Cloud“-basiert und verteilen die Medien von Computer-Netzwerken über das Internet an verschiedene Nutzer und Endnutzergeräte, wie in Abbildung 2.1 beispielhaft abgebildet ist. Damit jedoch ein solcher Streamingdienst funktionieren kann, müssen für die Endnutzer des Dienstes eine Vielzahl an technischen Systemen erstellt, betrieben und gewartet werden. Unter anderem spielen Server eine zentrale Rolle, um die Medien zu speichern und an die Endnutzer zu verteilen. Auch werden Software-Systeme benötigt, die beispielsweise Benutzerkonten

---

<sup>1</sup><https://www.eclipse.org/capella/>

<sup>2</sup><https://www.obeosoft.com/en/team-for-capella>

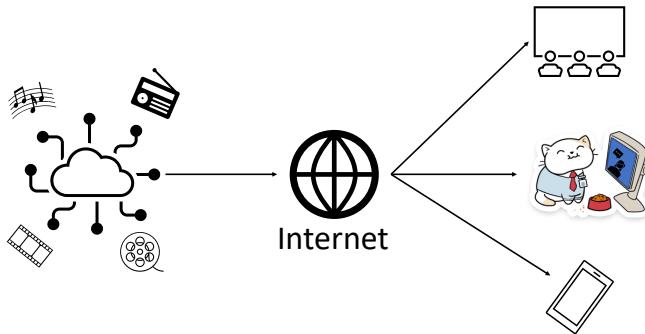


Abbildung 2.1.: Beispielhafte Abbildung eines Streamingdienstes

und Benutzersitzungen verwalten. In der Regel sind die von diesen Software-Systemen angebotenen Dienste integriert und setzen Kommunikation untereinander voraus. Die Beschreibung aller Komponenten, Funktionen und Eigenschaften des Streamingdienstes ist eine zeitaufwendige und komplexe Aufgabe, welche beispielsweise durch die Integration der Software-Systeme zusätzlich erschwert werden kann. Dies liegt vor allem auch daran, dass der Streamingdienst selbst aus vielen verschiedenen technischen Systemen besteht. Ferner werden auch Kompetenzen aus verschiedenen Fachrichtungen benötigt, um den Dienst erfolgreich erstellen und betreiben zu können. Als Beispiele für benötigte technische Systeme können die soeben genannten Server und die Software dienen. Die Software-Entwicklung, das Marketing, das Projekt-Management oder die Elektrotechnik zum Aufsetzen und Instandhalten der Server können unter anderem zu den benötigten Kompetenzen gezählt werden. Allerdings sind diese als grobe Kategorisierung von einer Vielzahl von verschiedenen Kompetenzen anzusehen. Beispielsweise werden in der Software-Entwicklung verschiedenste Kompetenzen benötigt. In der Backend-Entwicklung sind Kompetenzen zum Erstellen und Betreiben von Software-Infrastruktur, und in der Frontend-Entwicklung Kompetenzen zum Anbieten von Nutzeroberflächen, notwendig. Dementsprechend ist es unumgänglich, dass Personengruppen aufeinandertreffen, welche sich zwar mit demselben Dienst befassen, aber nicht dasselbe Verständnis oder dieselben Schwerpunkte bei der Betrachtung des Dienstes teilen. Dieses Problem kann die klare Kommunikation zwischen den Interessengruppen des Streamingdienstes behindern und kann komplexe Unterfangen wie deren Entwicklung sehr erschweren. Solche Komplikationen können unter anderem durch Kommunikationsfehler zwischen Projekt-Leitung und Investoren auftreten, was zu Fehlentscheidungen und der Missinterpretation von Anforderungen führen könnte. Dies wiederum hat zur Folge, dass Entwicklungsresultate rückgängig gemacht oder korrigiert werden müssten und somit zu einer erhöhten Schwierigkeit des Unterfangens beitragen, indem beispielsweise die Kosten des Streamingdienstes negativ beeinflusst werden.

## 2.2. Systems Engineering

Das **Systems Engineering** (SE) ist ein Tätigkeitsfeld, in welchem Methoden angewendet werden, um komplexe Systeme zu konzipieren, zu entwickeln, zu betreiben und zu warten. Die International Organization for Standardization (ISO) Norm ISO15288 definiert ein System als eine Kombination von interagierenden menschengemachten *Komponenten*. Diese Komponenten erfüllen gemeinsam als das System einen Zweck oder bieten eine Menge an *Funktionen* an [5]. Dabei sind die Komponenten voneinander abgrenzbar und erfüllen jeweils eine bestimmte Funktion innerhalb des Systems. Die Komponenten können als ein *Subsystem* oder als ein nicht weiter zu zerlegendes Element (*Systemelement*) interpretiert werden. Die Interpretation

einer Komponente als Subsystem erlaubt das erneute Zerlegen und Beschreiben. Durch wiederholtes Zerlegen und Beschreiben der Komponenten in Subsysteme und Systemelemente entsteht meist eine hierarchische Struktur [5] (siehe Abbildung 2.2). Diese Struktur kann in verschiedenen Ebenen unterteilt werden.

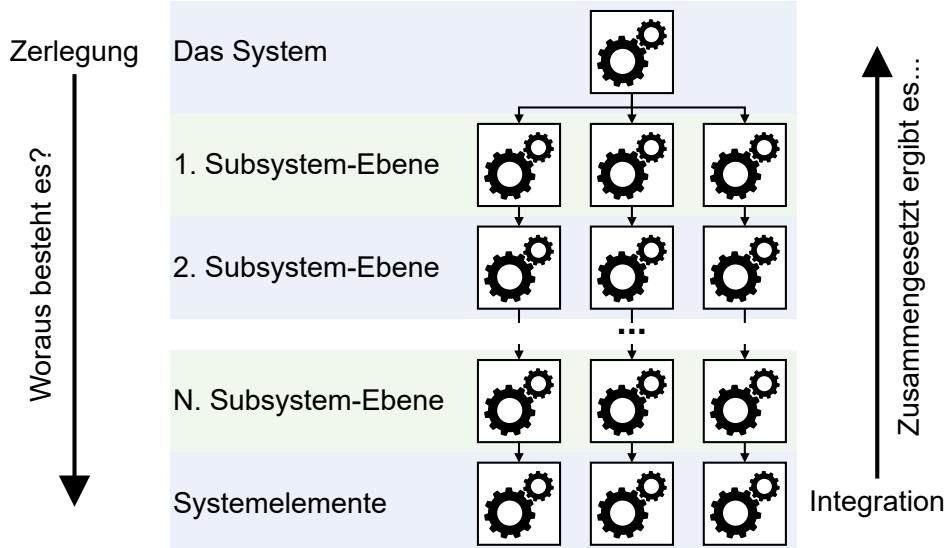


Abbildung 2.2.: Beispielhafte Hierarchie eines Systems, welches in seine Subsysteme und Systemelemente zerlegt und in Ebenen unterteilt ist.

Auf der höchsten Ebene befindet sich das System und darauffolgende Ebenen beinhalten die Subsysteme der Systeme der darüberliegenden Ebene. Die unterste Ebene beinhaltet die Systemelemente. Wird ein System in eine solche Hierarchie zerlegt, ergibt sich direkt eine Modularisierung des Systems. Die Begegnung der Komplexität des Systems mit Abstraktion wird durch diese Modularisierung gefördert und kann als Wahl einer Subsystem-Ebene zur Betrachtung und Untersuchung des Systems und dessen Komponenten umgesetzt werden. Somit kann eine Systembeschreibung mit zunehmender Detailliertheit iterativ und strukturiert vorgenommen werden, indem jeweils Ebenen nacheinander, angefangen beim System und endend bei Systemelementen, analysiert und beschrieben werden. Eine solche strukturierte Systembeschreibung durch Modularisierung eines Systems und anschließender Beschreibung der Elemente einer Ebene wird im Folgenden anhand des laufenden Beispiels aus Abschnitt 2.1 für die System-Ebene und die erste Subsystem-Ebene abstrakt vorgeführt und ist in Abbildung 2.3 visualisiert.

Der Streamingdienst ist das System, welches betrachtet werden soll. Dementsprechend ist dieses auf der höchsten Ebene abgebildet. Das System soll die Fähigkeit besitzen, Nutzern Unterhaltung in Form von Medien auf Anfrage anzubieten. Um dies zu erreichen, benötigt es unter anderem ein Backend und ein Frontend. Diese sind als Subsysteme der ersten Ebene anzusehen, und sind somit als solche in der Hierarchie abgebildet. Das Backend soll dem Streamingdienst alle Tätigkeiten ermöglichen, welche erledigt werden müssen, damit der Nutzer Medien auf Anfrage konsumieren kann. Im Gegensatz dazu soll das Frontend eine Schnittstelle zwischen dem Backend und dem Nutzer realisieren, damit der Nutzer dem Dienst in einer einfachen Art und Weise mitteilen kann, welche Medien konsumiert werden sollen. Das Backend-Subsystem lässt sich erneut in die Subsysteme Content-Provider und Load-Balancer zerteilen, während sich das Frontend beispielhaft in die Subsysteme Medienwiedergabe-Software und einer graphischen Nutzeroberfläche zerlegen lässt. Der Content-Provider soll in diesem Beispiel als Subsystem zum Bereitstellen der Medien dienen. Der Load-Balancer soll zur Verwaltung des Netzwerks und der Verbindungen zu den Nutzern eingesetzt werden, sodass

es während des Streams nicht zu Übertragungseinbrüchen kommt und so die Nutzererfahrung bei Gebrauch des Dienstes negativ beeinflusst wird. Eine graphische Nutzeroberfläche ist notwendig, damit die Nutzer des Streamingdienstes in einer bequemen Art und Weise das Angebot der Medien durchsuchen können. Allerdings benötigt das Konsumieren der gewählten Medien auch eine Medienwiedergabe-Software, um eine einfache Wiedergabe des Streams zu ermöglichen. Somit ergibt sich die zweite Subsystem-Ebene in Abbildung 2.3 aus dem Content-Provider, dem Load-Balancer, der Medienwiedergabe-Software und der graphischen Nutzeroberfläche. Als Folge dessen sind Content-Provider und Load-Balancer als Subsysteme des Backends, und Medienwiedergabe-Software und graphische Nutzeroberfläche als Subsysteme des Frontends in der zweiten Subsystem-Ebene in Abbildung 2.3 abgebildet.

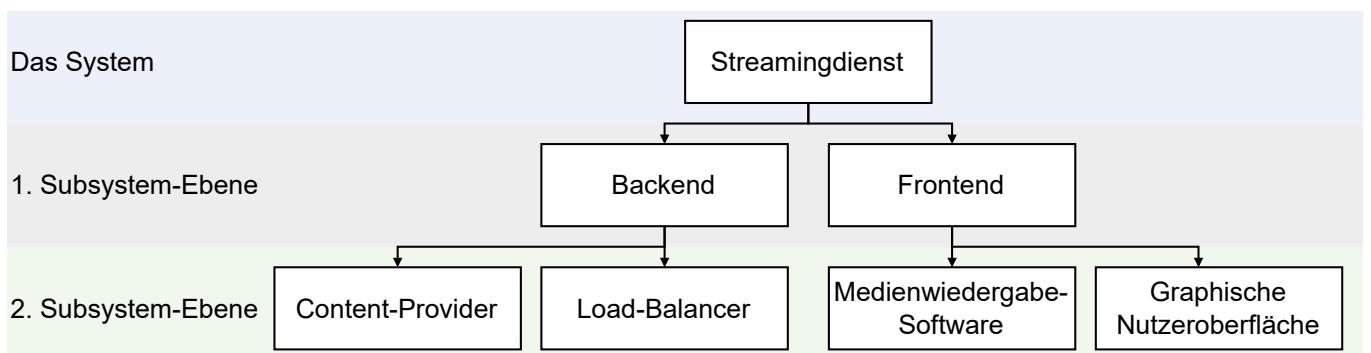


Abbildung 2.3.: Beispielhafte Zerlegung des Streamingdienstes aus Abschnitt 2.1 in eine Hierarchie wie in Abbildung 2.2.

Im Mittelpunkt einer SE-Methodik steht der Umgang mit der Komplexität eines Systems [6], um das Scheitern und das Entstehen von vermeidbaren Kosten und Problemen, die dem Betrieb des Systems entgegenstehen, zu verhindern. Dabei steht insbesondere das Entwerfen der *Systemarchitektur*, also die Zusammensetzung und innere Funktionsweise des Systems, im Fokus. Dies liegt daran, dass mit fortschreitendem Entwicklungsstand des Systems der Aufwand steigt, um Änderungen einzubringen und Probleme auszubessern. Des Weiteren wird das Einbringen von Änderungen mit fortschreitendem Entwicklungsstand meist teurer, da Anpassungen potenziell ein Neuentwickeln ganzer Subsysteme nach sich ziehen kann. Dementsprechend entstehen meist ein Großteil der Gesamtkosten eines Systems hauptsächlich beim Betreiben und Warten. Es ist allerdings zu beachten, dass sich diese hohen Kosten teilweise darauf zurückführen lassen, dass sich Systeme meist viel länger in Betrieb und Wartung befinden als in den anderen Phasen des Lebenszyklus. Diese asymmetrische Verteilung von Kosten und Zeit auf die späten Phasen im Lebenszyklus ist gewiss auch dem geschuldet, dass Systeme oft so schnell wie möglich nutzbar sein sollen und dessen Weiterentwicklung und Wartung als Dienstleistungen vermarktet werden. Somit fallen weitere Entwicklungskosten in späteren Lebensphasen an, welche nicht Teil der initialen Entwicklungskosten des Systems sind. Es ergibt Sinn, die initiale Architektur des Systems für Erweiterbarkeit zu optimieren und grundlegende Fehler zu vermeiden, damit die Weiterentwicklung und Wartung als Dienstleistung bestmöglich ohne negativen Einfluss auf zur Verfügung stehende Ressourcen angeboten werden kann. Das Ziel einer SE-Methode ist es, durch die intensive Auseinandersetzung mit der Systemarchitektur und einer systematischen Planung dessen zu versuchen, Architektur-Fehler so früh wie möglich aufzudecken und zu verhindern, damit die Menge an nachträglichen Änderungen und Ausbesserungen am System minimiert werden kann. Diese Zusammenhänge sind in Abbildung 2.4 in einem „typischen“ Entwicklungsprozess visualisiert. Der obere Balken stellt die grundlegenden Phasen des „typischen“ System-Lebenszyklus mit fortschreitender Zeit von links nach rechts dar. Der sich darunter befindende Pfeil visualisiert die Einflussmöglichkeit auf die Systemarchitektur. Je weiter links, das heißt umso früher im Lebenszyklus,

desto höher ist die Einflussmöglichkeit auf die Architektur. Umgekehrt, je weiter rechts, desto geringer ist die Einflussmöglichkeit. Die Pfeile darunter beschreiben den Kostenaufwand im Laufe des Lebenszyklus. Hierbei entstehen in den letzten zwei Phasen, also der Produktion oder Betriebsvorbereitung und Wartung und der Einstellung, circa 72% der Gesamtkosten, während die Kosten aller anderen Phasen sich nur zu 28% der Gesamtkosten aufaddieren. [7]

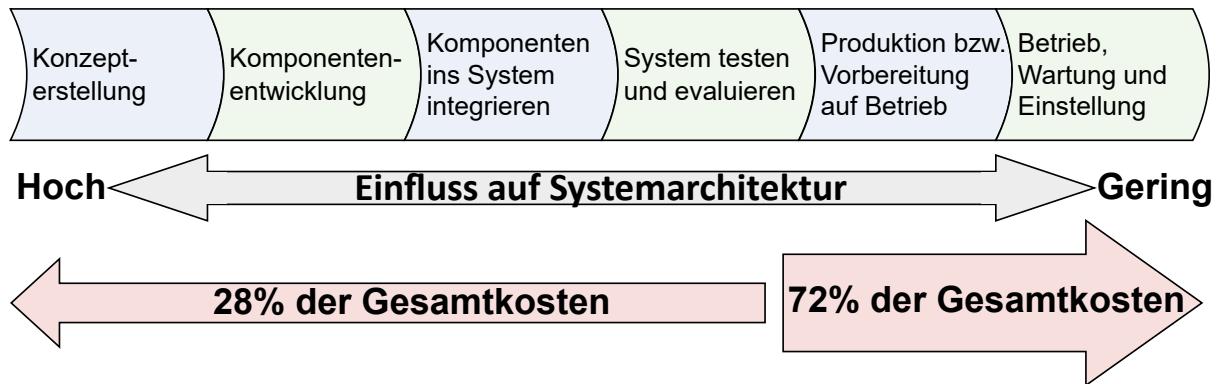


Abbildung 2.4.: Exemplarische Darstellung eines „typischen“ Lebenszyklus mit Gesamtkosten-Erwartungswerten und Darstellung der Einflussmöglichkeit auf Systemeigenschaften nach John V. Farr [7, Abbildung 3].

Die SE-Vorgehensweise ist in den Prozessen einer SE-Methodik definiert. Ein SE-Prozess besteht aus einer Reihenfolge von Aktivitäten, deren Ausführung durch eine SE-Methode beschrieben wird [8]<sup>3</sup>. International Council on Systems Engineering (INCOSE) fasst die Motivation von SE-Prozessen in die folgenden grundlegenden vier Ziele zusammen [6]:

1. Das Problem verstehen und verstehen, wofür das System benötigt oder gewollt wird.
2. Die Lösung zum Problem verstehen.
3. Risiken und Kosten des Systems bewerten.
4. Bewerten, ob das System das Problem auch löst.

Mit diesen vier Zielen lässt sich der generische SE-Entwicklungsprozess, dargestellt in einem „V“-Modell angelehnt an das Ursprüngliche von Forsberg und Mooz [10], mit Abbildung 2.5 beschreiben. Der Prozess beginnt mit dem Zerlegen und Beschreiben des Systems. Nach anschließender Entwicklung der Systemkomponenten werden Resultate analysiert und validiert. Wichtig ist hierbei, dass wie im originalen „V“-Modell Rückgriffe zu früheren Entwicklungsstufen nicht nur möglich, sondern auch notwendig sind. Beispielsweise sei nach Analyse und Validation eines Subsystems festgestellt worden, dass das Verhalten des Subsystems nicht wie erwartet ist. So entsteht ein Zyklus im Entwicklungsprozess und es wird erneut bei der Zerlegung und Beschreibung der Subsysteme begonnen. Im Gegensatz zum „typischen“ Entwicklungsprozess in Abbildung 2.4 sind hier die einzelnen Entwicklungsschritte nicht eindeutig voneinander trennbar und somit eher als interne Entwicklungszyklen zu betrachten.

<sup>3</sup>transitives Zitieren nach James N. [9].

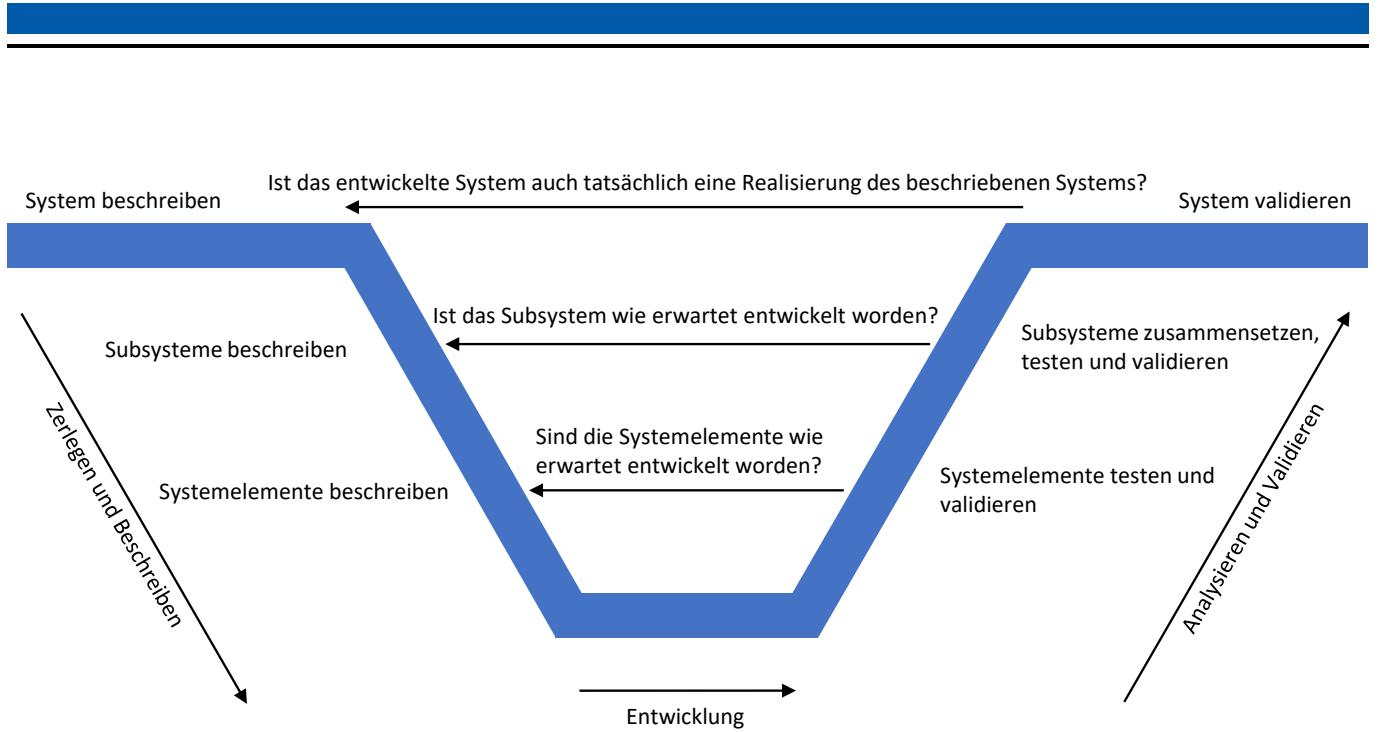


Abbildung 2.5.: Generischer SE-Entwicklungsprozess. Darstellung orientiert sich an dem „V“-Modell von Forsberg und Mooz [10]

## 2.3. Model-Based Systems Engineering (MBSE)

Das Model-Based Systems Engineering (MBSE) ist eine Weiterentwicklung des Systems Engineering (SE), welches Prinzipien des *Model-based Engineering (MBE)* verwendet und als eine Subdisziplin dessen angesehen werden kann. Im Gegensatz zum klassischen SE werden im MBSE und MBE nicht primär Dokumente, wie Lasten- oder Pflichtenhefte, zum Kommunizieren, Dokumentieren und Spezifizieren eingesetzt, sondern hauptsächlich Modelle [11]. Im Weiteren wird vorerst erläutert, was Modelle sind und anschließend weiter auf das MBSE und dessen Einordnung in den SE- und MBE-Disziplinen eingegangen.

Modelle sind im Grunde vereinfachte Repräsentationen der Realität und bilden ausgewählte Eigenschaften, Verhaltensweisen oder Abläufe ab. Abbildung 2.1 des laufenden Beispiels ist ein solches Modell, wie im Folgenden argumentiert wird. Die grundlegenden Eigenschaften von Modellen sind auf ihr *Abbildungs-, Verkürzungs- und pragmatisches Merkmal* zurückzuführen [12]. Das Abbildungsmerkmal eines Modells ist, dass jedes Modell stets etwas anderes abbildet oder repräsentiert. Abbildung 2.1 erfüllt dieses Merkmal, da es grundlegenden Eigenschaften und Zusammenhänge des Streamingdienstes, wie dass Nutzer Medieninhalte über das Internet konsumieren können, repräsentiert. Des Weiteren abstrahiert und vereinfacht jedes Modell (Verkürzungsmerkmal), das heißt, dass nicht alle Eigenschaften und Zusammenhänge im Modell repräsentiert werden. Auch dieses Merkmal wird von Abbildung 2.1 erfüllt. Es wird unter anderem nicht dargestellt, wie Nutzer mit dem Internet verbunden sind oder wie und welche Medien tatsächlich zu den Nutzern gelangen. Die Auswahl der dargestellten Eigenschaften und Zusammenhänge ist durch den Anwendungszweck des Modells bestimmt und kann nur im Kontext des Anwendungsfalles als eine Repräsentation angesehen werden (pragmatisches Merkmal). Abbildung 2.1 weist auch das pragmatische Merkmal auf und ist nur zur Vermittlung der Grundidee eines Streamingdienstes als Repräsentation verwendbar. Beispielsweise ist Abbildung 2.1 nicht als Repräsentation im Kontext der Software-Entwicklung für den Dienst verwendbar. Es sind keine für die Entwicklung relevanten Informationen, wie die Menge an Software-Komponenten und deren Funktionen, dargestellt. Dieses Wissen ist allerdings unabdingbar, um die Komponenten zu implementieren. Abbildung 2.1

weist alle Merkmale eines Modells auf und ist dementsprechend als ein Modell zu betrachten.

Das MBSE ist, wie bereits erwähnt, als eine Subdisziplin des MBE einzuordnen. Im *Model-driven Engineering (MDE)*, eine weitere Subdisziplin des MBE, werden Modelle nicht nur zur Unterstützung verwendet, sondern die Modelle sind ebenfalls ein Produkt des Ingenieursprozesses. Dem MDE unterzuordnen ist das *Model-driven Development (MDD)*. Hier sind Modelle ein treibender Faktor der (meist Software-)Entwicklung, indem Modelle zum automatisierten Erstellen von (Software-)Implementation verwendet werden. Eine Verfeinerung des MDD ist die von der *Object Management Group (OMG)* vorgeschlagene *Model-driven Architecture (MDA)*, welches im Grunde MDD zum Entwickeln von System-Architekturen unter Verwendung von OMG-Produkten ist [13]. MBSE als Subdisziplin des MBE kann je nach Methodik Elemente aus den Disziplinen MDE, MDD oder MDA beinhalten. Zusammenfassend kann MBSE als eine Vereinigung von MBE und SE verstanden werden, wie in Abbildung 2.6 visualisiert ist.

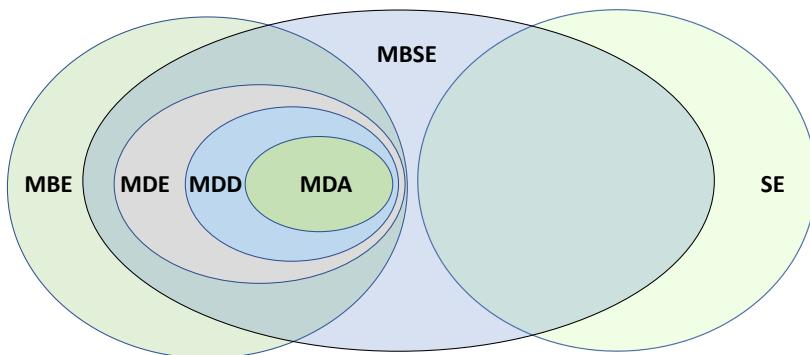


Abbildung 2.6.: Zusammenhang zwischen SE, MBSE und MBE-Disziplinen

Die Vorteile einer MBSE-Methodik gegenüber einer SE-Methodik sind auf das Modellieren des Systems zurückzuführen. Nach Bajzek et al. [14] solle das modellgestützte Vorgehen zu einem intensiven Auseinandersetzen mit dem System führen und somit die Durchdrachtheit des Systems auch in frühen Entwicklungsphasen fördern. Auch solle die Qualität der Systembeschreibungen verbessert werden, das heißt, dass Mehrdeutigkeiten in der Systembeschreibung verringert werden sollen, so Friedenthal et al. [15]. Des Weiteren solle ebenfalls eine höhere Vollständigkeit der Systembeschreibung und eine bessere Rückverfolgbarkeit von Anforderungen angezielt werden. Dies solle durch eine höhere Einbindung der Interessengruppen in die Entwicklung des Systems durch das Anbieten von an die Interessen und Kompetenzen der jeweiligen Interessengruppen angepasste Modelle erreicht werden [15]. Mit diesen angepassten Modellen wird es ermöglicht, dass Interessengruppen verschiedenster Domänen eine für sie passende Repräsentation des Systems erhalten und mit dem dadurch erlangtem Verständnis besser ihre Ziele und Wünsche kommunizieren und eigene Sachkenntnisse besser einbringen können. Des Weiteren wird so ermöglicht, dass jede Interessengruppe ein für sie angemessenes Niveau an Abstraktion bei der Systembetrachtung einhalten kann. Beispielsweise kann so verhindert werden, dass ein Software-Entwickler sich mit den technischen Details der Server-Hardware auseinandersetzen muss und sich auf die Beschreibung der Benutzeroberfläche, welche für den Streamingdienst entwickelt werden soll, konzentrieren kann.

## 2.4. Metamodelle

Wie zuvor erläutert, wird mit den während einer MBSE-Methodik entstehenden Modellen angestrebt, dass die verschiedenen Interessengruppen in den Entwicklungsprozess der System-Architektur eingebunden werden. Des Weiteren sollen auch die Dokumente einer regulären SE-Methodik mit diesen Modellen ersetzt werden. Damit dies allerdings möglich ist, ist es notwendig, dass die Modelle sinnvoll aufgebaut sind, damit der dargestellte Inhalt im Kontext des Systems eindeutig ist und von den Interessengruppen nicht verschieden interpretiert werden kann. Es ist zu berücksichtigen, dass Modelle auch nur im Kontext des Anwendungszweckes, für welches sie erstellt wurden, gültig sind. Somit ergibt es Sinn, innerhalb der verwendeten MBSE-Methodik den Verwendungszweck verschiedener Modellarten einheitlich festzulegen. Außerdem sollte die Struktur und der Anwendungszweck der zu verwendenden Modelle festgelegt und beschrieben werden. Wird anschließend diese Beschreibung mit allen Interessengruppen geteilt und als Spezifikation der Modelle verwendet, so kann verhindert werden, dass Interessengruppen Modelle missverstehen, oder dass Modelle unstrukturiert erstellt werden.

Im MBSE sind Dokumente mit Modellen zu ersetzen und somit erfolgt das Beschreiben und Spezifizieren der verwendeten Modelle selbst mit einem Modell. Dieses Modell, welches andere Modelle beschreibt und spezifiziert, wird *Metamodell* genannt. Auch die Struktur und Zusammensetzung dieses Metamodells kann selbst wieder mit einem Modell beschrieben und spezifiziert werden. Modelle, welche Metamodelle beschreiben, werden *Meta-Metamodell* genannt, da sie Metamodell des Metamodells sind. Theoretisch kann das Erstellen von Metamodellen von Modellen unendlich oft wiederholt werden und es entsteht eine unendliche Modellhierarchie bestehend aus verschiedenen Ebenen. Die unterste Ebene dieser Hierarchie ist das Modell und jede nächst höhere Ebene ist das Metamodell der sich darunter befindenden Ebene. Somit abstrahiert jede Ebene von der vorherigen. In der Praxis lässt sich allerdings nicht unendlich oft ein Metamodell für ein Modell wiederholt erstellen, da ab einem gewissen Punkt der Inhalt eines Metamodells so abstrakt ist, dass es sich selbst beschreibt und spezifiziert. Anstatt einer unendlichen Hierarchie entsteht in der Praxis eine endliche Hierarchie, wie die in Abbildung 2.7. Hier ist eine Hierarchie, bestehend aus vier Ebenen, dargestellt. Die höchste Ebene beinhaltet das Meta-Meta-Metamodell, während die tiefste das Modell enthält.

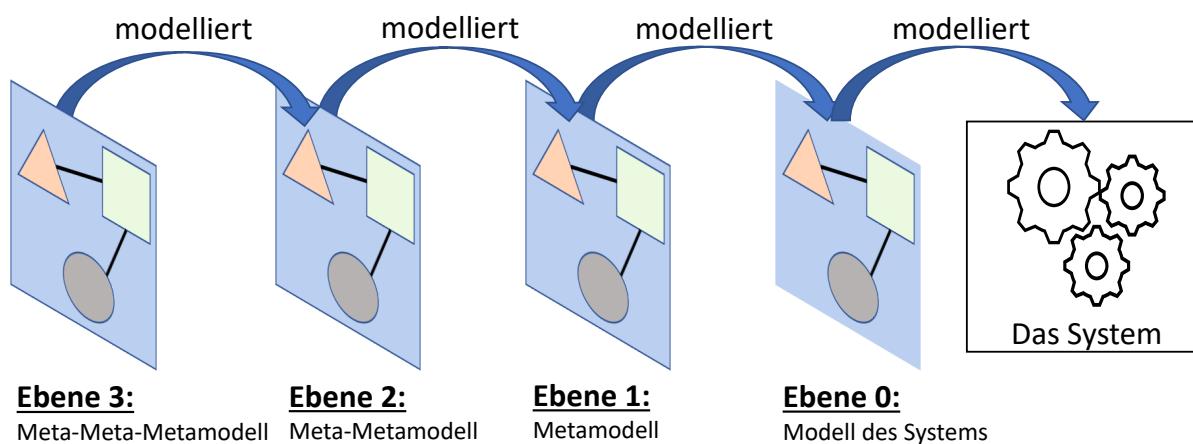


Abbildung 2.7.: Beispielhafte Modellhierarchie bestehend aus insgesamt vier Ebenen

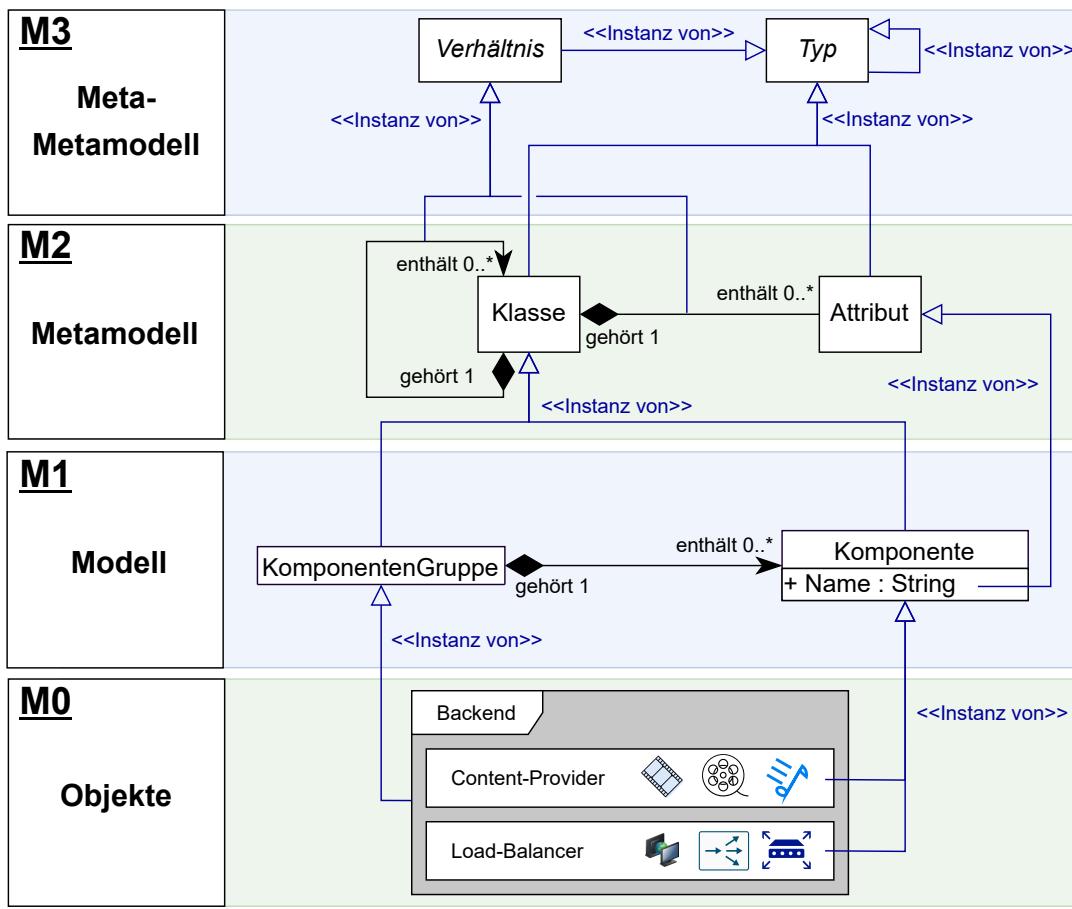


Abbildung 2.8.: Beispielhafte Visualisierung der Modellhierarchie nach **Meta Object Facility (MOF)**.

Damit vorhandene Modellierungswerkzeuge und -hilfen wiederverwendet werden können, ist es praktisch, eine geteilte Modellhierarchie festzulegen. Infolgedessen wird meist *Meta Object Facility (MOF)* [16], ein in der Industrie gängiger Modellhierarchie-Standard, eingesetzt. MOF ist von der OMG entwickelt worden und schlägt eine Hierarchie bestehend aus den vier Ebenen **M0** bis **M3** vor. Im Folgenden wird MOF mit Abbildung 2.8 anhand des laufenden Beispiels vorgestellt. Die **M0**-Ebene beinhaltet jene Objekte, welche modelliert werden sollen. Im Kontext des MBSE ist die Realisierung eines Systems mit dessen Subsystemen und Systemelementen der **M0**-Ebene zuzuteilen. Hierfür sind das **Backend**-Subsystem und dessen **Content-Provider**- und **Load-Balancer**-Subsystem aus Abbildung 2.3 beispielhaft dargestellt. Die **M1**-Ebene ist das eigentliche Modell, welches das System repräsentiert. Dieses Modell besteht hier aus **KomponentenGruppe**- und **Komponente**-Modellelementen. Des Weiteren sind **Komponenten** in einer Komposition-Beziehung mit **KomponentenGruppe**, womit **Komponenten** in **KomponentenGruppen** enthalten sein können. Das **Backend** kann als eine *Instanz* der **KomponentenGruppe** gesehen werden. Der **Content-Provider** und der **Load-Balancer** werden in dem Beispiel nicht genauer untersucht und werden somit als Systemelemente betrachtet. Diese Systemelemente können als *Instanzen* des **Komponente**-Modellelements gesehen werden. Das Metamodell, welches das Modell der **M1**-Ebene spezifiziert, ist Teil der **M2**-Ebene. In diesem Beispiel wird ein Metamodell, bestehend aus **Klassen** und **Attributen**, eingesetzt. Das Metamodell beschreibt, dass Modelle aus **Klassen** bestehen. **Klassen**-Instanzen besitzen **Attribute** und können andere **Klassen**-Instanzen *enthalten*. In diesem Beispiel sind **Komponente**-Modellelemente *Instanzen von* **Klassen**, so wie deren **Namen** *Instanzen von* **Attributen** sind. Die Struktur des Metamodells wird mit dem Meta-Metamodell, welches der **M3**-Ebene

zuzuordnen ist, beschrieben. Dieses besteht in diesem Beispiel aus den Elementen **Typ** und **Verhältnis**. Die Elemente des Metamodells, **Klasse** und **Attribut**, sind *Instanzen des Typs*. Die Beziehungen zwischen **Klassen** und anderen **Klassen** oder **Attributen** werden hier mit dem **Verhältnis**-Meta-Metamodellelement beschrieben. **Typen** sind selbst Instanzen von **Typen**. Auch sind **Verhältnis**-Elemente *Instanzen von Typen*. Das Modell der M3-Ebene spezifiziert sich selbst und ist somit dessen eigenes Metamodell.

Die während einer MBSE-Methodik entstehenden, an die Interessengruppen angepassten, Modelle werden in der Praxis mit MOF spezifiziert. Dabei werden verschiedene *Modeling Languages (MLs)* erstellt, welche als Spezifikation für die angepassten Modelle dienen. Diese MLs sind wie auch die Modelle für eine Interessengruppe und für einen Verwendungszweck optimiert und werden deshalb auch *Domain-specific Language (DSL)* genannt. Die Nutzung einer DSL soll das effiziente Modellieren innerhalb der Domäne, für welche sie optimiert ist, ermöglichen. DSLs können als eine Modellspezifikation eingesetzt werden, um Systemmodelle aus der Sicht der Domäne zu erstellen. Modelle, welche Systeme ausgehend von einer bestimmten Sichtweise darstellen, werden (*System-*)*Sichten* genannt. Neben dem Identifizieren und Erstellen von Sichten ist auch das Definieren von geeigneten DSLs eine Hauptaufgabe des MBSEs. Die MBSE-Methode ARCADIA und dessen komplementäres MBSE-Werkzeug Capella bieten geeignete DSLs zum Erstellen von Sichten. Diese Sichten überlappen sich allerdings teilweise im Informationsgehalt. Beispielsweise werden in Capella mehrere Sichten auf die Unterteilung des Systems in Subsysteme und Systemelemente als Komponentengruppen und Komponenten dargestellt. Diese Sichten stellen dasselbe System dar und unterscheiden sich lediglich in der Abstraktion der Unterteilung. Es muss dementsprechend sichergestellt werden, dass die verschiedenen Darstellungen dieser Unterteilung sich nicht widersprechen. Nimmt ein Capella-Nutzer eine Änderung an einer dieser Sichten vor, so muss er diese Änderungen in die anderen Sichten einfließen lassen. Wird beispielsweise eine neue Komponente einer Sicht hinzugefügt, so muss diese Komponente auch in jeder Sicht mit einer weniger abstrakten Unterteilung des Systems dargestellt werden. Fehlt die Darstellung dieser Komponente in den anderen Sichten, so stehen diese in Konflikt zueinander. Dementsprechend müssen die Sichten, welche sich Informationsgehalt teilen, synchron gehalten werden, um Widersprüche zwischen den Sichten zu vermeiden und die Darstellung des Systems fehlerfrei zu halten. Liegt in einer Sicht ein Widerspruch zu einer anderen Sicht vor, so muss dieser Widerspruch durch einen Synchronisierungsprozess behoben werden. Dieser Prozess kann entweder durch das manuelle Bearbeiten der Sichten eines Nutzers, oder mithilfe von *Graphtransformationen* automatisiert vorgenommen werden.

## 2.5. Graphtransformation

Ist eine Sicht bereits in einem inkonsistenten Zustand zu anderen Modellen, so ist es notwendig, diese Sicht zu bearbeiten, um Konsistenz wiederherzustellen. Das Bearbeiten einer Sicht lässt sich in Form von *Modelltransformationen* umsetzen. Hierbei lassen sich Modellelemente als *Knoten* und die Beziehungen zwischen den Modellelementen als *Kanten* eines *Graphen* interpretieren. Modelle sind somit graphenähnliche Strukturen, weshalb sich Konsistenz-wiederherstellende Modelltransformationen als Graphtransformationen umsetzen lassen. Dies ist möglich, indem Modelle als Graphen interpretiert werden. Eine Graphtransformation ist eine Methode, mit welcher regelbasiert deklariert wird, wie ein Graph geändert werden kann. Eine Menge an Graphtransformationen wird mittels einer *Graphgrammatik*, bestehend aus *Graphtransformationsregeln* (im folgenden *Regeln* bezeichnet), beschrieben. Eine Regel beschreibt eine Transformation und besteht aus einer *left-hand side (LHS)* und einer *right-hand side (RHS)*. Die LHS stellt die Vorbedingung der Regelanwendung in Form eines Teilgraphen dar, welcher im Modellgraphen vorkommen muss. Im Gegensatz dazu beschreibt die RHS das Ergebnis der Transformation und ist somit die Nachbedingung einer Regelanwendung. Die Differenz zwischen LHS und RHS lässt sich intuitiv als eine Beschreibung der Operationen, in welche sich

die Transformation zerlegen lässt, interpretieren [17]. Kanten und Knoten, welche in der LHS und der RHS vorkommen, stellen den *Kontext* einer Regel dar. Kontext-Elemente bleiben durch eine Regelanwendung unverändert. Elemente, welche nur in der LHS und nicht in der RHS vorkommen, werden gelöscht. Jene Elemente, welche nur in der RHS und nicht in der LHS vorkommen, werden neu erstellt. Diese Zusammenhänge werden nun im folgenden anhand des laufenden Beispiels aus Abschnitt 2.1 vorgeführt.

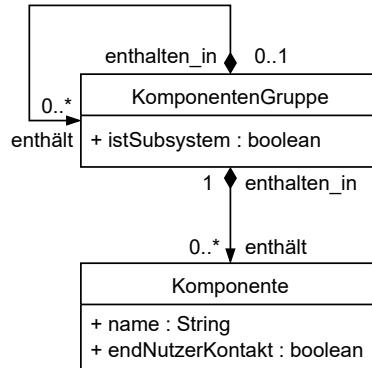
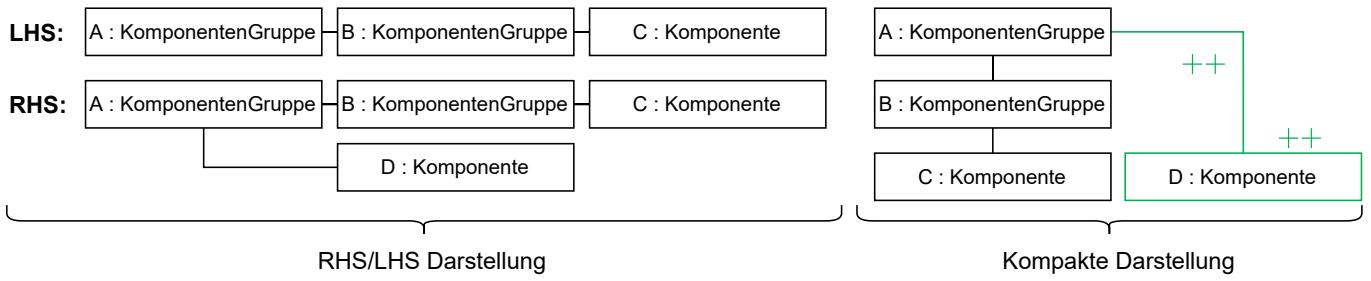


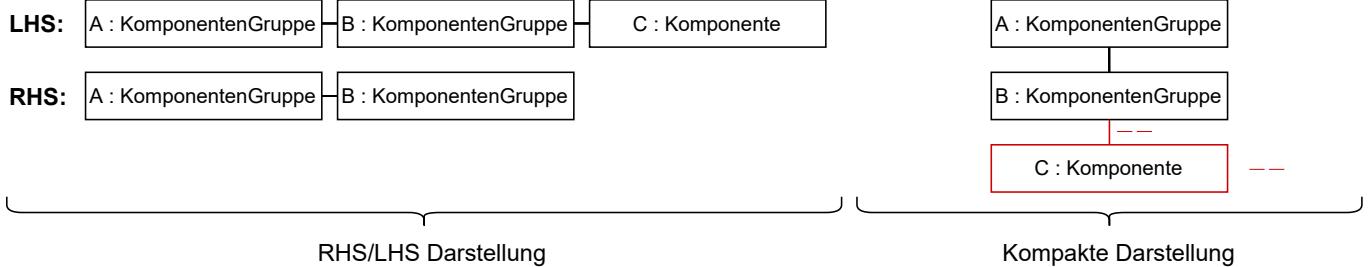
Abbildung 2.9.: Metamodell von Sichten, welche die Struktur von Subsystemen und Systemelementen abstrakt als **KomponentenGruppen** und **Komponenten** darstellt.

Zunächst wird eine neue Sicht definiert, welche das Modell der M1-Ebene aus Abbildung 2.8 erweitert. Das Metamodell ist in Abbildung 2.9 abgebildet. Diese Sicht kann während der Entwicklung des Streamingdienstes zur Visualisierung und Dokumentation der Subsysteme und Systemelemente verwendet werden, und wird in den folgenden Abschnitten zur Erläuterung von Zusammenhängen weiter verwendet. Das System und die Subsysteme des Streamingdienstes werden mit **KomponentenGruppe**-Modellelementen dargestellt. Damit zwischen Subsystemen und dem System unterschieden werden kann, besitzen **KomponentenGruppen** den *istSubsystem*-Wahrheitswert als Attribut. Ist eine **KomponentenGruppe** ein Subsystem, so ist dieses Attribut *wahr*, ansonsten ist es *falsch* und es handelt sich um das System selbst. Damit die Struktur des Streamingdienstes in der Sicht widergespiegelt werden kann, *enthalten* **KomponentenGruppen** möglicherweise unbegrenzt viele **KomponentenGruppen**, welche somit Subsysteme der **KomponentenGruppe** sind. Auch können **KomponentenGruppen** unbegrenzt viele **Komponenten** *enthalten*, welche die Systemelemente repräsentieren. Alle **Komponenten** besitzen die Zeichenkette *name* zum Speichern des Namens der Systemkomponente und den Wahrheitswert *endNutzerKontakt*, welcher angibt, ob ein regulärer Nutzer des Streamingdienstes direkt mit der **Komponente** interagieren können soll oder nicht.

Abbildung 2.10a zeigt eine Regel, mit welcher eine neue **Komponente** *D* unter Beachtung der Vorbedingung (LHS) erzeugt werden kann. Dabei wird zunächst geprüft, ob in der Sicht eine **KomponentenGruppe** *A* existiert, welche die **KomponentenGruppe** *B* enthält, in der wiederum eine **Komponente** *C* enthalten ist. Lässt sich ein solcher Teilgraph in der Sicht finden, so wird *D* als Inhalt in *A* eingefügt. In Abbildung 2.10a ist auf der linken Seite die Darstellung der LHS- und RHS-Teilgraphen zu sehen. Wie auf der rechten Seite zu sehen ist, lassen sich Regeln kompakt als einen Teilgraphen anstatt einer LHS und einer RHS darstellen. Hierbei werden Kontext-Elemente in Schwarz, neu zu erzeugende Elemente in Grün, mit einem **++** versehen, und zu löschen Elemente in Rot, mit einem **--** versehen, dargestellt. Analog zu dieser Darstellungsweise einer Regel ist in Abbildung 2.10b eine weitere gezeigt, welche eine **Komponente** *C* aus der Sicht löscht. Die Vorbedingung dieser Transformation ist, dass *C* in einer **KomponentenGruppe** *B* enthalten ist, welche selbst in einer **KomponentenGruppe** *A* enthalten ist.



(a) Beispielhafte Graphtransformationsregel, welche eine **Komponente D** in der **KomponentenGruppe A** erzeugt.



(b) Beispielhafte Graphtransformationsregel, welche eine **Komponente C** in der **KomponentenGruppe B** löscht.

Abbildung 2.10.: Beispiel Graphtransformationsregeln.

Werden nun die Regeln aus Abbildung 2.10a und Abbildung 2.10b auf den Beispiel-Graphen in Abbildung 2.11a angewendet, ändert sich dieser, wie im Folgenden beschrieben. Der Graph besteht initial aus der **KomponentenGruppe kg1**, welche die **KomponentenGruppe kg2** enthält. **kg2** enthält wiederum die **Komponente k1**. Dementsprechend sind die Vorbedingung beider Regeln erfüllt. Wird zunächst die Regel aus Abbildung 2.10a angewendet, so wird eine neue **Komponente k2** in **kg1** erzeugt, wie in Abbildung 2.11b dargestellt ist. Das anschließende Anwenden der Regel aus Abbildung 2.10b löscht die **Komponente k1**. Das Resultat der Regelanwendung ist in Abbildung 2.11c dargestellt.

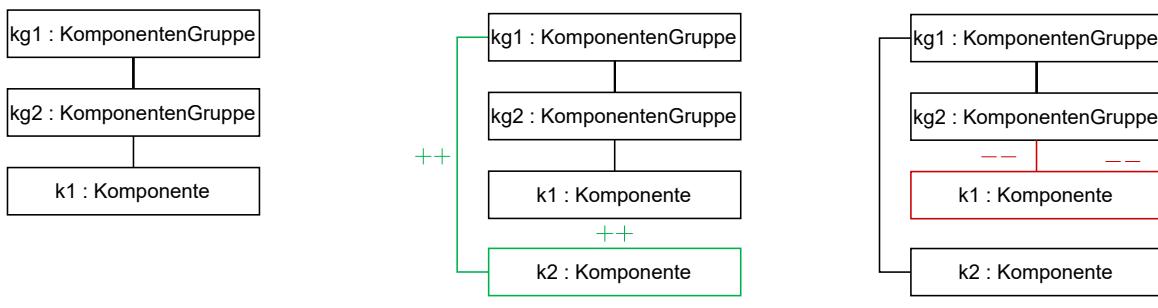


Abbildung 2.11.: Anwenden von Regeln aus Abbildung 2.10 auf Graphen in Abbildung 2.11a. Neu erzeugte Elemente in Grün, gelöschte in Rot und unveränderte (Kontext) in Schwarz dargestellt.

Es ist mittels Regeln theoretisch möglich, den Prozess zu beschreiben, wie die Konsistenz einer Sicht wiederhergestellt werden kann. Gerade dies ist allerdings nicht ohne Weiteres umsetzbar, da die Regeln nur den Zustand einer Sicht berücksichtigen. Die Anwendung von Regeln ist ohne den Kontext der anderen Sicht

zum Synchronisieren von Modellen nicht zielführend. Das Berücksichtigen einer zweiten Sicht ist allerdings mithilfe von TGGen implementierbar. Mittels einer TGG lassen sich anschließend bidirektionale Modelltransformationen herleiten, um zwei Sichten synchron zueinanderzuhalten und Inkonsistenzen zwischen diesen zu beheben.

## 2.6. Modellsynchronisation mittels Tripel-Graph-Grammatiken

Da die Interessengruppen eines Systems und dessen Systemkomponenten meist aus verschiedenen Domänen stammen, werden im MBSE verschiedene Systemsichten angelegt. Diese Sichten beschreiben gemeinsam dasselbe System, allerdings unter Berücksichtigung verschiedener Aspekte, thematischer Schwerpunkte oder variierender Abstraktionsniveaus. Der Informationsgehalt der Sichten kann sich überlappen. Dementsprechend müssen die Sichten zueinander konsistent sein, damit das System fehlerfrei und vollständig dargestellt wird. Eine Ursache von Inkonsistenzen zwischen zwei Sichten sind Änderungen an Elementen, welche eine Entsprechung in der anderen Sicht besitzen. Diese Änderungen müssen anschließend von der geänderten Sicht in die andere Sicht propagiert werden, damit keine Inkonsistenzen zwischen den Sichten hinterlassen werden. Es handelt sich bei dem Problem, Änderungen aus einer modifizierten Sicht korrekt in eine andere Sicht zu propagieren und dabei auftretende Konflikte aufzulösen, um ein *Synchronisierungsproblem*. Oft ist es allerdings der Fall, dass beide Sichten modifiziert worden sind und Änderungen nicht nur unidirektional von einer Sicht in die jeweils andere propagiert werden müssen, sondern dies auch bidirektional geschehen muss. Da im MBSE viele Sichten von dem System erstellt werden, welche Informationsgehalt teilen können, ist es auch möglich, dass mehr als zwei Sichten synchron gehalten werden müssen. Das Beheben von Inkonsistenzen zwischen zwei Sichten kann bereits sehr kompliziert sein, aber das Wiederherstellen von Konsistenz nach Änderung von vielen Sichten ist deutlich anspruchsvoller. Wie in Abschnitt 2.7 genauer erläutert wird, werden in Capella Sichten für die verschiedenen Schritte der ARCADIA-Methode angelegt. Jede dieser Sichten teilt Informationsgehalt mit einer Sicht des vorherigen und des nachfolgenden Schrittes. Es bietet sich deshalb in diesem Fall an, das Synchronisieren von vielen Sichten als das paarweise bidirektionale Lösen des Synchronisierungsproblems zwischen zwei Sichten benachbarter ARCADIA-Schritte zu implementieren. Dementsprechend wird zunächst eine Lösung des Synchronisierungsproblems benötigt, womit Änderungen bidirektional zwischen zwei Sichten propagiert werden können. Ein Ansatz, um dies zu ermöglichen, sind *Tripel-Graph-Grammatiken* (TGGen) auf Basis von *bidirektionalen Modelltransformationen*. TGGen sind ein von Schürr [1] eingeführtes Konzept, mit welchem Regelsätze deklariert werden können, die das zueinander konsistente Erzeugen eines Modellpaars beschreiben. Das Modellpaar besteht aus einem *Ursprungsmetamodell* und einem *Zielmetamodell*. Mit einer TGG-Regel wird das Erzeugen von Elementen in Ursprung- und Zielmodell deklariert. Entsprechungen zwischen Modellelementen des Ursprungsmetamodells und des Zielmetamodells werden in einem dritten Modell, dem *Korrespondenzmodell*, dargestellt. Korrespondenzmodellelemente repräsentieren die Entsprechungen und referenzieren jene sich entsprechenden Elemente im Ursprungsmetamodell und im Zielmetamodell. Dies wird anhand von Abbildung 2.12 beispielhaft demonstriert.

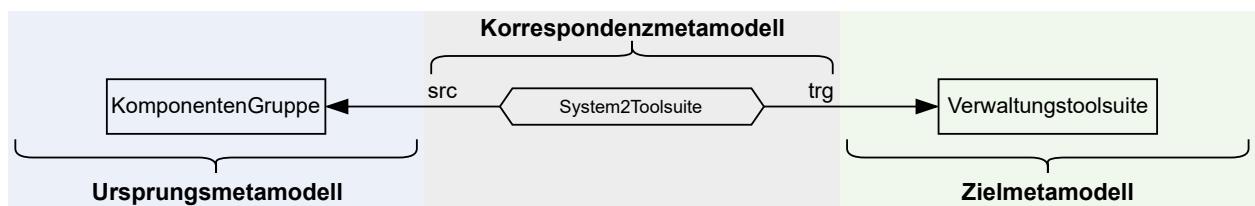


Abbildung 2.12.: Beispielhafte Visualisierung wie Korrespondenzelemente Modellelemente im Ursprungsmetamodell und Zielmetamodell über Referenzen in Bezug zueinander setzen.

Das Ursprungsmodell besteht in diesem Beispiel aus **KomponentenGruppe**-Modellelementen, welche **Verwaltungstoolsuite**-Modellelementen im Zielmodell entsprechen. Diese Entsprechung wird im Korrespondenzmodell mit dem **System2Toolsuite**-Modellelement dargestellt. Das **System2Toolsuite**-Modellelement setzt die **KomponentenGruppe** und die **Verwaltungstoolsuite** mittels Referenzen zueinander in Beziehung. Das **KomponentenGruppe**-Modellelement wird über die *src*-Referenz mit dem **Verwaltungstoolsuite**-Modellelement über die *trg*-Referenz des **System2Toolsuite**-Korrespondenzmodellelements verbunden.

In der Praxis ist das simultane Generieren von konsistenten Modellpaaren und eines Korrespondenzmodells oftmals nicht das Ziel. Stattdessen wird das Wiederherstellen von Konsistenz angestrebt. Damit dies erreicht werden kann, müssen TGG-Regeln zuvor *operationalisiert* werden, da sie nur das simultane Ändern des Modellpaars beschreiben. Dadurch lassen sich *Vorwärts- und Rückwärts-Regeln* ableiten, welche gemeinsam bidirektionale Modelltransformationen ausdrücken. Mithilfe der TGG-Regeln und deren operationalisierten Formen lassen sich Synchronisierer bauen, welche das Synchronisierungsproblem zwischen einem Modellpaar lösen können. Dies wird in den folgenden Abschnitten beispielhaft vorgestellt. Dafür wird zuvor ein Ursprungs- und ein Zielmodell definiert und Entsprechungen zwischen den Modellelementen beispielhaft eingeführt. Als Ursprungsmodell wird im Folgenden eine Sicht verwendet, welche zum Darstellen der Struktur des Streamingdienstes eingesetzt werden kann. Hierfür eignen sich Modelle, welche durch das Metamodell in Abbildung 2.9 spezifiziert werden, und das System zerteilt in Subsysteme, als **KomponentenGruppen** dargestellt, und Systemelemente, als **Komponenten** dargestellt, repräsentiert.

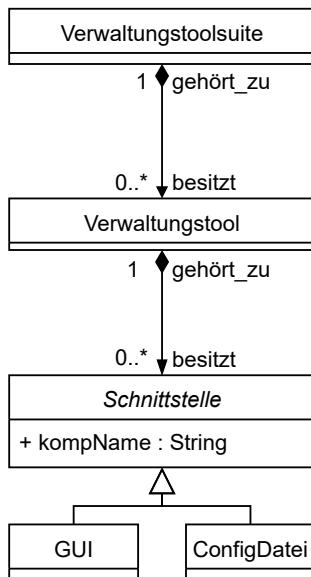


Abbildung 2.13.: Systemsicht, welche die Struktur einer **Verwaltungstoolsuite**, bestehend aus **Verwaltungstools** für die einzelnen **KomponentenGruppen** und **Schnittstellen** für die einzelnen **Komponenten**, darstellt.

Als Zielmodell wird eine Sicht verwendet, welche die Struktur einer **Verwaltungstoolsuite** darstellt. Dessen Metamodell in Abbildung 2.13 beschreibt, dass **Verwaltungstoolsuites** unbegrenzt viele **Verwaltungstools** besitzen. Diese **Verwaltungstools** besitzen unbegrenzt viele **Schnittstellen**, welche entweder eine **GUI** („Graphical User Interface“ engl. für graphische Nutzeroberfläche) oder eine **ConfigDatei** (engl. für Konfigurationsdatei) sind. Des Weiteren besitzen **Schnittstellen** das *kompName*-Attribut, welches eine Zeichenkette beinhaltet. Während der Planung der Architektur des Streamingdienstes wird nun beispielsweise entschieden, dass das System mittels einer **Verwaltungstoolsuite** konfigurierbar sein soll. Jedes Systemelement soll

durch das Editieren einer **ConfigDatei** (engl. für Konfigurationsdatei) einstellbar sein. Des Weiteren sollen Systemelemente auch mittels einer **GUI** („Graphical User Interface“ engl. für graphische Nutzeroberfläche) konfigurierbar sein, wenn die Endnutzer des Streamingdienstes direkt mit diesem Systemelement interagieren können. In anderen Worten wird eine zusätzliche **GUI** benötigt, wenn die Repräsentation des Systemelements als **Komponente** den Wert des *endNutzerKontakt*-Attribut auf *wahr* gesetzt hat. Die **GUI** und die **ConfigDatei** sollen dabei denselben Namen wie die **Komponente** besitzen und diesen im *kompName*-Attribut speichern. Ursprungs- und Zielmodell teilen sich Informationsgehalt, da sich Elemente der Sichten, wie im Folgenden beschrieben, entsprechen. Das System entspricht der **Verwaltungstoolsuite**, da entschlossen worden ist, dass das System mit der Toolsuite konfigurierbar sein soll. Existiert das System dargestellt als **KomponentenGruppe** in der ersten Sicht, aber keine **Verwaltungstoolsuite** in der zweiten, so widerspricht dies dem Entschluss, dass der Streamingdienst mit einer Toolsuite konfigurierbar sein soll. Subsysteme und **Verwaltungstools** entsprechen sich, da Subsysteme **Verwaltungstools** zum Konfigurieren vorsehen. **Komponenten** der ersten Sicht entsprechen den **Schnittstellen** der zweiten Sicht. Eine **Schnittstelle** wird im Grunde nicht benötigt, wenn sie zwar in der zweiten Sicht vorkommt und somit geplant ist, aber keiner **Komponente** in der ersten Sicht entspricht. Die Darstellung dieser Entsprechungen im Korrespondenzmodell wird im Folgenden präsentiert.

### 2.6.1. Erweiterung des laufenden Beispiels um ein Korrespondenzmodell

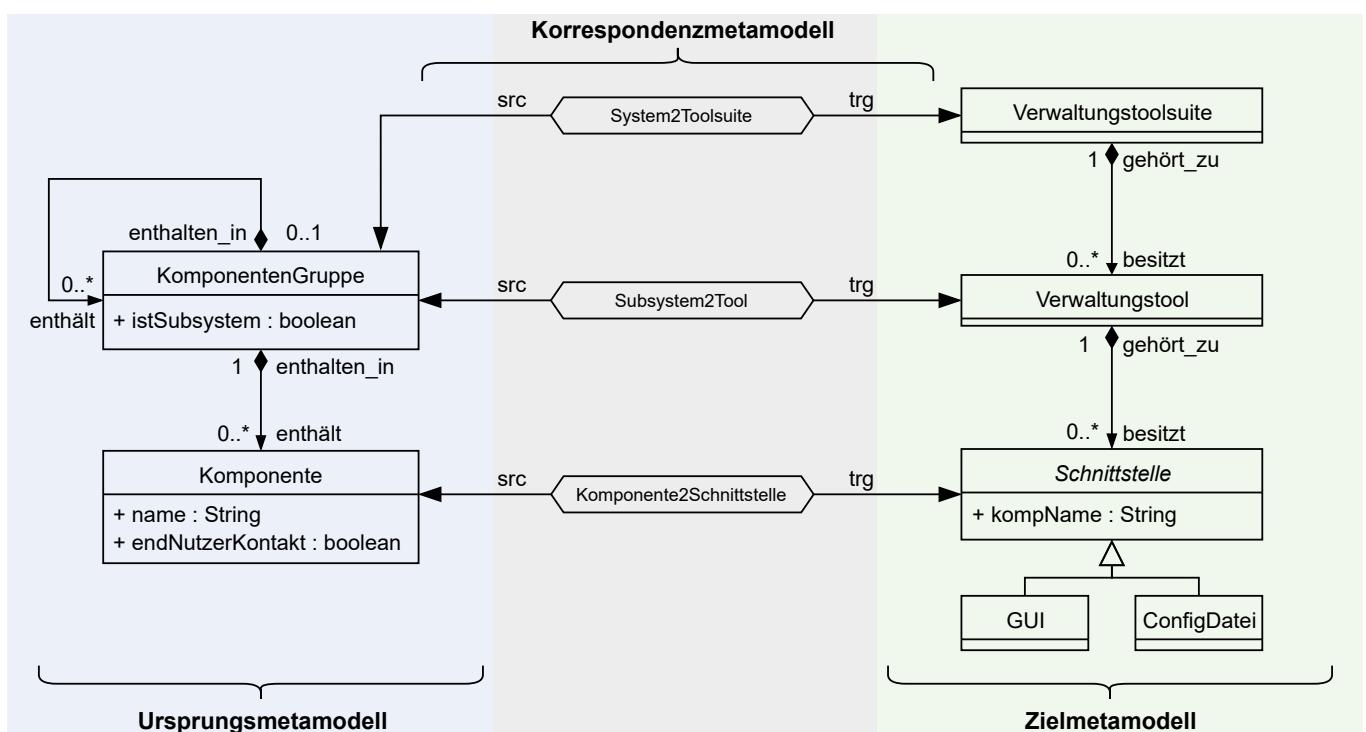


Abbildung 2.14.: Metamodelle des Ursprungs-, Korrespondenz- und Zielmodell

Wie zuvor erwähnt, referenzieren Modellelemente des Korrespondenzmodells jeweils ein Element aus dem Ursprungsmetamodell und dem Zielmetamodell und stellen die Entsprechung zwischen diesen referenzierten Elementen dar. Das Ursprungsmetamodell wird durch das Metamodell in Abbildung 2.9 spezifiziert und bildet die Struktur des Streamingdienstes, in **KomponentenGruppen** und **Komponenten** unterteilt, ab. Das Zielmetamodell stellt die **Verwaltungstoolsuite** zum Konfigurieren des Streamingdienstes, bestehend aus **Verwaltungstools** und

**Schnittstellen**, dar und wird mit dem Metamodell aus Abbildung 2.13 spezifiziert. Abbildung 2.14 visualisiert das fertige Korrespondenzmetamodell, welches gemeinsam mit den Metamodellen des Ursprungs- und Zielmodells abgebildet wird. Die Korrespondenzmodellelemente sind aus den Entsprechungen zwischen den Ursprungs- und Zielmodellelementen abzuleiten, wie nun vorgeführt wird. Das System Streamingdienst soll mit der **Verwaltungstoolsuite** konfigurierbar sein. Dementsprechend muss die Entsprechung zwischen dem System, als **KomponentenGruppe**-Modellelement dargestellt, und dem **Verwaltungstoolsuite**-Modellelement abgebildet werden. Hierfür wird die **System2Toolsuite**-Korrespondenz eingeführt. Der Bezug dieser Modellelemente wird durch das Referenzieren der **KomponentenGruppe** über die *src*-Kante („source“ engl. für Ursprung) und dem Referenzieren der **Verwaltungstoolsuite** über die *trg*-Kante („target“ engl. für Ziel) hergestellt. Da alle Subsysteme des Streamingdienstes eigene **Verwaltungstools** zum Konfigurieren haben sollen, ist die **Subsystem2Tool**-Korrespondenz notwendig. Das **Subsystem2Tool**-Modellelement setzt die **KomponentenGruppen** und die **Verwaltungstools** miteinander in Bezug, indem es die **KomponentenGruppe** über die *src*-Kante und das **Verwaltungstool** über die *trg*-Kante referenziert. Jede **Komponente** entspricht mindestens einer **Schnittstelle**. Somit wird die **Komponente2Schnittstelle**-Korrespondenz zum Darstellen dieser Beziehung definiert. **Komponente2Schnittstelle**-Modellelemente verbinden eine **Komponente** mittels der *src*-Referenz und eine der entsprechenden **Schnittstellen** mittels der *trg*-Referenz. Hat eine **Komponente** das *endNutzerKontakt*-Attribut auf *wahr* gesetzt, so benötigt es eine **GUI** und eine **ConfigDatei** als **Schnittstelle**. Somit entspricht die **Komponente** mehr als nur einer **Schnittstelle**. Besitzt ein Modellelement mehrere Entsprechungen, so wird es von mehreren Korrespondenzelementen referenziert. Beispielsweise wird die **Komponente** von zwei verschiedenen **Komponente2Schnittstelle**-Korrespondenzen referenziert und somit mit mehreren **Schnittstellen** in Bezug zueinander gesetzt. Es ist zu berücksichtigen, dass die Entsprechungen zwischen Ursprungs- und Zielmodellelementen auch abhängig von Attributen der Modellelemente sind. Unter anderem sind eine **KomponentenGruppe** und eine **Verwaltungstoolsuite** verbunden durch eine **System2Toolsuite**-Korrespondenz nur konsistent, wenn die **KomponentenGruppe** auch tatsächlich das System ist. Dementsprechend muss das *istSubsystem*-Attribut der **Komponente** *falsch* sein. Wäre dieses Attribut nämlich *wahr*, so sollte die **KomponentenGruppe** einem **Verwaltungstool** im Zielmodell entsprechen und nicht einer **Verwaltungstoolsuite**. Bedingungen an Referenzen und Attributen bei Korrespondenzen können mit TGGen mitberücksichtigt werden, wie im Folgenden demonstriert wird.

## 2.6.2. Tripel-Graph-Grammatik

Mit einer Tripel-Graph-Grammatik (TGG) wird die simultane Generierung zweier zueinander konsistenter Modelle regelbasiert spezifiziert. Die Regeln einer TGG sind deklarativ, genauer gesagt, sie beschreiben das Ergebnis einer solchen Generation, ohne eine Vorgehensweise anzugeben. TGGen sind eine Generalisierung von *kontextfreien Paar-Grammatiken* [18]. Eine Paar-Grammatik-Regel drückt eine eins-zu-eins-Korrespondenz zwischen einem Element aus dem Ursprungs-Modell und einem Element aus dem Ziel-Modell aus. Mittels dieses Ausdruckes können kontextfrei korrespondierende Elemente generiert werden. Dies bedeutet, dass bei der Generierung der Elemente keine Nebenbedingungen, wie die Existenz bestimmter anderer Elemente, berücksichtigt werden. TGGen erweitern Paar-Grammatiken um Kontextsensitivität und der Möglichkeit, auch n-zu-m-Korrespondenzen auszudrücken [1]. Diese Erweiterung wird durch das Einbeziehen eines dritten Modells, dem Korrespondenz-Modell, ermöglicht, welches die Beziehungen zwischen den Modellelementen des Ursprungs- und des Ziel-Modells abbildet. Eine TGG spezifiziert jeweils die Generierung von *konsistenten Tripeln*. Ein Tripel besteht jeweils aus einem Ursprungs-, Ziel- und Korrespondenz-Modell. Ein konsistentes Tripel ist dabei ein Tripel, dessen Beziehungen und Elemente so angeordnet sind, dass der geteilte Informationsgehalt zwischen den Ursprungs- und Ziel-Modellelementen vollständig und widerspruchsfrei ist. Die Sprache aller konsistenten Tripel wird durch eine TGG und dessen Regeln beschrieben. Die Idee hinter der Nutzung einer

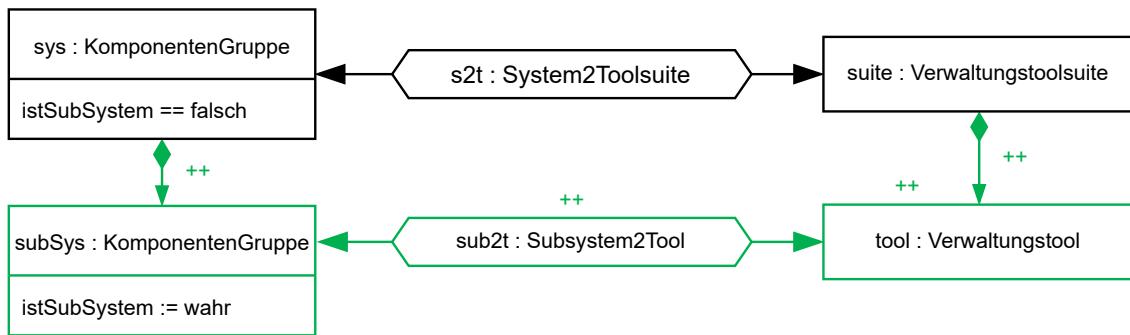
TGG zum Generieren eines Tripels ist, dass nach Anwenden der TGG-Regeln auf ein bereits konsistentes Tripel erneut ein konsistentes Tripel entsteht. Es ist somit allein durch das Anwenden von TGG-Regeln nicht möglich, das Tripel in einen inkonsistenten Zustand zu versetzen.



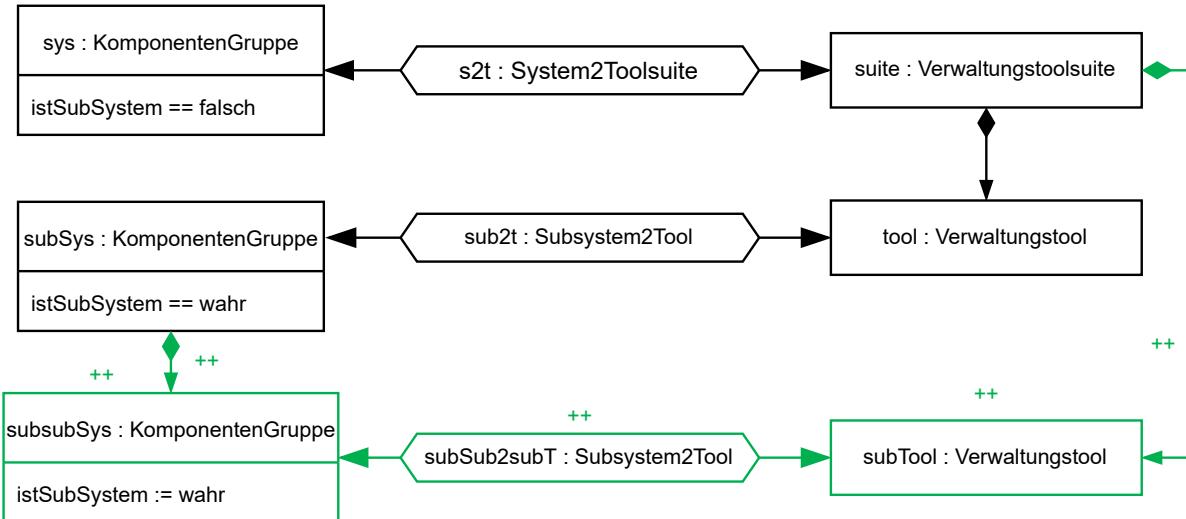
Abbildung 2.15.: TGG-Regel **R1** zum Generieren des Systems und der Verwaltungstoolsuite.

Der Aufbau und das Prinzip einer TGG wird im folgenden anhand von Beispiel-Modellen veranschaulicht, welche durch die Meta-Modelle aus Abschnitt 2.6.1 spezifiziert sind. Eine TGG-Regel deklariert das Ergebnis einer Modelltransformation, welche in jeder der drei *Modelldomänen* Modellelemente simultan erzeugen kann. Hierbei sind die Modelldomänen entweder das Ursprungs-, Ziel- oder Korrespondenzmetamodell. In Abbildung 2.15 ist die TGG-Regel **R1** abgebildet. Modellelemente der Ursprungs-Domäne befinden sich auf der linken und Modellelemente der Ziel-Domäne auf der rechten Seite als Rechtecke dargestellt. Die Korrespondenzelemente werden zwischen den korrespondierenden Ursprungs- und Ziel-Elementen in der Mitte mit einem Sechseck statt eines Rechtecks zur einfachen Unterscheidung visualisiert. Neu zu erzeugende Elemente werden analog zu den Graphtransformationen aus Abschnitt 2.5 in Grün und mit einem **++** dargestellt. Diese Regel erzeugt im Ursprungsmodell die **KomponentenGruppe** *sys*, dessen *istSubsystem*-Attribut *falsch* ist. Dementsprechend handelt es sich bei *sys* um das System. Analog wird im Zielmodell die korrespondierende **Verwaltungstoolsuite** *suite* erzeugt. Damit diese Korrespondenz auch im Tripel repräsentiert wird, wird das **System2Toolsuite**-Korrespondenzelement *s2t*, welches *sys* und *suite* referenziert, im Korrespondenzmodell erzeugt. Die TGG-Regel **R1** ist kontextfrei, da alle Modellelemente in der Regel neu erzeugt werden. Da kein Kontext vor der Regelanwendung geprüft werden muss, ist **R1 axiomatisch** und kann somit unbegrenzt oft ausgeführt werden. TGG-Regeln sind kontextsensitiv, weshalb zu Beginn der Generierung des konsistenten Tripels eine axiomatische Regel ausgeführt werden muss. Für das laufende Beispiel ist es sinnvoll, **R1** zu Beginn auszuführen, da die **KomponentenGruppe**, welche das System repräsentiert, und dessen entsprechende **Verwaltungstoolsuite** transitiv alle anderen Modellelemente des jeweils Ursprungs- oder Zielmodells enthalten. Mittels TGGen lassen sich auch Attributbedingungen spezifizieren, welche angeben, welche Attributwerte erzeugte Modellelemente annehmen, oder welche Werte existierende Kontext-Modellelemente vorweisen müssen [19]. Dies ermöglicht das Setzen des *istSubsystem*-Attributs der **KomponentenGruppe** *sys* auf *falsch*, womit im Ursprungsmodell ausgedrückt wird, dass *sys* das System repräsentiert und kein Subsystem.

Die TGG-Regel **R2** in Abbildung 2.16a deklariert das Erzeugen der **KomponentenGruppe** *subSys* und das korrespondierende **Verwaltungstool** *tool*. Das zu erzeugende **Subsystem2Tool**-Korrespondenzmodellelement *sub2t* setzt *subSys* und *tool* mittels Referenzen zueinander in Bezug. Da *subSys* ein Subsystem darstellen soll, wird das *istSubsystem*-Attribut auf *wahr* gesetzt. Subsysteme sind im System enthalten. Ebenfalls sind **Verwaltungstools** in einer **Verwaltungstoolsuite** enthalten. Dementsprechend müssen das System und die entsprechende **Verwaltungstoolsuite** Teil des Kontexts sein, damit Subsysteme und **Verwaltungstools** erzeugt werden können. Daraus folgt, dass die **KomponentenGruppe** *sys*, die **Verwaltungstoolsuite** *suite* und das **System2Toolsuite**-Korrespondenzmodellelement *s2t*, welches *sys* und *suite* referenziert, Teil des Kontexts sind. *sys* soll das System repräsentieren und somit wird überprüft, dass das *istSubsystem*-Attribut *falsch* ist. Wie in Abschnitt 2.5 bei den Graphtransformationsregeln werden Kontextmodellelemente in Schwarz dargestellt. Damit die Kompositionenbeziehungen zwischen Systemen und Subsystemen, und zwischen **Verwaltungstoolsuites** und **Verwaltungstools** im Tripel abgebildet wird, werden diese Referenzen ebenfalls angelegt. Mit der Regel



(a) TGG-Regel **R2** zum Generieren von Subsystemen des Systems und deren **Verwaltungstools**.

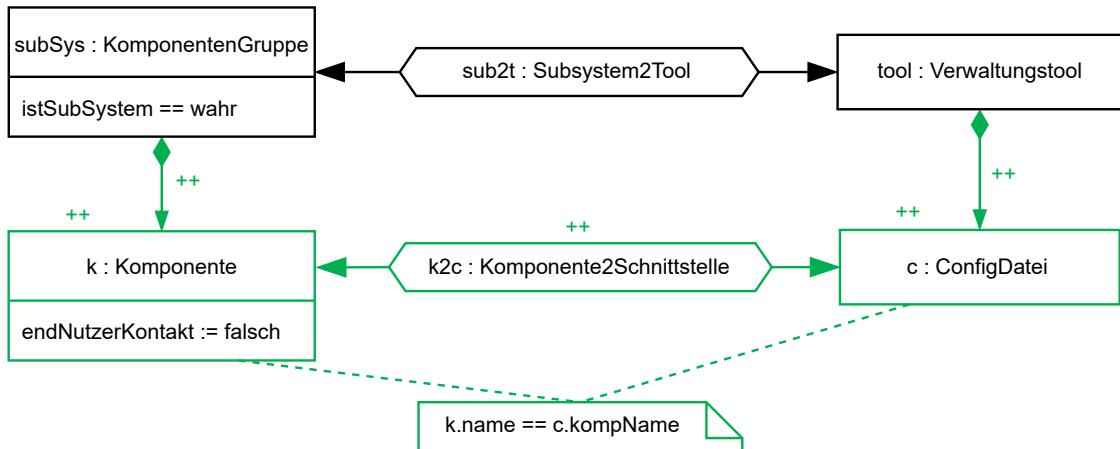


(b) TGG-Regel **R3** zum Generieren von Subsystemen der Subsysteme mit entsprechenden **Verwaltungstools**.

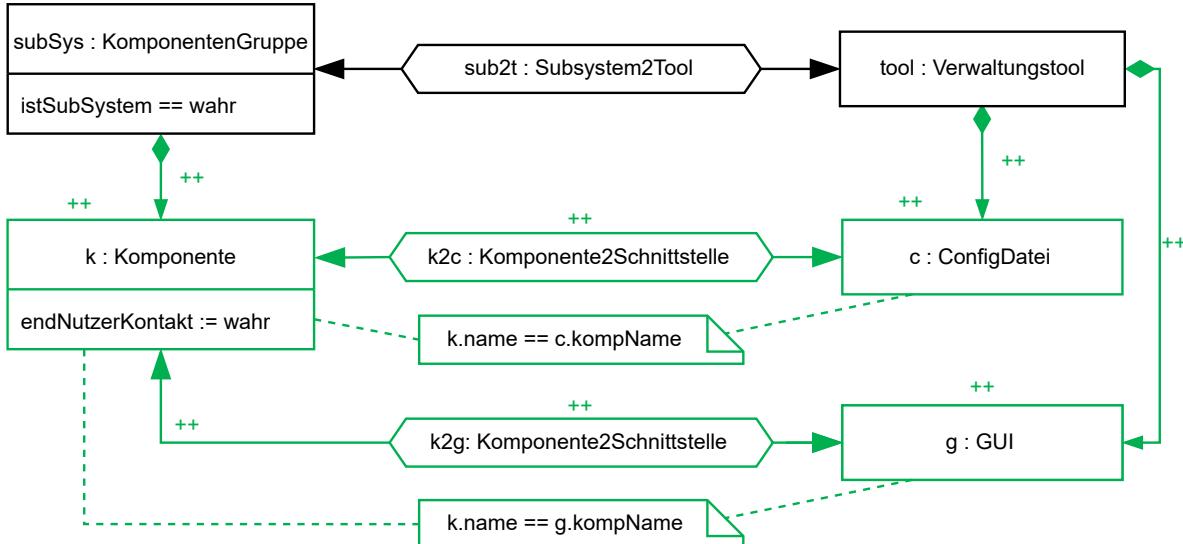
Abbildung 2.16.: TGG-Regeln **R2** und **R3** zum Generieren von Subsystemen und deren **Verwaltungstools**.

**R2** ist es möglich, die Subsysteme von Systemen zu generieren. Subsysteme können allerdings selbst wieder Subsysteme enthalten. Dementsprechend wird die TGG-Regel **R3** in Abbildung 2.16b benötigt, welche das Generieren eines Subsystems, als **KomponentenGruppe** dargestellt, in einem anderen Subsystem, ebenfalls als **KomponentenGruppe** dargestellt, ermöglicht. Wie in **R2** sind *sys*, *s2t* und *suite* Teil des Kontexts. Des Weiteren wird im Kontext vorausgesetzt, dass eine **KomponentenGruppe** *subSys*, dessen *istSubsystem*-Attribut *wahr* ist, mit einem entsprechenden **Verwaltungstool** *tool* in *suite* existiert. Die Entsprechung zwischen *subSys* und *tool* wird im Kontext mit dem **Subsystem2Tool**-Korrespondenzmodellelement *sub2t* vorausgesetzt, welches *suite* und *tool* referenziert. In dieser Regel werden die **KomponentenGruppe** *subsubSys*, das **Verwaltungstool** *subTool* und das beide in Beziehung zueinander setzende **Subsystem2Tool**-Korrespondenzmodellelement *subSub2subT* neu erzeugt. Das *istSubsystem*-Attribut von *subSys* wird dabei auf *wahr* gesetzt, um darzustellen, dass es ein Subsystem ist. *subsubSys* wird in *subSys* und *subTool* wird in *suite* angelegt. Dadurch, dass diese Regel nicht beschreibt, in welchem Modellelement *subSys* enthalten ist, lässt sich diese Regel rekursiv anwenden und es können Subsysteme in jeglichen Subsystemen im Modell erzeugt werden.

Nach der Erzeugung von **KomponentenGruppe**- und **Verwaltungstool**-Modellelementen können mit den Regeln **R4** und **R5**, abgebildet in Abbildung 2.17, **Komponenten** und korrespondierende **Schnittstellen** generiert werden. Der Unterschied zwischen **R4** und **R5** ist, dass **R4** das Generieren von **Komponenten**



(a) TGG-Regel **R4**. Generation von **Komponenten**, welche nur eine **ConfigDatei** benötigen.



(b) TGG-Regel **R5**. Generation von **Komponenten**, welche eine **ConfigDatei** und eine **GUI** benötigen.

Abbildung 2.17.: TGG-Regeln zum Generieren von **Komponenten** und deren korrespondierenden **Schnittstellen**.

deklariert, welche nur einer **ConfigDatei** als **Schnittstelle** entsprechen. Im Gegensatz zu **R4** deklariert **R5** das Generieren von **Komponenten**, welche einer **ConfigDatei** und einer **GUI** entsprechen. Demnach unterscheiden sich die Regeln hauptsächlich in der Anzahl der zu erzeugenden Modellelementen. Beide Regeln setzen die Existenz einer **KomponentenGruppe** *subSys*, dessen *istSubSystem*-Attribut *wahr* ist, und das entsprechende **Verwaltungstool** *tool* im Kontext voraus. Auch das Korrespondenzelement *sub2t*, welches *subSys* und *tool* referenziert, ist in den Regeln **R4** und **R5** Teil des Kontexts. Eine **Komponente** *k*, eine **ConfigDatei** *c* und ein **Komponente2Schnittstelle**-Korrespondenzelement *k2c*, welches *k* und *c* referenziert, werden in beiden Regeln neu erzeugt, da jede **Komponente** einer **ConfigDatei** entsprechen soll. Dabei wird *k* in *subSys* und *c* in *tool* erzeugt. Auch wird mittels einer Attributbedingung sichergestellt, dass das *name*-Attribut von *k* identisch zum *kompName*-Attribut von *c* ist. In **R4** wird des Weiteren auch das *endNutzerKontakt*-Attribut der **Komponente** *k* auf *falsch* gesetzt. Im Gegensatz dazu setzt **R5** dieses Attribut auf *wahr*. Es wird auch die zusätzliche **Schnittstelle** *g* in *c* als eine **GUI** und das weitere **Komponente2Schnittstelle**-Korrespondenzelement *k2g* erzeugt.

GUI  $g$  und Komponente  $k$  werden von  $k2g$  referenziert, um die Entsprechung der beiden Modellelemente im Korrespondenzmodell darzustellen. Eine weitere Attributbedingung stellt in R5 sicher, dass das *name*-Attribut von  $k$  ebenfalls identisch zum *kompName*-Attribut von  $g$  ist.

Mit den Regeln R1 bis R5 ist es möglich ein konsistentes Tripel bestehend aus den in Abschnitt 2.6.1 beschriebenen Modellen simultan zu generieren. In der Praxis werden allerdings oft Modelltransformationsregeln zum Ineinander-Übersetzen oder Synchronisieren von Modellen benötigt. Generell lassen sich solche Modelltransformationsregeln anhand ihrer Transformationsrichtung in *unidirektionale* und *bidirektionale* Regeln klassifizieren. Unidirektionale Transformationen können nur in eine Richtung angewendet werden und entsprechen einer Übersetzung der zu transformierenden Modellelemente eines Modells in das jeweils andere. Beispielsweise könnte mit einer unidirektionalen Modelltransformation das Zielmodell ausgehend vom UrsprungsmodeLL abgeleitet werden, was einer Übersetzung des UrsprungsmodeLLs in das Zielmodell entspricht. Mit dieser Transformation ist es allerdings nicht möglich, Modellelemente des Zielmodells in das UrsprungsmodeLL zu übersetzen. Damit dies ebenfalls möglich ist, müssen bidirektionale Transformationen angewendet werden, welche das Übersetzen in beide Richtungen erlauben. Bidirektionale Transformationen lassen sich mit einem komplementären Paar an unidirektionalen Transformationen implementieren [20]. Eine Regel dieses Paares transformiert das Zielmodell ausgehend vom UrsprungsmodeLL und wird *Vorwärts-Regel* genannt. Die zur Vorwärts-Regel komplementäre *Rückwärts-Regel* transformiert das UrsprungsmodeLL ausgehend vom Zielmodell. Solche Paare an Vorwärts- und Rückwärts-Regeln lassen sich automatisiert aus TGG-Regeln ableiten [21], indem diese *operationalisiert* werden. Das Operationalisieren einer TGG funktioniert zum Ableiten der Vorwärts- oder Rückwärts-Regel in beiden Fällen identisch. Dementsprechend wird dies im folgenden anhand der Regel R2 aus Abbildung 2.16a für das Ableiten der Vorwärts-Regel demonstriert.

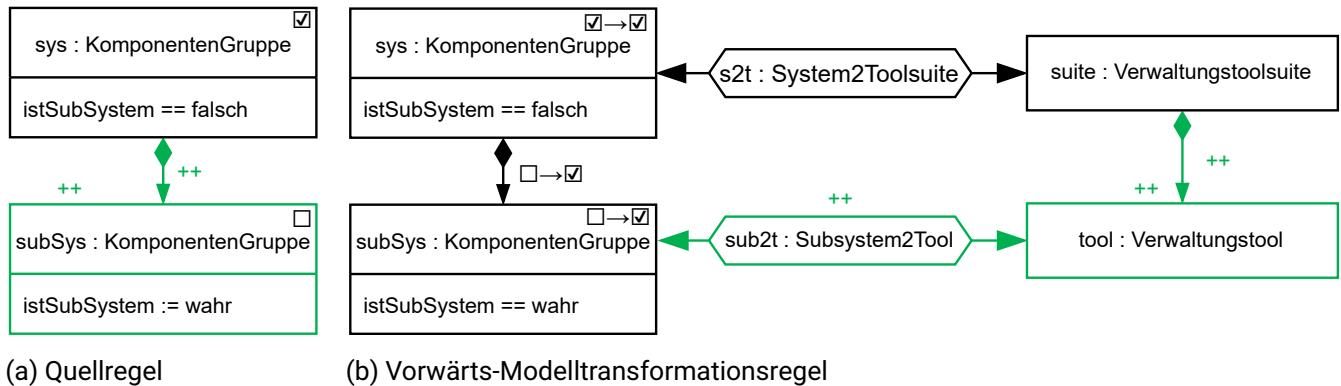


Abbildung 2.18.: Operationale Regel der TGG-Regel R2 aus Abbildung 2.16a.

Eine TGG-Regel wird für die Vorwärts-Richtung operationalisiert, indem sie in eine *Quellregel* und eine Vorwärtsregel aufgespalten wird. Das Paar bestehend aus Quell- und Vorwärts-Regel wird auch *operationale Regel* genannt. In Abbildung 2.18 wird die operationale Regel von R2 beispielhaft dargestellt. Die Quellregel beschreibt eine Änderung im UrsprungsmodeLL als eine Graphtransformationenregel. Diese kann von der TGG-Regel abgeleitet werden, indem alle deklarierten Elemente ausgelassen werden, welche nicht Teil des UrsprungsmodeLLs sind. Im Falle von R2 bleiben somit lediglich das Kontext-KomponentenGruppe-ModeLlement sys und das zu erzeugende KomponentenGruppe-ModeLlement subSys übrig. Die resultierende Graphtransformationenregel beschreibt in diesem Fall das Erzeugen von subSys und das Anhängen von subSys als Inhalt von sys und ist in Abbildung 2.18a abgebildet. Die Vorwärts-Regel deklariert, wie die durch die Anwendung der Quellregel ausgelöste Änderung im UrsprungsmodeLL in das Korrespondenz- und ZielmodeLL übersetzt wird. Diese wird von der TGG-Regel abgeleitet, indem alle ModeLlemente, welche in der

Quellregel vorkommen, als Kontext interpretiert werden. Dementsprechend ergibt sich die in Abbildung 2.18 dargestellte Modelltransformationsregel. Sie unterscheidet sich von **R2** aus Abbildung 2.16a nur darin, dass nun *subSys* und die Kompositionssreferenz von *sys* auf *subSys* Kontext sind. In der TGG-Regel **R2** wurde zuvor das *istSubSystem*-Attribut von *subSys* auf *wahr* gesetzt. Dementsprechend wird in der Vorwärts-Regel mittels einer Attributbedingung vorausgesetzt, dass das *istSubSystem*-Attribut von *subSys* *wahr* ist, anstatt es auf *wahr* zu setzen. Das Anwenden von der Quellregel und der Vorwärtsregel resultiert in den gleichen Änderungen am Tripel wie die Anwendung der ursprünglichen TGG-Regel. Allerdings könnten die durch das Anwenden der Quellregel ausgelösten Änderungen mehrfach übersetzt werden. Um dies zu verhindern, werden Übersetzungsmarkierungen analog zu denen von Lauder et al. [22] eingeführt. Modellelemente, welche noch nicht übersetzt worden sind, werden mit dem folgenden leeren Quadrat markiert:  $\square$ . Bereits übersetzte Modellelemente erhalten das folgende Quadrat mit Haken als Markierung:  $\checkmark$ . Elemente, welche von der Quellregel neu erzeugt werden, sind vorerst unübersetzt. Nach der Anwendung der Vorwärtsregel sind diese Elemente übersetzt und ihre Markierung wird von  $\square$  zu  $\checkmark$  geändert. Die von der Quellregel neu erzeugten Elemente werden in der Vorwärtsregel mit  $\square \rightarrow \checkmark$  gekennzeichnet, um das Setzen der Markierung zu symbolisieren. Der Kontext der TGG-Regel sollte bereits übersetzt worden und dementsprechend markiert sein. Das Anwenden der Vorwärtsregel ändert ihre Markierungen nicht, was mit  $\checkmark \rightarrow \checkmark$  gekennzeichnet wird. Durch das Verwenden dieser Übersetzungsmarkierungen wird erreicht, dass durch das Anwenden von Quell- und Vorwärts-Regeln nur Änderungen am Tripel vorgenommen werden, welche auch durch das Anwenden von TGG-Regeln erreicht werden können. Unter anderem wird so verhindert, dass Vorwärts-Regeln mehrfach ausgeführt werden können. Erzeugt ein Nutzer nun Modellelemente im Ursprungsmodell, welche durch Quellregeln beschrieben werden können, so sind diese Änderungen vorerst unübersetzt und das Tripel ist gegebenenfalls in einem inkonsistenten Zustand. Neu erzeugte Elemente müssen übersetzt werden und somit Entsprechungen in den anderen Modellen besitzen. Dafür können die Vorwärts-Regeln angewendet werden, welche die unübersetzten Elemente übersetzen und gegebenenfalls die Konsistenz des Tripels wiederherstellen, ohne dass der Nutzer dies manuell vornehmen muss. Es kann allerdings sinnvoll sein, Änderungen am Tripel, ausgelöst durch das Anwenden einer Vorwärts-Regel, wieder rückgängig zu machen. Dies ist beispielsweise dann der Fall, wenn ein Nutzer weitere Änderungen vornimmt, welche in Konflikt zur vorherigen Übersetzung stehen. Die Rückgängigmachung einer Vorwärts-Regel-Anwendung lässt sich durch das Anwenden der *invertierten* Vorwärts-Regel bewerkstelligen [22].

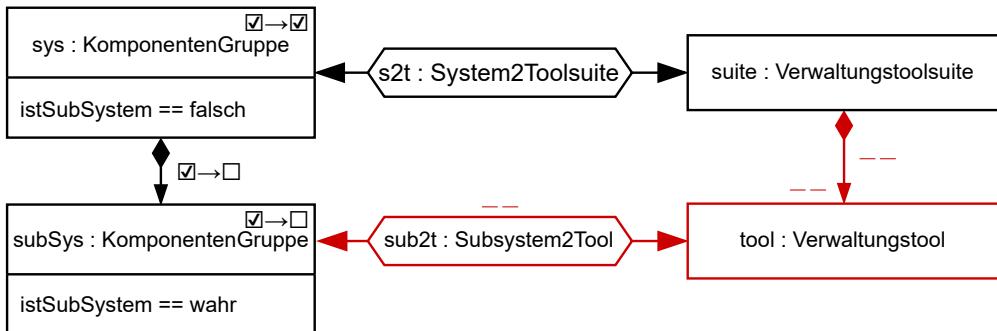


Abbildung 2.19.: Invertierte Vorwärts-Modelltransformationsregel der Vorwärts-Regel aus Abbildung 2.18b.

Die invertierte Vorwärtsregel beschreibt das Abbauen von Strukturen, welche von der regulären Vorwärts-Regel aufgebaut worden sind. Abbildung 2.19 zeigt die invertierte Vorwärts-Regel der Regel aus Abbildung 2.18b. Elemente, welche durch das Anwenden der Vorwärts-Regel erzeugt worden sind, werden gelöscht und sind rot, mit einem  $--$  versehen, dargestellt. Jene Modellelemente, deren Entsprechungen durch das Anwenden der invertierten Vorwärtsregel gelöscht werden, waren zuvor übersetzt und mit einem  $\checkmark$  markiert. Nun sind

sie unübersetzt und werden mit einem  $\square$  markiert. Dementsprechend werden diese Modellelemente in der invertierten Vorwärts-Regel mit einem  $\checkmark \rightarrow \square$  gekennzeichnet, um den Übergang von übersetzt zu nicht übersetzt darzustellen. Uniübersetzte Modellelemente sind nun auch als unübersetzt markiert und eine andere Vorwärts-Regel kann gewählt und angewendet werden, um die Modellelemente neu zu übersetzen.

### 2.6.3. Inkrementelle Modellsynchronisierung

Mit einer Modellsynchronisierung wird die Konsistenz zwischen einem Ursprungs- und Zielmodell, welche sich im Informationsgehalt überlappen, wiederhergestellt. Konsistenz kann verloren gehen, indem am geteilten Informationsgehalt Änderungen im Ursprungs- und/oder Zielmodell vorgenommen werden. Die Synchronisierung eines Modellpaars ist mit *Batch-Übersetzungen* implementierbar. Eine Batch-Übersetzung ist das Ableiten eines noch nicht existierenden Modells von einem anderen. Dies ist mit den operationalisierten TGG-Regeln umsetzbar, indem durch Anwenden der Vorwärts-Regeln das Ziel- vom Ursprungsmodell abgeleitet wird. In dieser Art und Weise kann ein Modell vom jeweils anderem neu abgeleitet werden, um ein konsistentes Modellpaar zu erhalten. Das Verwenden von Batch-Übersetzungen ist allerdings in der Regel zur Modellsynchronisierung nicht zu bevorzugen. Ist beispielsweise das Zielmodell nicht konsistent zum Ursprungsmodell, so wird das Zielmodell bei der Modellsynchronisierung durch Batch-Übersetzung vollständig verworfen und neu abgeleitet. Informationen, welche keine Entsprechungen im Ursprungsmodell besitzen, gehen dabei allerdings verloren, da sie nicht aus dem Ursprungsmodell abgeleitet werden können. Des Weiteren wird die Batch-Übersetzung mit zunehmender Modellgröße teurer. Deswegen ist in der Regel eine Modellsynchronisierung, welche nur einzelne Änderungen propagiert, zu bevorzugen. Der Prozess einer solchen Modellsynchronisierung ist grob in zwei Schritte einteilbar. Im ersten Schritt werden inkonsistente Strukturen durch das Anwenden von invertierten operationalen Regeln abgebaut. Im zweiten Schritt werden unübersetzte Strukturen übersetzt. Eine solche Modellsynchronisierung wird *inkrementell* genannt, wenn sich dabei nur mit den inkonsistenten Modellteilen auseinandergesetzt wird.

Anhand von Abbildung 2.20 wird ein beispielhafter Synchronisierungsprozess vorgeführt. Kantenbezeichnungen, Instanz- und Typ-Namen der Korrespondenzmodellelemente werden nicht dargestellt. Initial sind Ursprungs- und Zielmodell zueinander konsistent, wie in Abbildung 2.20a abgebildet wird. Das Ursprungsmodell besteht aus einer **KomponentenGruppe** *sys*, dessen *istSubsystem*-Attribut auf *falsch* gesetzt ist. In *sys* ist die **KomponentenGruppe** *subSys* enthalten, welche wiederum die **Komponente** *k* enthält. Das *istSubsystem*-Attribut von *subSys* ist *wahr*. Der *Name* von *k* ist „Video Decoder“ und das *endNutzerKontakt*-Attribut ist *wahr*. Im Zielmodell befinden sich die korrespondierende **Verwaltungstoolsuite** *suite*, das **Verwaltungstool** *tool*, die **GUI** *gui* und die **ConfigDatei** *config*. Alle Strukturen des Ursprungsmodells sind somit korrekt im Zielmodell dargestellt und sich entsprechende Elemente werden durch Korrespondenzmodellelemente miteinander verbunden. Wie in Abbildung 2.20b abgebildet ist, wird nun das **endNutzerKontakt**-Attribut der **Komponente** *k* von *wahr* zu *falsch* geändert. Durch diese Änderungen sind das Ursprungs- und Zielmodell nicht mehr konsistent zueinander. Dies ist dem geschuldet, dass im laufenden Beispiel festgelegt worden ist, dass **Komponenten**, deren *endNutzerKontakt*-Attribut *falsch* ist, nur einer **ConfigDatei** anstatt einer **ConfigDatei** und einer **GUI** als **Schnittstelle** entsprechen sollen. Um die Konsistenz zwischen Ursprungs- und Zielmodell wiederherzustellen, werden die Modelle nun synchronisiert. Im ersten Schritt werden mithilfe von invertierten operationalen Regeln inkonsistente Modellstrukturen abgebaut. In diesem Fall wird die invertierte operationale Regel der TGG-Regel R5 aus Abbildung 2.17b angewendet, wodurch *config* und *gui* im Zielmodell gelöscht werden. Die Korrespondenzmodellelemente, welche zuvor die Entsprechung zwischen *gui*, *config* und *k* dargestellt haben, werden ebenfalls gelöscht. Die **Komponente** *k* besitzt nun keine Entsprechungen im Zielmodell und ist unübersetzt. Dementsprechend wird dessen Übersetzungsmarkierung von  $\checkmark$  zu  $\square$  geändert. Nun kann *k* durch das Anwenden einer Vorwärtsregel neu übersetzt werden. Im zweiten Schritt werden nun

unübersetzte Modellelemente übersetzt. In diesem Fall ist lediglich *k* als unübersetzt markiert und es wird R4 aus Abbildung 2.17a angewendet. Dadurch wird die **ConfigDatei** *config* als Entsprechung im Zielmodell erzeugt. Ebenfalls wird ein Korrespondenzelement angelegt, welches *k* und *config* in Beziehung zueinander setzt. Die **Komponente** *k* ist nun übersetzt und wird dementsprechend als übersetzt markiert. Ursprungs- und Zielmodell sind wieder konsistent zueinander und die Modellsynchronisierung ist nun abgeschlossen. Der Modellzustand nach der Synchronisierung ist in Abbildung 2.20d abgebildet.

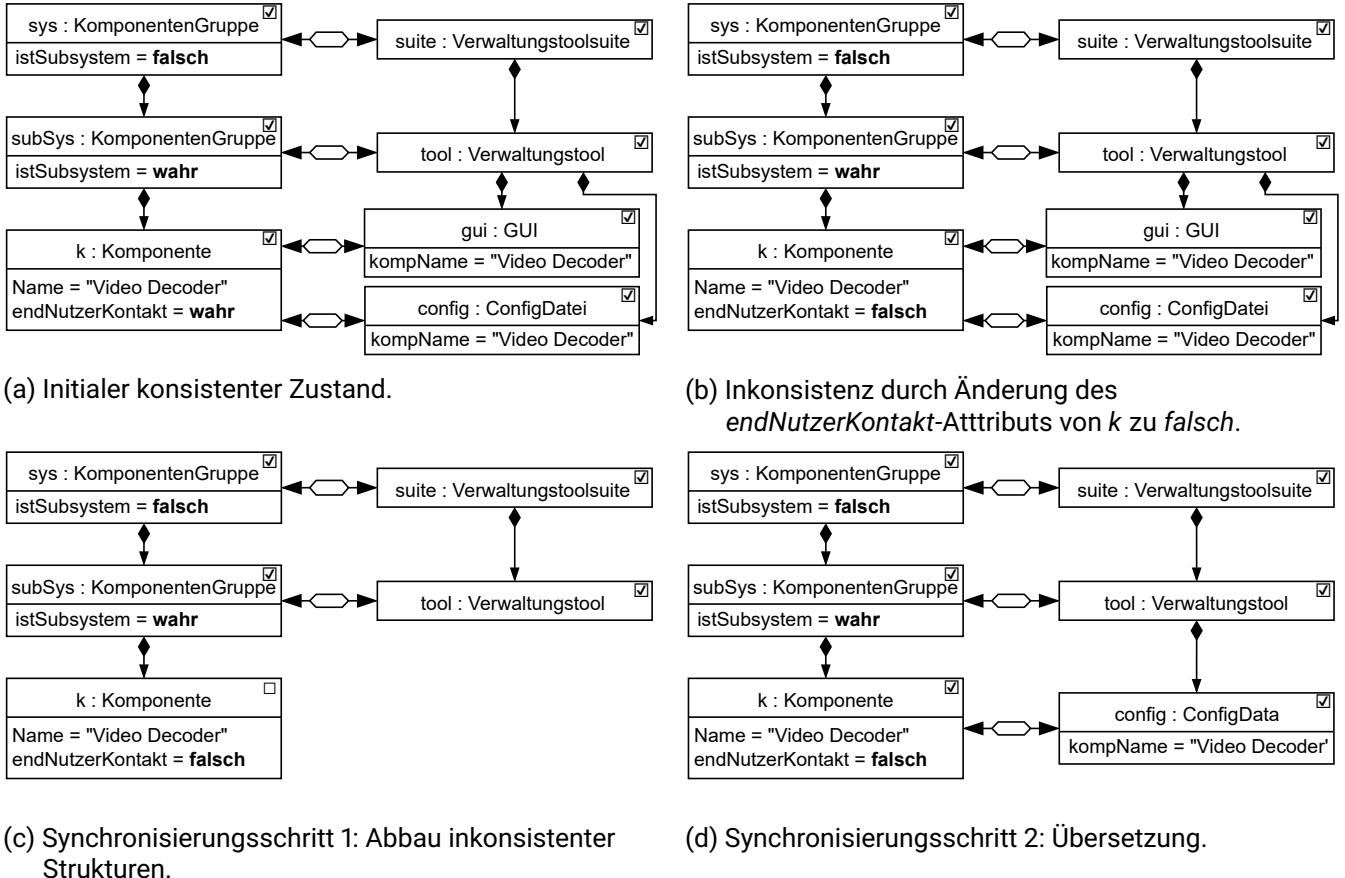


Abbildung 2.20.: Beispielhafter Synchronisierungsprozess.

Die Algorithmen zum Steuern der Modellsynchronisierung können zwischen „Bottom-Up“ rekursiv oder „Top-Down“ iterativ unterschieden werden [23]. Ein „Bottom-Up“ rekursiver Algorithmus wählt in der Regel ein zufälliges Modellelement und startet ausgehend von diesem den Synchronisierungsprozess *Kontext-getrieben*. Dabei wird überprüft, ob alle Kontextelemente von Regeln, welche dieses zufällig gewählte Element betreffen, bereits korrekt übersetzt worden sind. Beispielsweise wird eine **KomponentenGruppe** *subSys* gewählt, dessen *istSubsystem*-Attribut *wahr* ist. Dementsprechend wird geprüft, ob *subSys* in einem System, also einer **KomponentenGruppe** mit *istSubsystem*-Wert *falsch*, enthalten ist. Das Vorhanden sein der zum System korrespondierenden **Verwaltungstoolsuite** wird ebenfalls geprüft (vergleiche Abbildung 2.16a). Sind alle Kontextelemente vorhanden und übersetzt, so wird das gewählte Element selbst übersetzt. Für alle Kontextelemente, die nicht übersetzt worden sind, wird der soeben beschriebene Algorithmus *rekursiv* angewendet bis das initial gewählte Element übersetzt werden kann. „Bottom-Up“ beschreibt diese rekursive Vorgehensweise. Es wird sozusagen *unten* („bottom“ auf Engl.) bei dem gewählten Knoten im Modellgraphen begonnen und *hoch* („up“ auf Engl.) in Richtung des Wurzelknotens, welcher alle Modellelemente translativ beinhaltet, beim

Überprüfen und Übersetzen vorgearbeitet. Die Entscheidungen, welche Elemente als Nächstes überprüft und gegebenenfalls übersetzt werden, werden zur Laufzeit *lokal* an der Stelle getroffen, an welcher sich der Algorithmus gerade im Modell befindet. Durch diese lokale Betrachtung können „Bottom-Up“ rekursive Ansätze Probleme mit der Vollständigkeit der Synchronisierung haben [23]. Es ist in anderen Worten schwierig zu garantieren, dass Tripel nach der Synchronisierung bei Verwendung eines „Bottom-Up“ rekursiven Ansatzes auch konsistent sind. „Top-Down“-Ansätze unterscheiden sich von „Bottom-Up“-Ansätzen darin, dass nicht ausschließlich lokale Entscheidungen getroffen werden, sondern Informationen über das Modell gesammelt und mit diesen die Modelle *iterativ* synchronisiert werden. Die Informationen können unter anderem durch eine topologische Sortierung der Modelle erhalten werden. Dies kann Einblick über die Abhängigkeiten zwischen Modellen und Modellelementen *global*, also das gesamte Modellpaar betreffend, aufdecken. Mit diesem gesammelten Wissen wird versucht, eine möglichst effiziente Übersetzungsreihenfolge zu bestimmen, welche jedoch nicht immer existiert. Kann keine effiziente Reihenfolge gefunden werden, so müssen alle Elemente überprüft und gegebenenfalls übersetzt werden. Präzedenz-getriebene „Top-Down“ Algorithmen führen eine Präzedenzanalyse durch, mit welcher die Übersetzungsreihenfolge, in diesem Kontext *Präzedenz* genannt, bestimmt wird. Der von Leblebici et al. [24] vorgestellte „Top-Down“ inkrementelle Präzedenz-getriebene Ansatz verwendet inkrementelle Graphmustersucher, um Regelanwendungsstellen in Modellen zu finden. Dabei wird das effiziente Reagieren auf Modelländerungen *Delta-basiert* ermöglicht, indem Indexstrukturen aufgebaut werden. Mit diesen Strukturen lässt es sich vermeiden, das gesamte Modell nach jeder Änderung neu explorieren zu müssen. Auch Inkonsistenzen lassen sich auf diese Art und Weise erkennen, wenn eine vormals gültige TGG-Regelanwendung nicht mehr intakt ist. Diese Inkonsistenzen lassen sich werkzeuggestützt beheben, wie Fritzsche et al. [2] gezeigt und im State of the Art Graph-Transformation-Werkzeug eMoflon::IBeX<sup>4</sup> implementiert haben.

## 2.7. Die ARCADIA MBSE-Methode

Architecture Analysis and Design Integrated Approach (ARCADIA) [25] ist eine vom Automobilkonzern Thales zwischen 2005 und 2010 entwickelte MBSE-Methode, welche seit 2018 in der experimentellen Norm XP Z67-140 [25] festgehalten ist. Das Ziel der ARCADIA-Methode ist das modellbasierte Erstellen und Validieren von System-Architekturen zum Entwickeln komplexer Systeme. Dementsprechend teilt ARCADIA einige gemeinsame Aspekte mit MDA, vor allem da ARCADIAS ML direkt durch Systems Modeling Language (SysML) inspiriert worden ist<sup>5</sup>. Generell besteht die ARCADIA-Methode aus fünf verschiedenen Schritten. In jedem dieser Schritte wird das System ausgehend von jeweils anderen Standpunkten betrachtet und für diese mehrere Systemsichten in Form von Diagrammen erstellt. Die Summe der Systemsichten eines Schrittes ergeben zusammen ein Modell des gesamten Systems. Dabei ist zu beachten, dass jeder nachfolgende Schritt die System-Architektur spezifischer und somit weniger abstrakt darstellt als das Modell des vorherigen Schrittes. Dies bedeutet im Zusammenhang von ARCADIA, dass Elemente aus dem Modell eines Schrittes auch in dem Modell des nachfolgenden Schrittes repräsentiert und gegebenenfalls mit mehr Informationen angereichert werden müssen. Solche Modellelemente, welche andere Elemente des vorherigen Schrittes repräsentieren oder mit Informationen anreichern, werden *Realisierungen* genannt. Die Beziehung zwischen einem realisierten Modellelement und einer Realisierung wird als *Realisierungsbeziehung* bezeichnet und werden in Capella mit modelliert. Jedes Modellelement soll von mindestens einem Element im Modell des nachfolgenden Schrittes

<sup>4</sup><https://emoflon.org/>

<sup>5</sup>Eine genaue Gegenüberstellung von ARCADIA und SysML ist auf der Capella Webseite gegeben:  
[https://www.eclipse.org/capella/arcadia\\_capella\\_sysml\\_tool.html](https://www.eclipse.org/capella/arcadia_capella_sysml_tool.html)

realisiert werden. Die Abbildung 2.21 zeigt eine visuelle Zusammenfassung der ARCADIA-Methode, welche von Navas [26] übernommen worden ist. Der fünfte Schritt ist nicht abgebildet.

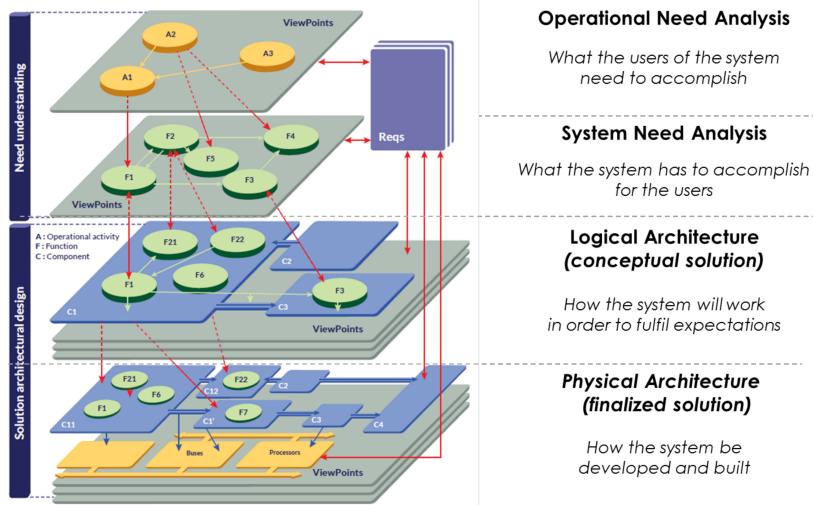


Abbildung 2.21.: Visuelle Zusammenfassung der ARCADIA-Methode von Navas [26].

Der erste Schritt, die *Operational Analysis (OA)*, befasst sich mit den Bedürfnissen der Interessengruppen. Die Beantwortung der Frage „Was möchte der Nutzer mit dem System erreichen?“ steht hier im Fokus und es wird besonderes Augenmerk auf die Ziele der Interessengruppen gelegt. Es werden die verschiedenen mit dem System interagierenden Personengruppen identifiziert und deren Interaktionen miteinander und mit dem System modelliert.

Der zweite Schritt, die *System Need Analysis (SA)*, manchmal auch *Context Architecture (CTX)* genannt, befasst sich mit den Anforderungen an das System. Das System muss Funktionen anbieten, damit die Nutzer ihre Ziele, welche zuvor in der OA modelliert wurden, tatsächlich erreichen können. Diese Funktionen werden in diesem Schritt modelliert. Die Leitfrage der SA lautet „Was muss das System für seine Nutzer erreichen?“. Die OA und SA sind als eine Bedürfnisanalyse zu verstehen. Die OA analysiert und modelliert die Bedürfnisse der Nutzer, während die SA die Bedürfnisse des Systems als Entität untersucht. Die modellierten Bedürfnisse werden als Anforderungen an das System zusammengestellt. Die letzten drei Schritte der ARCADIA-Methode beschäftigen sich damit, wie die in der OA und SA modellierten Bedürfnisse umgesetzt werden können und es werden Modelle einer tatsächlichen, umsetzbaren System-Architektur erstellt.

Schritt drei, die *Logical Architecture (LA)*, beschäftigt sich mit der Leitfrage „Wie wird das System die Bedürfnisse erfüllen?“ und analysiert die in der SA modellierten Funktionen. Diese Funktionen werden in Subfunktionen zerlegt. Ebenfalls wird festgelegt, wie welche Subfunktionen miteinander interagieren. Des Weiteren wird das System in der LA in logische Komponenten zerlegt. Subfunktionen, welche die logischen Komponenten anbieten sollen, werden den Komponenten in diesem Schritt zugewiesen. Auch nicht-funktionale Anforderungen, wie Zuverlässigkeit oder Effizienz, an die Subsysteme und Systemelemente können in diesen Schritt modelliert und den entsprechenden logischen Komponenten und Funktionen zugeordnet werden.

Der vierte Schritt, die *Physical Architecture (PA)*, beschäftigt sich mit der tatsächlichen Umsetzung des Systems. Hier wird modelliert, wie die logischen Komponenten und Subfunktionen der LA tatsächlich implementiert werden. Die Leitfrage beim Modellieren des Systems in diesem Schritt lautet „Was wird entwickelt und wie wird es umgesetzt?“.

Der letzte Schritt, die End Product Breakdown Structure and Integration Contracts (EPBS), ist mit einer Verteilung der Entwicklungsarbeit an die verschiedenen Entwicklungsabteilungen oder Zulieferanten und Festhalten von Anforderungen an diese gleichzusetzen. Auch das wird mit einem Modell festgehalten, dargestellt und dokumentiert. Die EPBS ist somit getrennt von den anderen Schritten zu betrachten, da hier kein Systemmodell in dem Sinne einer Systemarchitektur entsteht, sondern eine Beschreibung der Entwicklungsarbeit vorgenommen wird.

Es ist anzumerken, dass diese Schritte nicht in der vorgestellten Reihenfolge abgearbeitet werden müssen. ARCADIA legt die Reihenfolge, wie ein System entwickelt wird, nicht fest [27]. ARCADIA kann unter anderem dazu verwendet werden, um ein bereits existierendes System zu untersuchen, wobei die Schritte in umgekehrter Reihenfolge als hier vorgestellt vorgenommen werden. Allerdings ergibt es Sinn, die verschiedenen Schritte in Reihenfolge zu erläutern, da jeder Schritt das Systemmodell des vorherigen mit mehr Informationen anreichert. *Capella*<sup>6</sup> ist das zu ARCADIA zugehörige Open-Source MBSE-Werkzeug, welches auch initial von Thales entwickelt worden ist. Capella integriert und implementiert die ARCADIA-Methode und bietet seinen Nutzern eine *Eclipse*-basierte<sup>7</sup> Plattform, um Systeme nach ARCADIA zu modellieren. In der Literatur ist Capella meist nicht von ARCADIA zu trennen. Auch sind Großteile der ARCADIA-Methode in Capella selbst dokumentiert, wie Juan Navas, leitender Modellierungs- und Simulations-Experte bei Thales, im Capella-Forum mitteilt<sup>8</sup>. Dementsprechend implementiert Capella nicht nur die ARCADIA-Methode, sondern dokumentiert auch den neusten öffentlich zugänglichen Stand der Methode. Besonders Anforderungen an Architektur-Diagramme und Modelle sind im Quellcode von Capella dokumentiert und haben sich über die Zeit hinweg weiterentwickelt und geändert. Da innerhalb der Bearbeitungszeit dieser Arbeit keinen Zugriff auf die XP Z67-140-Norm [25] erhalten werden konnte und der neuste Entwicklungsstand von ARCADIA scheinbar im Quellcode von Capella dokumentiert ist, wurden Informationen zu ARCADIA und Anforderungen an Sichten dem Capella-Quellcode entnommen. In ARCADIA und Capella sind jegliche Anforderungen an Modelle an das Modellierungsprojekt anzupassen. Die in Capella dokumentierten und in Form von Modellvalidierungsregeln implementierten Anforderungen sind laut S. Lacrampe nicht als generelle Anforderungen an Capella-Modellkonsistenz anzusehen, sondern sind projektspezifisch zu wählen und vollkommen optional, wie aus dem folgenden Kommentar<sup>9</sup> aus dem Capella-Forum hervorgeht:

„[...] validation rules are not here to run all of them, one should think about selecting the right ones that apply to a given project [...] in your case, just deactivate all validation rules [...].“

Capella-Nutzer werden ermutigt, eigene Anforderungen an ihre Modelle zu stellen und diese für Capella in Form von Modell-Validierern zu implementieren. Eine Dokumentation zur ARCADIA-Methode ist indirekt der Dokumentation der Capella-ML zu entnehmen. Die Capella-ML ist mithilfe von dem *Eclipse Modeling Framework (EMF)* und dessen ML *EMF core (Ecore)* spezifiziert worden. EMF ist eine beliebte MOF-Implementation [28]. Allerdings entspricht Ecore eher dem *Essential Modelling Object Facility (EMOF)*, dem essenziellen Anteil der MOF-Sprache. EMOF ist im Gegensatz zum *Complete Modelling Object Facility (CMOF)*, welche die gesamte MOF-Sprache beinhaltet, nur der elementare Teil von MOF, der zum Entwickeln von DSLs benötigt wird.

In den folgenden Abschnitten wird ausführlicher auf die ersten drei Schritte der ARCADIA-Methode eingegangen und anhand des laufenden Beispiels aus Abschnitt 2.1 und ausgewählten ARCADIA/Capella-Diagrammen das Modellieren und Entwickeln nach ARCADIA beispielhaft visualisiert. Die PA wird ausgelassen, da dort alle Vorgehensweisen, Konzepte und Diagramme der LA wiederverwendet werden, aber eine tatsächlich realisierbare Systemarchitektur angefertigt wird. Auch die EPBS wird nicht aufgezeigt, da hier keine neuen

<sup>6</sup><https://github.com/eclipse/capella>

<sup>7</sup><https://www.eclipse.org/>

<sup>8</sup><https://forum.mbse-capella.org/t/arcadia-rules-available/5076>

<sup>9</sup><https://forum.mbse-capella.org/t/what-is-the-proper-way-of-using-capella/5289/4>

Konzepte verwendet werden und lediglich eine Zuordnung von Anforderungen und Entwicklern an die zu realisierenden Komponenten stattfindet.

### 2.7.1. Operational Analysis

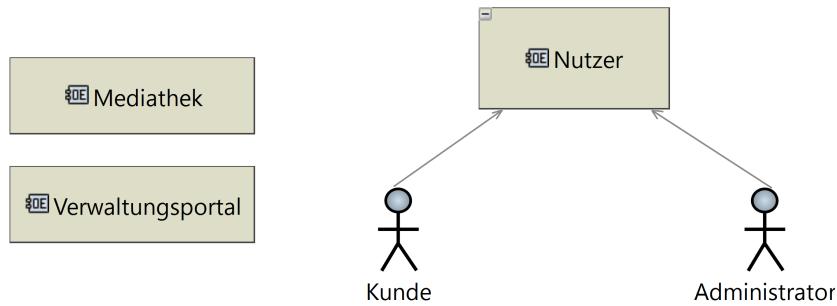


Abbildung 2.22.: Operational Entity Breakdown (OEBD) des Streamingdienstes aus Abschnitt 2.1.

Das Ziel der **Operational Analysis** ist das Spezifizieren der Bedürfnisse und Ziele der Interessengruppen. Dies geschieht durch das Beschreiben von Interaktionen zwischen Fähigkeiten und Aktivitäten von nicht zum System gehörenden Elementen wie zukünftige Nutzer oder andere Systeme. Hierfür werden zuerst alle *operationale Entitäten* (auf engl. „Operational Entities“) festgestellt und modelliert. Operationale Entitäten sind Objekte oder Subjekte, also Personen oder Systeme, welche miteinander und mit dem zu entwickelnden System interagieren. Ist eine solche Entität eine Person, so wird von einem *operationalen Akteur* gesprochen. Der Zusammenhang, welches operationale Entitäten existieren und wie diese in Beziehung zueinander stehen, wird in Capella mit einem OEBD-Diagramm, wie in Abbildung 2.22, modelliert. Capella-Breakdown-Diagramme visualisieren die Existenz von Subjekten oder Objekten und deren Abhängigkeiten zueinander. Im Falle des Streamingdienstes existieren **Nutzer**, eine **Mediathek** und ein **Verwaltungsportal**, welche miteinander und mit dem System Streamingdienst interagieren werden. Es ist allerdings unerwünscht, dass alle **Nutzer** den Dienst mittels des **Verwaltungsportals** verwalten dürfen. Deswegen lässt sich die Entität **Nutzer** in die Untertypen **Kunde** und **Administrator** unterteilen. Die Unterteilung wird wie bei der Vererbung in Klassendiagrammen mit einem Pfeil von dem Untertypen zum Obertypen, in diesem Fall von dem **Kunden** zum **Nutzer**, symbolisiert.

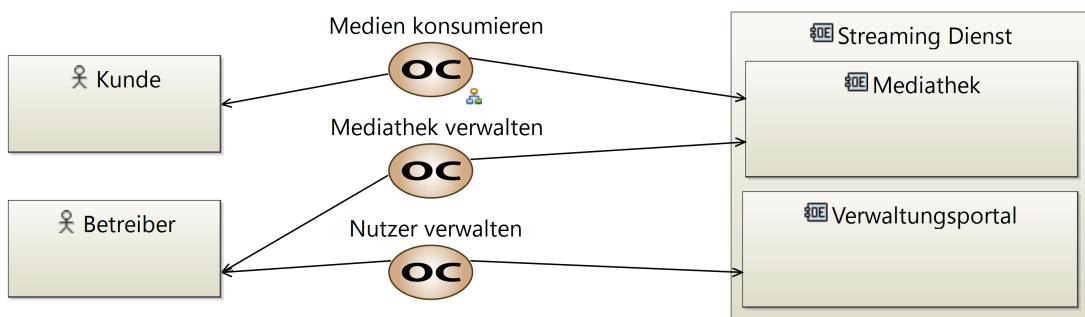


Abbildung 2.23.: Operational Capability Blank (OCB) des Streamingdienstes aus Abschnitt 2.1.

Nachdem die operationale Entitäten definiert sind, können *operationale Fähigkeiten* (auf engl. „Operational Capabilities“) mit einem Universal Modelling Language (UML) Use-Case-Diagramm ähnlichen OCB-Diagramm, wie in Abbildung 2.23, modelliert und zugeteilt werden. Operationale Fähigkeiten sind Dienstleistungen

oder Funktionen, welche das System anbieten soll, um deren Ziele zu erfüllen. Beispielsweise sollen alle **Nutzer** des Dienstes, in der Lage sein, Medien zu konsumieren. Demnach benötigen **Nutzer** die operationale Fähigkeit **Medien streamen**. Diese Fähigkeit betrifft sowohl die **Nutzer** als auch die **Mediathek** des Dienstes, da dort die Medien gespeichert werden. Diese Zuteilung wird mit Pfeilen, ausgehend von der Fähigkeit zu den betreffenden Entitäten, symbolisiert. In diesem Diagramm sind **Kunden** und **Administratoren** als Teil des **Nutzer**-Blocks dargestellt. Hiermit wird visuell ausgedrückt, dass alle Fähigkeiten, welche einen **Nutzer** betreffen, auch für **Kunden** und **Administratoren** gelten. Weitere Fähigkeiten dargestellte Fähigkeiten sind, dass **Administratoren** mit dem **Verwaltungsportal** die **Mediathek** und andere **Nutzer** verwalten können. Nach Modellierung der Entitäten und deren Fähigkeiten können die zugehörigen *operationalen Aktivitäten* definiert und modelliert werden. Aktivitäten sind in diesem Kontext die Aktionen von und die Interaktionen zwischen Entitäten, in welche sich operationale Fähigkeiten zerlegen lassen. Nach der Definition der Aktivitäten kann die Fähigkeit in einem beispielhaften Szenario mit einem Operational Entity Scenario (OES)-Diagramm modelliert werden. Ein OES-Diagramm ist im Grunde ein UML Sequenz-Diagramm, mit dem die Aktivitäten von verschiedenen Entitäten auf vertikalen Zeitachsen aufgetragen werden. Abbildung 2.24 zeigt ein beispielhaftes Szenario für die Fähigkeit **Medien streamen**, welches sich in die vier Aktivitäten **Video vorschlagen**, **Video auswählen**, **Video bereitstellen** und **Video anschauen** zerlegen lässt. An der Fähigkeit **Medien streamen** sind **Nutzer** und die **Mediathek** beteiligt, demnach die zwei Zeitachsen des Diagramms zu diesen Entitäten. Das Szenario beschreibt, wie ein **Nutzer** ein Video aus den Vorschlägen der **Mediathek** auswählt, dieses anschließend von der **Mediathek** an den **Nutzer** ausgeteilt und vom Nutzer konsumiert wird.

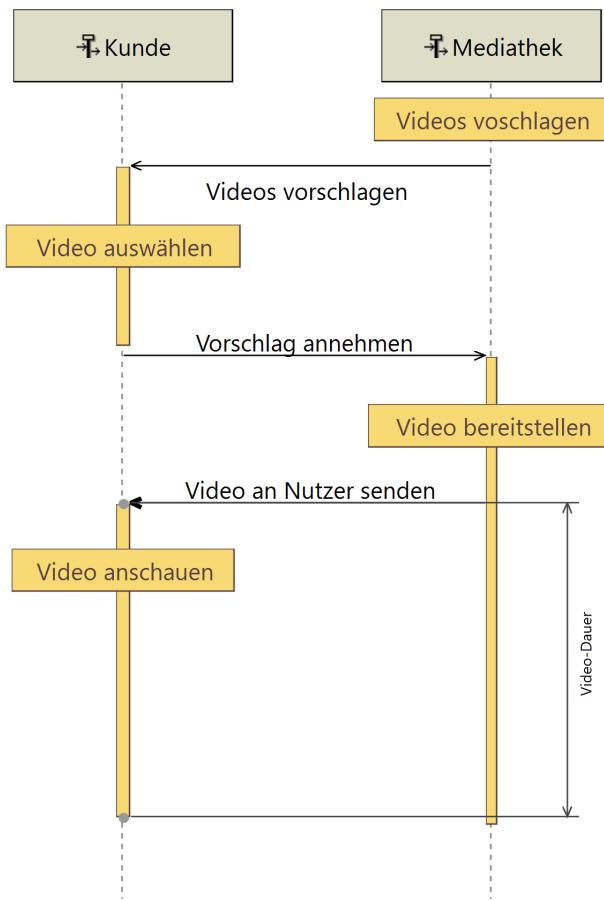


Abbildung 2.24.: OES des Streamingdienstes aus Abschnitt 2.1.

## 2.7.2. System Need Analysis

Der Fokus der **System Need Analysis** liegt darauf, welche Funktionen das System anbieten werden muss, damit die operationalen Fähigkeiten implementiert werden können. Da die ARCADIA-Methode besagt, dass in jedem Schritt alle Modellelemente aus dem vorherigen Schritt auch repräsentiert werden müssen, werden im folgenden Realisierungen für die Modellelemente aus Abschnitt 2.7.1 erzeugt und mit Informationen angereichert. Operationale Akteure werden als *System-Akteure* und operationale Entitäten als *System-Komponenten* dargestellt. Fähigkeiten, in denen Akteure und Komponenten eingebunden sind, werden als *System-Fähigkeiten* dargestellt. Die durch das Zerlegen von Fähigkeiten entstehenden Aktivitäten werden allerdings nun als *System-Funktion* betrachtet. Die zuvor genannten neuen Repräsentationen für die Konzepte der OA werden in der SA eingeführt, um den Perspektivenwechsel von OA zu SA zu betonen und das Anreichern von Informationen, welche durch die SA-Sichtweise auf das System gewonnen werden können, zu ermöglichen.

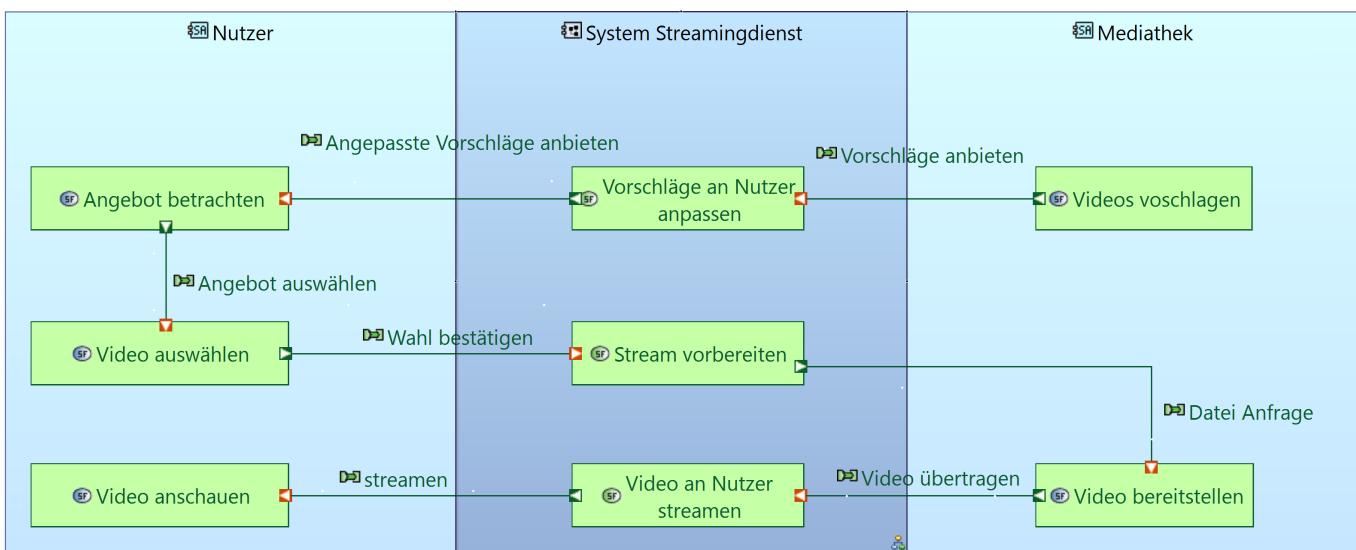


Abbildung 2.25.: **System Architecture Blank (SAB)** des Streamingdienstes aus Abschnitt 2.1.

In der SA sollen nun nicht mehr die Bedürfnisse von den Interessengruppen und anderen externen Systemen betrachtet werden, sondern die Bedürfnisse des Systems selbst. Dementsprechend müssen die Modellelemente der OA erneut untersucht werden und sind somit aus der Sichtweise des Systems neu darzustellen. Im SA-ARCADIA-Schritt ist das Ziel die Funktionen genauer zu definieren, in Sub-Funktionen zu zerlegen, und weitere Funktionen für eine mögliche Implementierung der Fähigkeiten zu modellieren. Am Ende dieses Schrittes entsteht als Hauptprodukt ein SAB-Diagramm, welches ausgewählte Elemente der System-Architektur mit einem SA-Fokus visualisiert. Im Folgenden wird auf Basis des Diagramms in Abbildung 2.24 vorgeführt, wie das SAB-Diagramm in Abbildung 2.25 erstellt wird.

Die Entitäten **Nutzer** und **Mediathek** aus Abbildung 2.24 müssen in der SA repräsentiert werden, somit sind diese auch in Abbildung 2.25 als System-Komponenten (hellblaue Rechtecke) dargestellt. Da nun die Funktionen, welche das System anbieten werden muss, untersucht werden, wird nun erstmals das System **Streamingdienst** in diesem Modell (als dunkelblaues Rechteck) repräsentiert. Die Aktivitäten des **Nutzers**, **Video auswählen** und **Video anschauen**, sind als Funktionen (grüne Rechtecke) repräsentiert und durch das Darstellen innerhalb des **Nutzer**-Rechtecks konkret zugeordnet. Äquivalent sind die Aktivitäten der **Mediathek**, **Videos vorschlagen** und **Videos austeilten**, als Funktion dargestellt und der **Mediathek** zugeordnet. Ein

**Streamingdienst** ist in der Regel in der Lage, das Angebot an Medien an spezifische **Nutzer** anzupassen, um den Zugang zu bevorzugten Inhalten zu vereinfachen. Hierfür sollte das System in der Lage sein **Vorschläge an den Nutzer anzupassen** und somit kann dies als Funktion zugeordnet werden. Nachdem das Angebot an den **Nutzer** angepasst wurde, sollte der **Nutzer** auch die Fähigkeit haben, das **Angebot zu betrachten** und daraus ein Video auszuwählen. Dementsprechend wird der Funktionsumfang des **Nutzers** um die Funktion **Video auswählen** erweitert. Nachdem der **Nutzer** seine **Wahl bestätigt** hat, sollte das System die Funktion **Stream vorbereiten** anbieten, damit beispielsweise das Abspielen unterbrechungsfrei geschehen kann. Ist die Datei von der **Mediathek** aufgefunden und das Video an den **Streamingdienst** übermittelt worden, so kann der **Streamingdienst** die Datei an den **Nutzer** verteilen, ohne dass die **Mediathek** die Übertragung an alle **Nutzer** übernehmen muss. Das System übernimmt das Streaming und dementsprechend muss es die Funktion **Video an Nutzer streamen** anbieten.

### 2.7.3. Logical Architecture

In der Logical Architecture ist die Untersuchung der System-Funktionen zentral. Die Funktionen werden in *logische Funktionen* aufgeteilt und die Art und Weise, in welcher die logischen Funktionen miteinander interagieren, wird modelliert. Auch müssen das System und die System-Komponenten beschrieben und in *logische Komponenten* unterteilt werden, damit ihnen die logischen Funktionen zugeteilt werden können. Kommunikationswege und Art der Interaktion zwischen logischen Komponenten und Funktionen sind auch zu modellieren. Das in diesem Schritt entstehende Modell repräsentiert, im Gegensatz zur OA und SA, keine Bedürfnisse, sondern eine tatsächliche abstrakte Implementierung des Systems, welche die Bedürfnisse erfüllen kann.

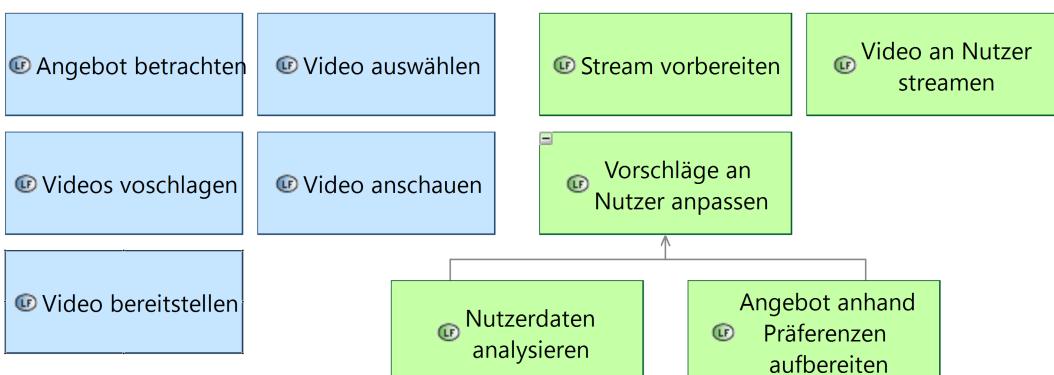


Abbildung 2.26.: Logical Function Breakdown (LFBD) des Streamingdienstes aus Abschnitt 2.1.

Das Zerlegen von Funktionen in Sub-Funktionen wird mit einem LFBD-Diagramm modelliert. Hier sind alle Funktionen, welche in der SA definiert wurden, dargestellt. Diese Funktionen werden durch das Anlegen neuer Funktionen verfeinert und mittels Pfeilen einander zugeordnet. In Abbildung 2.26 ist ein beispielhaftes LFBD-Diagramm dargestellt. Alle Funktionen, welche zuvor in der SA den System-Komponenten zugeordnet wurden, welche in der OA als System-externe Entitäten modelliert worden sind, werden als blaue Rechtecke dargestellt. Im Gegensatz dazu sind die Funktionen, welche dem System zugeordnet wurden, grün dargestellt. Hiermit wird dem Capella-Nutzer die Unterscheidung zwischen Funktionen des Systems und Funktionen System-externer Entitäten erleichtert. In Abbildung 2.26 wird auch beispielhaft die Funktion **Vorschläge an Nutzer anpassen** in die Subfunktionen **Nutzerdaten analysieren** und **Angebot anhand Präferenzen aufbereiten**

unterteilt. Der Pfeil ausgehend von einer Funktion zu einer anderen symbolisiert, dass die Funktion ein Teil der anderen ist.

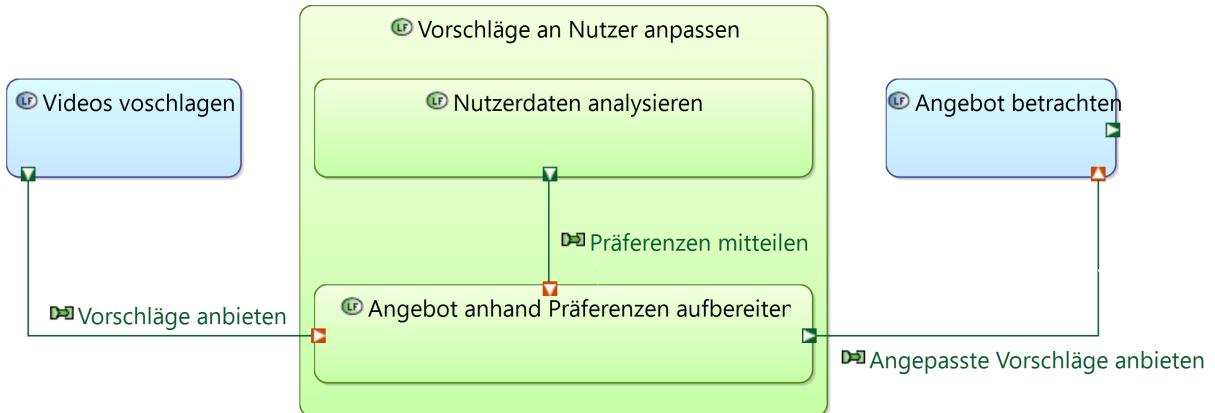


Abbildung 2.27.: Logical Dataflow Blank (LDFB) des Streamingdienstes aus Abschnitt 2.1.

Nachdem die Funktion **Vorschläge an Nutzer anpassen** zerlegt worden ist, kann nun mit einem LDFB-Diagramm modelliert werden, wie die Funktionen miteinander interagieren und kommunizieren. Dies geschieht analog zu dem Vorgehen wie im SAB-Diagramm in Abbildung 2.25, nur dass keine System-Komponenten und nur ausgewählte Funktionen dargestellt werden. Das LDFB-Diagramm in Abbildung 2.27 stellt ausschließlich die Funktion **Vorschläge an Nutzer anpassen**, dessen Subfunktionen, und die Funktionen **Videos vorschlagen** und **Angebot betrachten**, mit welchen interagiert wird, dar.

Normalerweise enthält das LDFB alle Funktionen und Subfunktionen der gesamten LA. Hier wird aber das Beispiel auf die genannten Funktionen reduziert, damit die LA-Konzepte überschaubar und kompakt vermittelt werden können. Des Weiteren wird ein weiterer Kommunikationskanal zwischen den Subfunktionen **Nutzerdaten analysieren** und **Angebot anhand Präferenzen filtern** modelliert, womit das zum Filtern notwendige *Mitteilen der Präferenzen* als ein Informationsaustausch modelliert wird. Nachdem alle System-Komponenten und Funktionen zur Genüge in logische Komponenten und logische Funktionen zerteilt sind, kann die logische Architektur in einem Logical Architecture Blank (LAB)-Diagramm dargestellt werden. Das Vorgehen und Ergebnis ist mehr oder weniger identisch zum Erstellen eines SAB. Der Unterschied liegt hierbei vor allem im Detailgrad. Das SAB-Diagramm ist intuitiv als eine abstrakte und vereinfachte Darstellung des LAB-Diagramms zu verstehen. LAB-Diagramm und LDFB-Diagramme sind die Hauptprodukte der LA.

## 2.8. Das Capella MBSE-Werkzeug

In den folgenden Abschnitten wird das Capella MBSE-Werkzeug in der Version 5.2<sup>10</sup>, welches die ARCADIA-MBSE-Methode implementiert, vorgestellt. Anschließend werden auf ausgewählte Funktionen der kommerziellen Capella-Erweiterung Team for Capella ebenfalls in der Version 5.2<sup>11</sup> eingegangen, welche verwendet werden kann, um mit verschiedenen Nutzern simultan ein Capella-Projekt zu editieren.

<sup>10</sup><https://github.com/eclipse/capella/tree/v5.2.0>

<sup>11</sup><https://www.obeosoft.com/en/team-for-capella-releases>

## 2.8.1. Die Capella Nutzeroberfläche

Capella basiert in der Version 5.2 auf der Eclipse-IDE 2020-06<sup>12</sup> Entwicklungsumgebung, womit das generelle Layout der Bedienungsoberfläche identisch zu Eclipse ist. Nach Öffnen oder Erstellen eines Projekts werden die Projekt-Dateien im *Projekt-Explorer*, in Abbildung 2.28 blau umrandet, angezeigt und der *Activity-Explorer*, in Abbildung 2.28 rot umrandet, wird für das Projekt geöffnet. Der Projekt-Explorer ist ein hierarchisches Drop-down-Menü, welches Projekte, Projekt-Dateien und zugehörige Diagramme anzeigt. Rechts eingerückte Elemente sind im darüber links eingerückten Element enthalten. Die Elemente, welche im Projekt-Explorer am weitesten links eingerückt sind, sind Capella-Projekte. In Abbildung 2.28 ist das Projekt *StreamingDienst* und dessen Bestandteile zu sehen. Jedes Capella-Projekt besteht aus drei Dateien. Die *afm*-Datei beinhaltet Capella-spezifische Daten und Einstellungen. Unter anderem wird in dieser Datei gespeichert, welche Capella Version verwendet wurde und welche zusätzlichen Erweiterungen eingesetzt worden sind. Modelldaten und -instanzen eines Projekts sind in der *capella*-Datei (ehemals „melodymodeller“ genannt) gespeichert. Die *aird*-Datei ist der Speicherort aller Diagramme des Projekts. Die Modelle des *StreamingDienst*-Projekts werden im Projekt-Explorer als Teil der *aird*-Datei angezeigt und Modellelemente können nach den ARCADIA-Schritten gruppiert ausgewählt und editiert werden. Auch Diagramme können über das „Representations per category“ Menü-Element nach dem ARCADIA-Schritt gruppiert eingesehen und ausgewählt werden. Der Activity-Explorer ist ein graphisches Menü, welches dem Nutzer das Entwickeln nach ARCADIA erleichtern soll. Die Hauptansicht dieses Menüs, welche in Abbildung 2.28 rot umrandet ist, besteht aus jeweils einem pfeilförmigen Knopf für jeden ARCADIA-Schritt mit einer kurzen Beschreibung des Ziels des Schrittes.

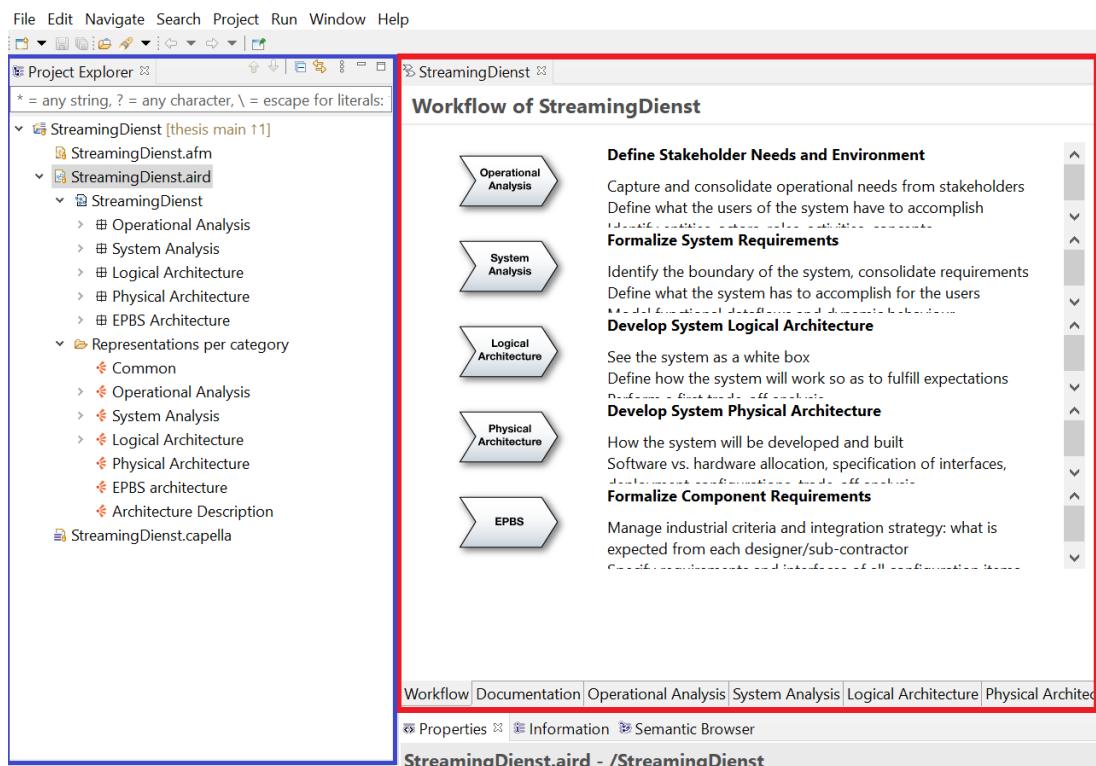


Abbildung 2.28.: Capella Nutzeroberfläche nach Öffnen eines Projekts. Projekt-Explorer blau und Activity-Explorer rot umrandet.

<sup>12</sup><https://projects.eclipse.org/releases/2020-06>

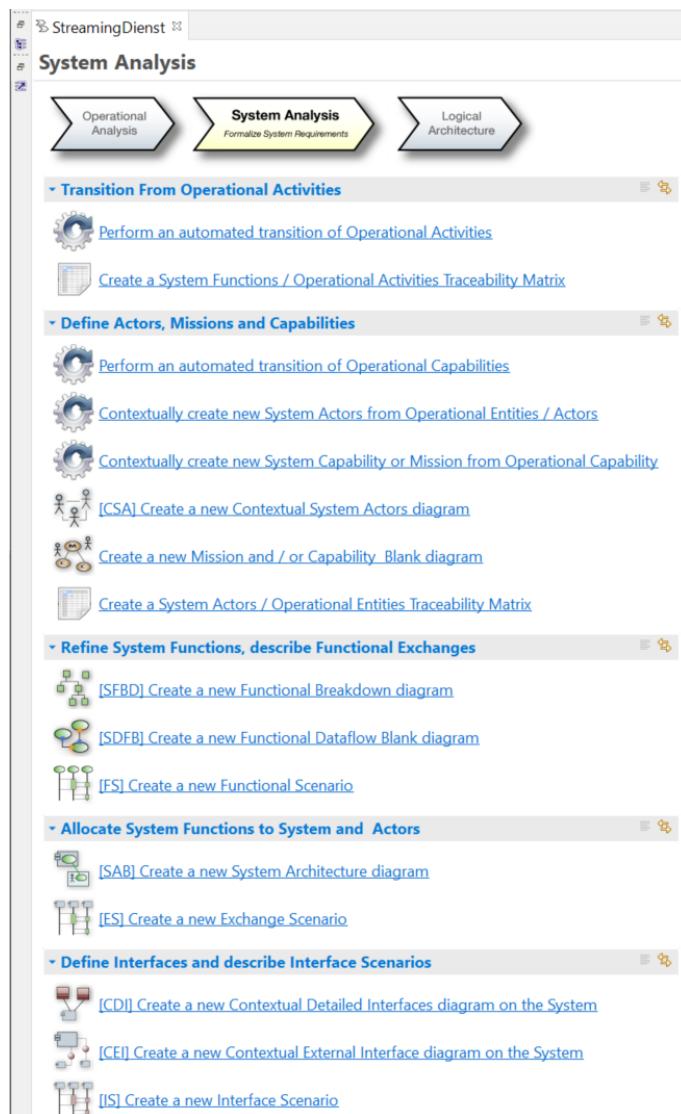


Abbildung 2.29.: Activity-Explorer nach Wählen des SA ARCADIA-Schrittes.

Wie Abbildung 2.29 anhand der SA gezeigt wird, öffnet sich nach Anklicken eines dieser Knöpfe eine Übersicht im Activity-Explorer mit einer Auswahl an Modellierungstätigkeiten des gewählten ARCADIA-Schrittes. Zu beachten ist jedoch, dass über den Activity-Explorer nicht alle Modellierungstätigkeiten dem Nutzer zur direkten Verfügung gestellt werden, sondern einige essenzielle Funktionen nur durch das Öffnen von Kontextmenüs im Projekt-Explorer erreichbar sind. Abbildung 2.30 zeigt das Kontextmenü nach dem Rechtsklicken auf eine operationale Entität des OA-Modells. Es werden unter anderem die Optionen „Add Capella Element“, zum Erzeugen neuer Modellelemente, und „New Diagram/Table...“, zum Hinzufügen neuer Capella-Diagramme, angezeigt und sind mit einem grünen Rechteck hervorgehoben.

Wie in Abschnitt 2.7 erwähnt, muss jedes Modellelement eines ARCADIA-Schritts im Modell des folgenden Schritts zumindest repräsentiert werden. Damit Nutzer nicht manuell die Realisierungen anlegen müssen, bietet Capella die *Transitions*-Funktion an. Eine Transition von Elementen legt für diese Elemente eine Realisierung im Modell des folgenden ARCADIA-Schrittes an und erzeugt eine Verknüpfung zwischen jedem ursprünglichen Element und dessen Realisierung. In der Regel gibt es für jeden Capella-Modell-Elementtypen

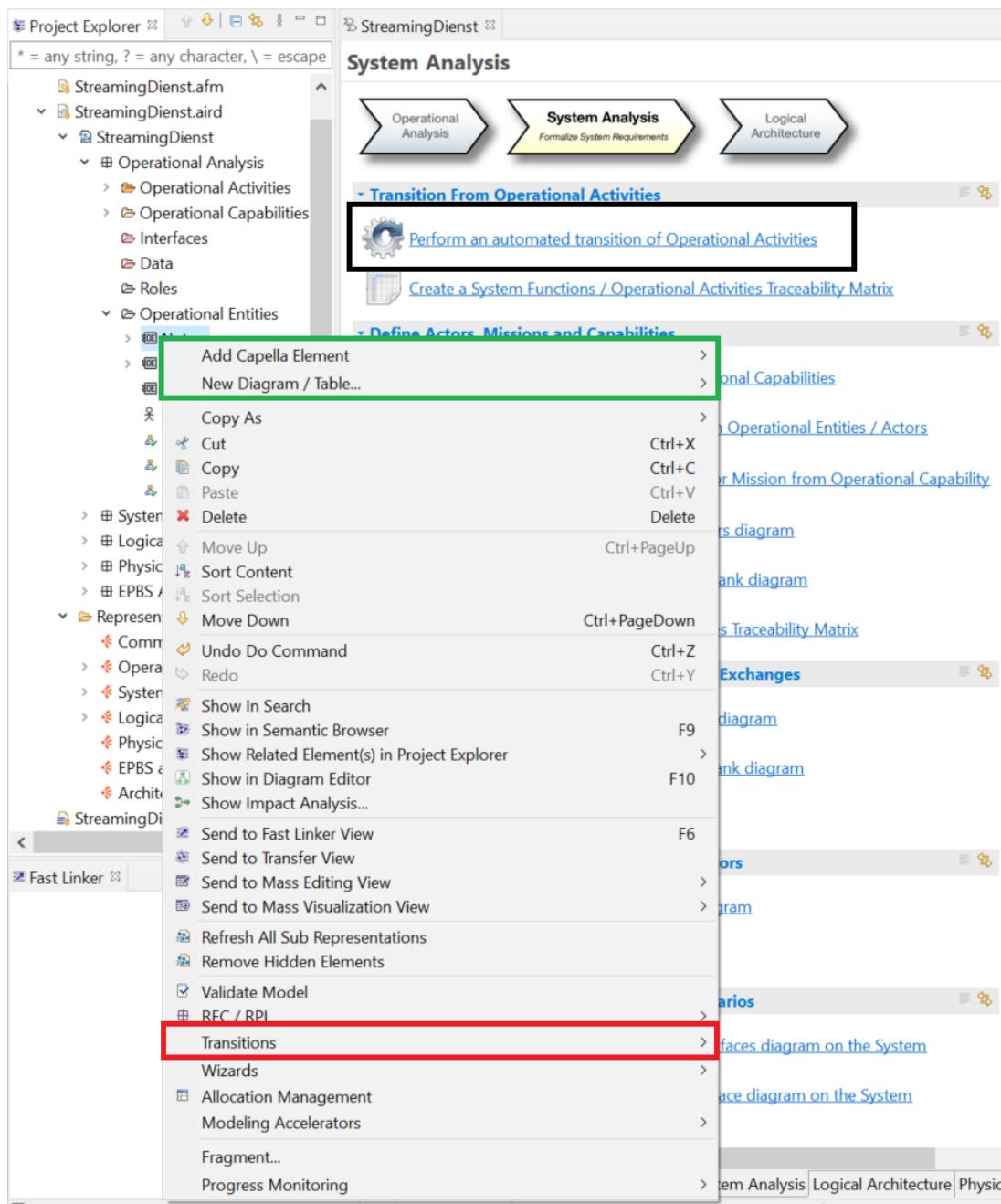


Abbildung 2.30.: Activity- und Projekt-Explorer. Menü zum Erstellen von Diagrammen und Modellelementen grün umrandet. Beispielhafte Transitionsmöglichkeit im Activity-Explorer Schwarz umrandet und Menü aller Transitionsmöglichkeiten im Kontextmenü Rot umrandet.

eine Transitions-Funktion, welche über den Activity-Explorer oder dem Projekt-Explorer aufgerufen werden kann. Transitions-Funktionen sind im Activity-Explorer mit einem zahnradförmigen Knopf markiert, wie in Abbildung 2.30 beispielhaft mit einer schwarzen Umrandung hervorgehoben ist. Diese Transitions-Funktion ist mit „Perform an automated transition of Operational Analysis“ (engl. für „Automatisierte Transition der OA durchführen“) beschriftet. Obwohl die Beschriftung darauf schließen lässt, dass eine Transition aller Modellelemente durchgeführt werden würde, ist dies allerdings nicht der Fall und es werden nicht zwingend

Realisierungen für alle Element-Typen der OA in der SA angelegt. Beispielsweise werden mit dieser Transitions-Funktion keine operationale Entitäten oder Szenarien (siehe Abbildung 2.24) von der OA in die SA überführt. Dies muss entweder manuell vorgenommen werden, oder es müssen mehrere andere Transitions-Funktionen verwendet werden. Des Weiteren werden nicht alle Transitions-Funktionen im Activity-Explorer angeboten. Die Restlichen sind im Projekt-Explorer in Kontext-Menüs zu finden. Die Transition von Elementen mit dem Projekt-Explorer erfolgt durch das Auswählen eines Elements und Öffnen des Kontextmenüs mittels Rechtsklicken auf das entsprechende Element. In diesem Kontextmenü, wie in Abbildung 2.30 für eine operationale Entität zu sehen ist, lässt sich dann das Sub-Menü „Transitions“ (in der Abbildung rot umrandet) öffnen und die erwünschte Transitions-Funktion für das Element auswählen.

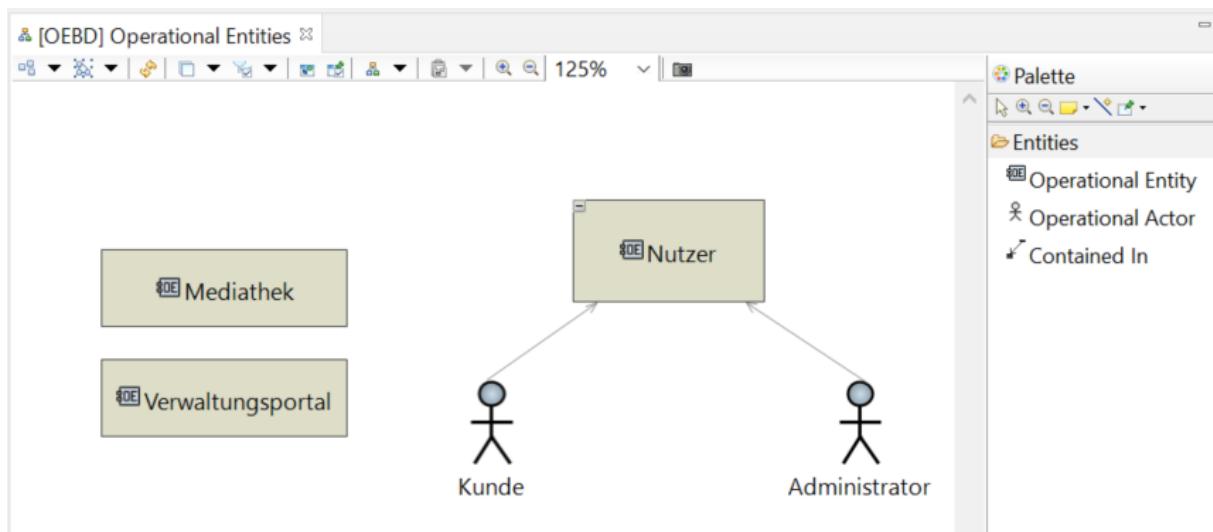


Abbildung 2.31.: Capella Nutzeroberfläche beim Editieren des OEBD aus Abbildung 2.23.

Das Erstellen und Editieren von Diagrammen wird in Capella mit Sirius-basierte<sup>13</sup> Diagramm-Editoren ermöglicht. In Abbildung 2.31 wird beispielhaft der Diagramm-Editor für OEBD-Diagrammen gezeigt, mit dem auch das Diagramm aus Abschnitt 2.7.1 erstellt wurde. Auf der rechten Seite wird dem Nutzer eine Palette angeboten, mit welcher diagrammspezifische Modellelemente erzeugt werden können. Ein OEBD-Diagramm visualisiert alle operationale Entitäten. Dementsprechend lassen sich mit den Schaltflächen in Abbildung 2.31 rechts operationale Entitäten und Akteure anlegen. Auch Kompositionenbeziehungen sind mit der „Contained In“-Schaltfläche erstellbar. Hierfür wird zuerst die Schaltfläche, dann eine Entität oder einen Akteur und anschließend die Entität, welche die andere enthalten soll, ausgewählt. Je nach Diagrammtyp werden bereits existierende Modellelemente automatisch im Diagramm angezeigt. Beispielsweise werden in Breakdown-Diagrammen Modellelemente nach deren Erzeugung automatisch dem Diagramm hinzugefügt. Im Kontext eines OEBD-Diagramms würde dies bedeuten, dass eine operationale Entität, welche in einem anderen Diagramm angelegt worden ist, automatisch dem OEBD-Diagramm hinzugefügt wird. Sieht ein Diagrammtyp das automatische Hinzufügen von Elementen nicht vor, so wird eine Schaltfläche rechts angeboten, mit welcher vorhandene Modellelemente dem Diagramm hinzugefügt werden können, falls diese noch nicht im Diagramm abgebildet sind.

<sup>13</sup><https://www.eclipse.org/sirius/>

## 2.8.2. Team for Capella

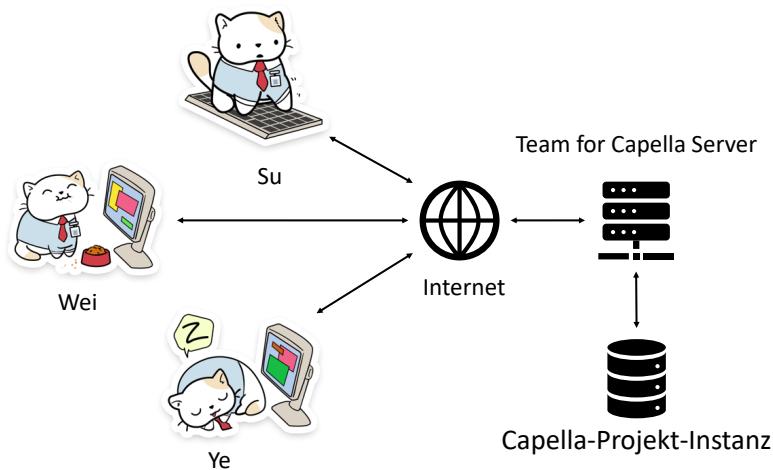


Abbildung 2.32.: Visualisierung der grundlegenden Funktionsweise der *Team for Capella* Erweiterung anhand Nutzer Su, Wei und Ye, welche an einer gemeinsamen Capella-Projekt-Instanz arbeiten.

Da, wie bereits erwähnt, Anforderungen an die Konsistenz von Capella-Modellen optional sind, können Modelle sehr frei gestaltet und Anforderungen an die Modellstruktur flexibel an ein Projekt angepasst werden. Allerdings erschwert dieser Freiheitsgrad den Umgang mit Capella-Modellen beim gemeinsamen Modellieren von Systemarchitekturen. Wie in Abschnitt 2.2 und Abschnitt 2.3 dargestellt wurde, ist die Entwicklung einer qualitativ hochwertigen Systemarchitektur eine sehr komplexe Angelegenheit und wird in der Praxis von einer Vielzahl von verschiedenen Personen mit unterschiedlichen Interessen umgesetzt. Dementsprechend müssen die Systemmodelle, welche mit Capella angelegt werden, auch oft von verschiedenen Capella-Nutzern gleichzeitig bearbeitet werden können. Allerdings unterstützt Capella kein gemeinsames Bearbeiten von Capella-Projekten. Eine naheliegende Lösung, um diese Einschränkung zu umgehen, ist das Kopieren von Capella-Modellen, das Bearbeiten der Kopien und das anschließende Zusammenführen der bearbeiteten Kopien in eine neue Modellinstanz. Gerade dieser Ansatz ist schwer umzusetzen, da ein erfolgreiches Darstellen aller Änderungen an den Projekt-Kopien das Erkennen von Konsistenzverletzungen und Auflösen von möglichen Konflikten beinhaltet, und ist nicht trivial und in der Regel sehr aufwendig. Des Weiteren existieren derzeit wenige Lösungen zum Zusammenführen von verschiedenen Capella-Projekt-Kopien zu einem neuen Ursprungsmodell, in welchem alle Änderungen widergespiegelt sind. Da Capella-Modelle EMF-basiert sind, lässt sich das Werkzeug *EMF Diff/Merge*<sup>14</sup> verwenden, um zwei Capella-Projekte miteinander zu vergleichen. Capella bietet dem Nutzer eine angepasste Version des EMF Diff/Merge Werkzeugs an, mit welchem der Inhalt zweier *capella*-Dateien verglichen und einzelne Differenzen dieser Dateien angezeigt werden können. Das Zusammenführen dieser zwei Capella-Projekt-Kopien nach Vergleich der *capella*-Dateien muss anschließend vom Nutzer vorgenommen werden, indem entschieden wird, welche Änderungen übernommen und welche verworfen werden. Des Weiteren ist EMF Diff/Merge nicht in der Lage, Anforderungen an die Konsistenz von Capella-Modellen zu berücksichtigen und Verletzungen an die Konsistenz anzuzeigen, obwohl Capella bereits die Konsistenz eines Modells mit Validierungsregeln überprüfen kann. Diese Funktionalität zu implementieren, ist dem Capella-Nutzer überlassen. Capella bietet zwar die Funktionalität an, die Konsistenz eines Capella-Projekts zu validieren und auf Konsistenzverletzungen hinzuweisen, allerdings sind die implementierten Validierungsregeln in dem Sinne unvollständig, dass Nutzer die Validierungsregeln anpassen oder gar selbst

<sup>14</sup><https://projects.eclipse.org/projects/modeling.emf.diffmerge>

implementieren müssen. Es ist somit nach Implementation fehlender Validierungsregeln theoretisch möglich, mit Capella das nach Zusammenführen der geänderten Projekt-Kopien resultierende Capella-Projekt auf Konsistenz zu prüfen. Das Wiederherstellen von Konsistenz muss aber manuell vorgenommen werden. Anstatt verschiedene Modell-Kopien zu editieren und Änderungen zusammenzuführen, kann die proprietäre Capella-Erweiterung *Team for Capella*<sup>15</sup> genutzt werden. Team for Capella soll das gemeinsame und simultane Bearbeiten eines Capella-Projekts ermöglichen, indem eine gemeinsame Capella-Projekt-Instanz von allen Teilnehmern am Entwicklungsprozess geteilt wird. Ein Server stellt die geteilte Projekt-Instanz zur Nutzung bereit und verwaltet diese. Nutzer können sich mit diesem Server verbinden und auf die geteilte Projekt-Instanz zugreifen, wie in Abbildung 2.32 anhand Nutzern Ye, Su und Wei visualisiert wird.

Durch das Teilen der gemeinsamen Instanz wird vermieden, dass verschiedene Projekt-Kopien entstehen, deren Differenzen gefunden, verglichen und zusammengeführt werden müssen, da alle Nutzer Änderungen direkt an derselben Projekt-Instanz vornehmen. Die Capella-Werkzeug-Instanzen der Nutzer müssen dafür jederzeit mit dem Server verbunden sein, um Zugriff auf die Projekt-Instanz zu erhalten. Problematisch beim Teilen einer Projekt-Instanz ist das gleichzeitige Editieren von Capella-Modellen, da die Änderungen in Konflikt zueinander stehen können. Team for Capella löst dies, indem Modellelemente und deren näheren Umgebung gesperrt werden und Nutzer diese nicht gleichzeitig editieren können. Die allgemeine Verwendung von Team for Capella und das Sperren von Modellelementen wird im Folgenden beispielhaft anhand der Capella-Modelle aus Abschnitt 2.7 dargestellt.

Die Nutzung von Team for Capella ist aus Sicht des Capella-Nutzers nur marginal unterschiedlich zur Nutzung von Capella ohne die Erweiterung. Der größte Unterschied besteht darin, dass ein Capella-Projekt bei Verwendung von Team for Capella keine *capella*-Datei enthält, wie in Abbildung 2.33b dargestellt wird.

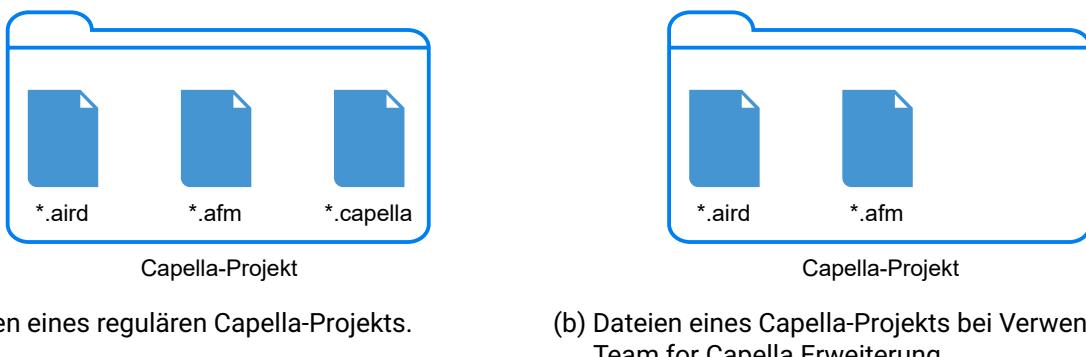


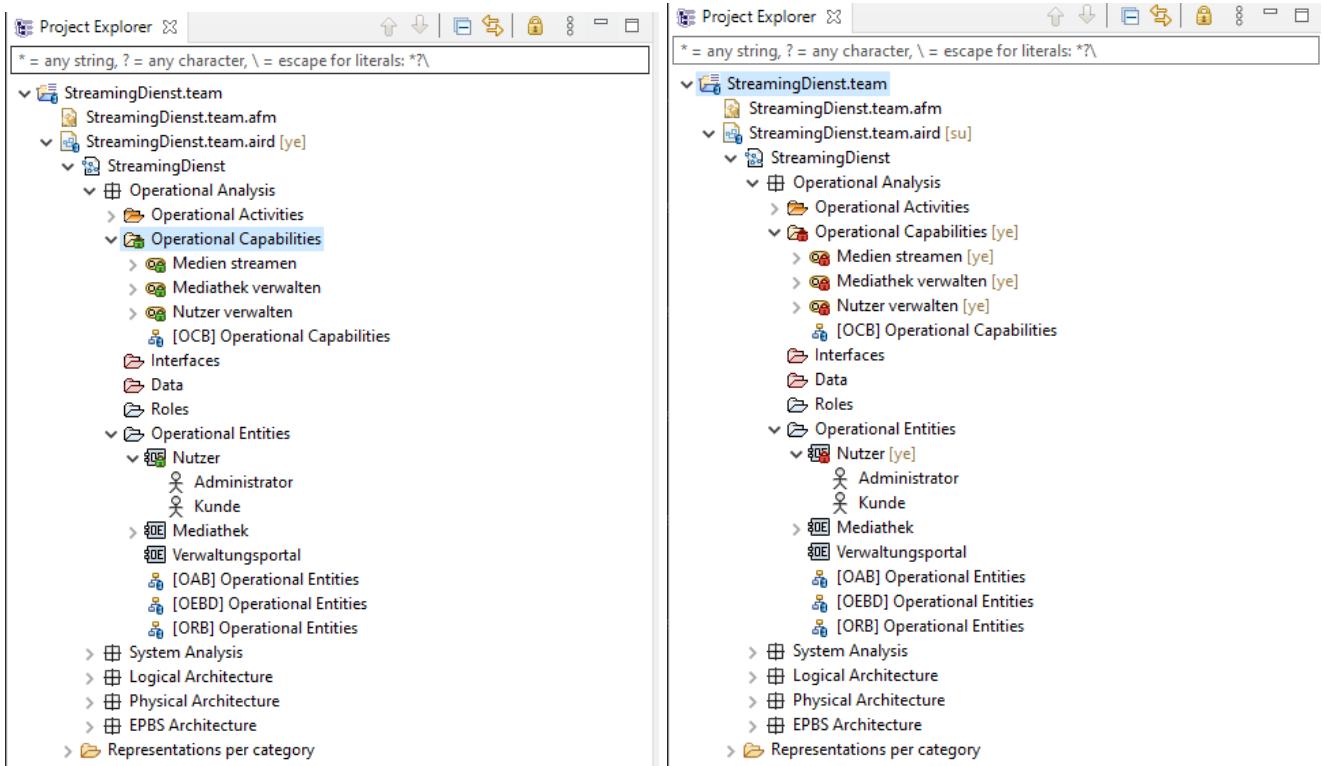
Abbildung 2.33.: Vergleich der Capella-Projekt-Dateien zwischen regulären Projekten und Projekten bei Verwendung von Team for Capella.

Wie bereits in Abschnitt 2.8.1 erläutert, besteht ein reguläres Capella-Projekt aus einer *aird*-, *afm*- und einer *capella*-Datei. Letztere Datei ist allerdings nicht Teil eines Capella-Projekts bei Verwendung von Team for Capella. Die *capella*-Datei beinhaltet alle Modelle des Projekts und direkter Zugriff auf diese Datei ist mit einem direkten Zugriff auf die Modelle gleichzusetzen. Direkter Zugriff auf die Modelle durch das Öffnen der *capella*-Datei ist unerwünscht, weswegen diese Datei nicht angeboten wird, sondern alle Modelle zentral vom Server verwaltet und bereitgestellt werden. Auch direkter Zugriff auf die Diagramme eines Projekts ist nicht möglich. Den Nutzern werden lediglich Kopien der Diagramme ausgeteilt. Änderungen an diesen Diagrammen werden ebenfalls vom Server verwaltet und eingepflegt. Durch diese zentrale Verwaltung muss beim Bearbeiten eines Projekts jederzeit eine Verbindung zum Server vorhanden sein. Das es nicht möglich

<sup>15</sup><https://www.obeosoft.com/en/team-for-capella>

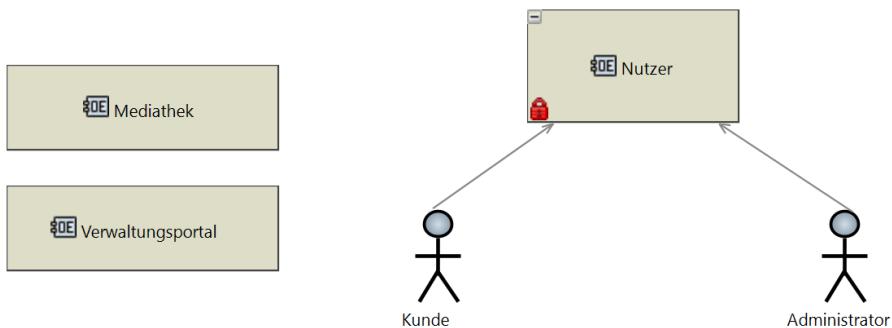
ist, die Capella-Modelle direkt durch das Editieren der *capella*-Datei zu bearbeiten, und der Zwang zu einer konstanten Serververbindung können die Capella-Nutzer unter Umständen in ihrer Entwicklungstätigkeit einschränken. Unter anderem lassen sich Modelle ausschließlich durch das Ändern von Diagrammen oder mit den Kontext-Menüs im *Projekt-Explorer* bearbeiten. Dies hat zur Folge, dass keine neuen oder zusätzliche Realisierungsbeziehung erzeugt werden können, da dies ausschließlich durch das Bearbeiten der *capella*-Datei möglich ist.

Damit keine Konflikte entstehen, sperrt der Server Modellelemente, damit nur ein Nutzer diese bearbeiten kann. Das Sperren von Modellelementen kann in *automatisches* und *manuelles* Sperren unterteilt werden. Sperrt ein Nutzer eine Menge von Modellelementen manuell, so können andere Nutzer diese so lange nicht bearbeiten, bis diese Sperre manuell wieder entfernt wird. Manuelle Sperren können durch das Aufrufen eines Kontext-Menüs durch das Rechtsklicken auf einem Modellelement im Projekt-Explorer oder einer graphischen Darstellung in einem Diagramm angelegt werden. Dabei kann sich der Nutzer entscheiden, ob nur das ausgewählte Modellelement oder auch alle Elemente, welche transitiv im ausgewählten Modellelement enthalten sind, auch Nachfahren genannt, gesperrt werden. Das Sperrverhalten wird im Folgenden mithilfe Abbildung 2.34 dargestellt. Sperrt Nutzer Ye beispielsweise die operationale Entität **Nutzer**, so wird dieses Modellelement für Ye mit einem grünen Vorhängeschloss versehen. Grüne Vorhängeschlösser an Elementen symbolisieren, dass diese Elemente nicht von anderen Nutzern verändert werden können. Ein anderer Nutzer, wie Su, bekommt anstatt grüne Vorhängeschlösser, rote angezeigt, welche symbolisieren, dass diese nicht editiert werden können. Außerdem wird hinter dem Element-Namen im Projekt-Explorer in eckigen Klammern angezeigt, dass Ye die Elemente gesperrt hat. Sperrt allerdings Ye das operationale Fähigkeits-Paket **Operational Capabilities** und alle dessen Nachkommen, so wird nicht nur das Paket gesperrt, sondern auch alle Modellelemente, welche transitiv sich in einer Kompositionen-Beziehung zu dem Paket befinden. In diesem Fall bedeutet dies, dass auch die operationale Fähigkeiten **Medien streamen**, **Medien verwalten** und **Nutzer verwalten** gesperrt werden. Dementsprechend werden genannte Elemente für Ye mit grünen (siehe Abbildung 2.34a) und für Su mit roten (siehe Abbildung 2.34b) Vorhängeschlössern versehen. Wichtig zu beachten ist, dass wenn Modellelemente manuell gesperrt werden, dies nicht bedeutet, dass dessen graphische Repräsentationen in Diagrammen auch gesperrt sind. Genauer gesagt kann Su zwar die graphische Darstellung der operationalen Entität **Nutzer** im OEBD-Diagramm aus Abbildung 2.31 bearbeiten, aber nicht das Modellelement selbst ändern oder löschen. Unter anderem sind Position und Größe der graphischen Darstellung des **Nutzers** manipulierbar, aber nicht dessen Eigenschaften und Beziehungen. Sobald Su allerdings beginnt die graphische Darstellung zu manipulieren, wird das Diagramm gesperrt und andere Nutzer können dieses nicht graphisch bearbeiten, bis Su das Diagramm wieder freigibt. Su ist in dem Fall, dass ein Modellelement manuell gesperrt worden ist, auch nicht in der Lage, Nachkommen des **Nutzers** in einem Diagramm anzulegen oder zu entfernen. Allerdings kann der **Nutzer** und dessen Nachkommen aus einem Diagramm entfernt werden, wenn die vom Diagramm bereitgestellte Sicht auf das System dies vorsieht. Zu beachten ist hierbei, dass nur die graphische Repräsentation aus dem Diagramm und nicht das Modellelement selbst entfernt wird. Auch werden in den Diagrammen, wie im Projekt-Explorer, Vorhängeschlösser auf gesperrten Elementen angezeigt (siehe Abbildung 2.31). Automatische Sperren von Modellelementen werden durch das Bearbeiten des Modells oder von Diagrammen ausgelöst. Vorgenommene Änderungen werden durch das Abspeichern bestätigt, womit automatische Sperren aufgehoben werden. Wird Team for Capella nicht genutzt, so besteht das Abspeichern aus dem Übernehmen von Änderungen in die Projekt-Dateien. Änderungen in Diagrammen werden in die *aird*-Datei und Änderungen am Modell in die *capella*-Datei geschrieben. Mit der Team for Capella Erweiterung werden allerdings Änderungen an den Server geschickt, welcher diese dann in die geteilte Capella-Instanz einpflegt und die anderen Nutzern benachrichtigt, dass Änderungen vorliegen. Änderungen werden in der Regel von Team for Capella automatisch geladen und repräsentiert, allerdings ist es manchmal nötig als Nutzer ein Neuladen von Diagrammen manuell mittels Kontext-Menüs oder Tastaturkurzbefehlen auszulösen.



(a) Projekt-Explorer des Nutzers Ye, nachdem operationale Entität **Nutzer** und das operationale Fähigkeits-Paket **Operational Capabilities** mit allen Nachkommen gesperrt worden sind.

(b) Projekt-Explorer der Nutzerin Su, nachdem Ye Sperren manuell vorgenommen hat.



(c) OEBD aus Abbildung 2.31 mit von Ye gesperrten **Nutzer**.

Abbildung 2.34.: Manuelles Sperren von Elementen und dessen Auswirkungen. Nutzer Ye sperrt Elemente und Nutzer Su kann diese nicht bearbeiten.

Beim Anlegen von automatischen Sperren ist es wichtig zwischen Änderungen, welche das Modell, und Änderungen, welche ausschließlich Diagramme modifizieren, zu unterscheiden. Das Verschieben oder Anordnen von Elementen in einem Diagramm sind Beispiele für Änderungen, welche keine Capella-Modelle modifizieren, sondern lediglich das Diagramm modifizieren. In solchen Fällen sperrt Team for Capella alle graphischen Elemente des Diagramms und hindert andere Nutzer vom Modifizieren des Diagramms. Die anderen Nutzer können allerdings die Modellelemente selbst modifizieren. Dieses Sperrverhalten kann allerdings problematisch sein, da die graphische Repräsentation eines Modellelements in einem Diagramm bearbeitet

werden kann, während ein andere Nutzer das Modellelement selbst aus dem Projekt löscht. Es ist somit den Capella-Nutzern beim Bearbeiten von Diagrammen überlassen, zusätzliche manuelle Sperren vorzunehmen, damit nicht die Modellelemente gelöscht werden, welche gerade in den Diagrammen bearbeitet werden.

Angenommen Su sei momentan am Bearbeiten des OEBDs in Abbildung 2.34c, um es visuell für ein Meeting aufzubereiten. Das Diagramm wird dadurch automatisch gesperrt. Allen anderen Nutzern ist es nicht mehr möglich das Diagramm zu bearbeiten, da alle Schaltflächen in der Palette nicht angezeigt werden, wie in Abbildung 2.35 zu sehen ist. Andere Bearbeitungsmöglichkeiten, welche nicht die Schaltflächen in der Palette benötigen, sind durch die Sperre funktionslos und lediglich das Betrachten des Diagramms ist möglich.

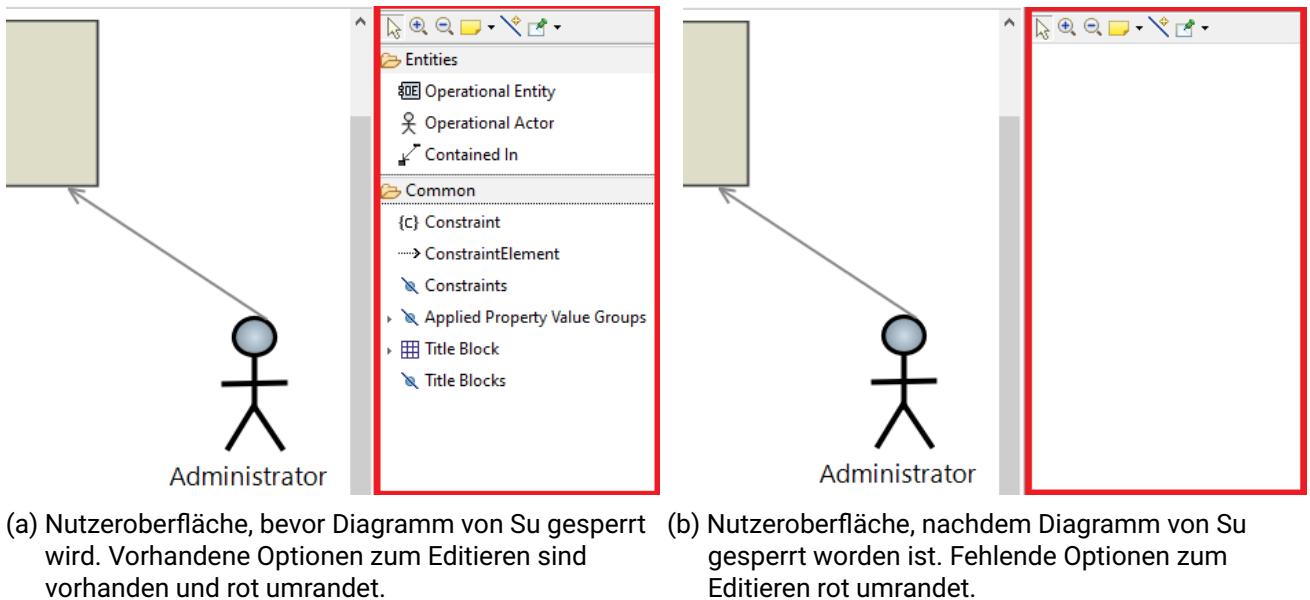


Abbildung 2.35.: Ye's Nutzeroberfläche beim Editieren des Diagramms aus Abbildung 2.31, nachdem Su eine automatische Sperre auslöst.

Es ist Ye nun nicht möglich, das Diagramm zu bearbeiten. Team for Capella erlaubt ihm allerdings das direkte Modifizieren der Modellelemente, welche im Diagramm repräsentiert werden. Mithilfe des Projekt-Explorers löscht Ye nun die operationale Entität **Nutzer** und alle seine Nachkommen. Obwohl Su das Diagramm bearbeitet, werden die Modellelemente gelöscht. Su wird auf das Löschen dieser Elemente durch rote Kreuze neben den graphischen Repräsentationen aufmerksam gemacht. Die graphischen Darstellungen der gelöschten Modellelemente können nicht mehr bearbeitet werden, und das Aufrufen von Informationen über die gelöschten Elemente schlägt fehl, wie in Abbildung 2.36 gezeigt wird. Es ist auch ohne Weiteres nicht möglich, die Änderungen von Ye rückgängig zu machen. Hierfür muss Nutzer Su einen Administrator kontaktieren, welcher mit der im Team for Capella Server integrierten Versionskontrollsoftware das Projekt auf eine Version vor Ye's Änderungen zurücksetzen kann.

Änderungen, welche im Projekt-Explorer vorgenommen werden, manipulieren das Modell direkt und es werden automatische Sperren ausgelöst. Wird ein Element einem anderen hinzugefügt, so werden beide gesperrt. Das Löschen eines Modellelements führt zu dem Sperren aller Modelle, welche von dieser Löschung betroffen sind. Zu den betroffenen Modellelementen zählen das Modellelement selbst und alle dessen Nachfahren. Es wird aber auch das Element, welches das zu löschenende Modellelement enthält, gesperrt. Werden lediglich Referenzen oder Attribute eines Modellelements geändert, so wird nur das Modellelement selbst gesperrt. Es ist zu beachten, dass Team for Capella nur innerhalb eines Systemmodells eines ARCADIA-Schrittes Modellelemente

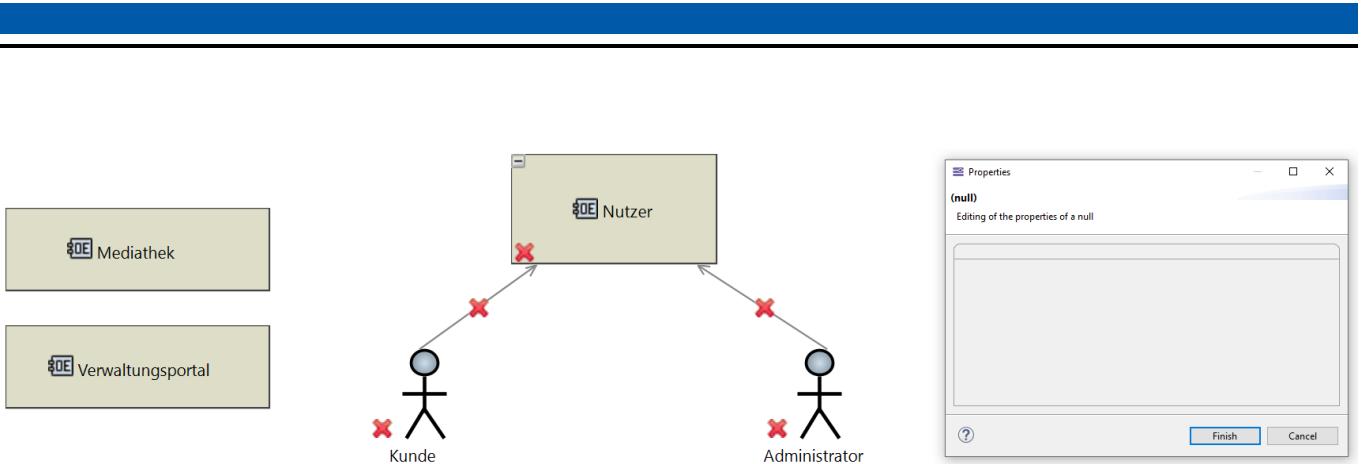


Abbildung 2.36.: Su's Nutzeroberfläche beim Editieren des Diagramms aus Abbildung 2.31, nachdem Ye das Modellelement der operationalen Entität **Nutzer** und dessen Nachkommen gelöscht hat. Gelöschte Elemente sind mit roten Kreuzen versehen. Aufrufen von Informationen über gelöschte Elemente öffnet eine leere Übersicht (Fenster rechts).

sperren kann. Unter anderem werden nicht Realisierungen eines Modellelements im Modell des folgenden ARCADIA-Schrittes gesperrt, wie im Folgenden vorgeführt wird. Su sperrt eine System-Komponente in einem konsistenten Modell, in dem alle Modellelemente jeweils im nachfolgenden Schritt realisiert werden. Diese System-Komponente realisiert mindestens eine operationale Entität im OA-Modell. Auch wird die System-Komponente von mindestens einer logischen Komponente realisiert. Allerdings bleiben die realisierenden logischen Komponenten und die realisierten operationale Entitäten von der Sperre unbeeinflusst. Dieses Sperrverhalten führt beim gleichzeitigen Bearbeiten mehrerer ARCADIA-Schritt-Modellen möglicherweise zu Problemen bei der Verwaltung und Aufrechterhaltung von Verfolgbarkeit von Anforderungen. Dies ist der Fall, da das Modell eines vorherigen ARCADIA-Schrittes Anforderungen an das Modell des nachfolgenden Schrittes stellt. Mit dem Löschen eines Modellelements im Modell des vorherigen ARCADIA-Schrittes wird auch die Realisierungsreferenz von dessen Realisierung auf das gelöschte Modellelement entfernt. Durch den Verlust dieser Referenz wird der Grund, wieso die Realisierung im nachfolgenden ARCADIA-Schritt-Modell überhaupt benötigt wurde, nicht mehr dargestellt. Dieses Problem gilt auch im umgekehrten Fall. Wird das realisierende Modellelement aus einem nachfolgenden ARCADIA-Schritt-Modell gelöscht, welche die einzige Realisierung des realisierten Modellelements im Modell des vorherigen Schrittes ist, so ist das realisierte Modellelement nach ARCADIA kein notwendiges Modellelement. Das realisierte Modellelement besitzt nun keinen Grund zur Existenz im Modell mehr. Löscht Su beispielsweise eine System-Komponente, während Ye die realisierte operationale Entität bearbeitet und mit Informationen anreichert, so ist das Capella-Modell möglicherweise inkonsistent. Die operationale Entität muss durch eine System-Komponente realisiert werden. Existiert keine andere Realisierung als diese, welche Su gelöscht hat, so ist die operationale Entität nach der ARCADIA-Methode überflüssig. Entweder muss die operationale Entität und dessen Nachkommen entfernt werden, oder Ye muss manuell eine neue Referenz von einer System-Komponente auf die operationale Entität anlegen. Allerdings ist das Setzen dieser Referenz bei Verwendung von Team for Capella nicht ohne Weiteres möglich. Realisierungsreferenzen sind in Capella ausschließlich durch das Verwenden der Transitions-Funktion oder durch das manuelle Bearbeiten der *capella*-Datei, und somit des gesamten Capella-Modells, erzeugbar. Das Verwenden der Transitions-Funktion erzeugt jedoch auch neue Komponenten mit den Realisierungsreferenzen. Des Weiteren ist das Bearbeiten der *capella*-Datei bei Verwendung der Team for Capella Erweiterung nicht möglich, da Nutzer keinen Zugriff auf diese haben. Selbst wenn das Wiederherstellen der Nachverfolgbarkeit direkt möglich wäre, ist dies je nach Projektgröße nicht nur schwierig, sondern auch teuer. Das Wiederherstellen wird des Weiteren dadurch verkompliziert, dass keine Benachrichtigung von Team for Capella bei solchen Änderungen angezeigt werden. Somit kann es geschehen, dass das Fehlen von Referenzen zur Nachverfolgbarkeit erst erkannt wird, wenn das gesamte Modell validiert wird.

## 3. Implementierung

---

Wie in den vorherigen Abschnitten erläutert wurde, benötigt das Erstellen und Modellieren einer Systemarchitektur eine Vielzahl an aus verschiedenen Domänen stammenden Personen. Dementsprechend ist es unumgänglich, dass viele Sichten erstellt und in der Regel auch gleichzeitig bearbeitet werden müssen. Capella implementiert die ARCADIA-MBSE-Methode und eignet sich als ein MBSE-Werkzeug besonders zum Planen, Entwickeln und Modellieren von Systemarchitekturen. Capella bietet allerdings keine Möglichkeiten, um mit mehreren Nutzern gemeinsam ein System zu modellieren. Auch das kommerzielle Produkt Team for Capella bietet zum gemeinsamen Bearbeiten von Capella-Projekten keine optimale Lösung an. Zum einen wird eine konstante Serververbindung vorausgesetzt, womit zusätzliche Infrastruktur nötig ist. Zum anderen wird das gemeinsame Bearbeiten eines Projekts durch das Sperren von Modellelementen ermöglicht. Wie in Abschnitt 2.8.2 skizziert wurde, kann dieses Sperren zu Problemen beim simultanen Bearbeiten von Modellen verschiedener ARCADIA-Schritten führen. Dementsprechend ist das Ziel dieser Arbeit herauszufinden, ob das Synchronisieren der Systemmodelle der einzelnen ARCADIA-Schritte in Capella mittels TGG grundlegend möglich ist. Ist dies möglich, so lassen sich Werkzeuge auf Basis der TGG für Capella-Modelle bauen, welche dem Capella-Nutzer beim Modellieren unterstützen. Unterstützung könnte in Form von automatischem Anlegen und Verwalten der Realisierungen von Modellelementen in den Systemmodellen der nachfolgenden ARCADIA-Schritte oder in Form von Konsistenzvalidierung und -wiederherstellung angeboten werden. Wie in Abschnitt 3.1 gezeigt wird, ist die Struktur der verschiedenen ARCADIA-Systemmodelle größtenteils identisch. Daraus folgt, dass die Untersuchung zweier Modelle benachbarter ARCADIA-Schritte ausreichend ist, um das Synchronisieren von Capella-Modellen mithilfe von TGG zu untersuchen. Eine beispielhafte Implementation der TGG wird mit dem State of the Art Graph-Transformations-Werkzeug eMoflon::IBeX<sup>1</sup> umgesetzt. eMoflon::IBeX wurde gewählt, da es mit EMF-basierten Modellen, wie Capella-Modellen, umgehen kann. Als Ursprungs- und Zielmodell der TGG werden hierfür SA- und LA-Systemmodelle gewählt, da der Übergang von SA zu LA mit dem Übergang von einer Bedürfnisanalyse zur Modellierung einer implementierbaren Lösung gleichzusetzen ist. In diesem Kapitel wird auf Metamodelle von Capella eingegangen, um dem Leser ein grundlegendes Verständnis dafür zu vermitteln, wie Capella Informationen repräsentiert und speichert. Infolgedessen wird auch ein Korrespondenzmodell für das SA- und LA-Systemmodell spezifiziert, damit TGG-Regeln zum Generieren von Tripeln bestehend aus SA-, LA-Systemmodellen und einem Korrespondenzmodell definiert werden können. Anschließend werden ausgewählte implementierte TGG-Regeln vereinfacht vorgestellt und erläutert.

### 3.1. Capella Metamodelle

---

Capella verwendet EMF als MOF-Implementation, um die Modelle der einzelnen ARCADIA-Schritte zu spezifizieren. Dementsprechend enthält Capella eine Reihe von EMF-Modellen, welche die Struktur der Capella-Modelle beschreiben und als deren Metamodelle eingesetzt werden. Anforderungen an die Konsistenz der

---

<sup>1</sup><https://emoflon.org/>

Capella-Modelle werden allerdings in diesen Metamodellen nicht abgebildet. Unter anderem wird mit diesen nicht spezifiziert, dass Modellelemente eines ARCADIA-Schrittes im Modell des nachfolgenden ARCADIA-Schrittes mindestens eine Entsprechung besitzen müssen. Zusätzliche Anforderungen an die Modellkonsistenz, welche nicht mit den EMF-Modellen ausgedrückt werden können, werden in Capella als Regeln zur EMF-Modellvalidierung implementiert. Im Folgenden wird ein grober Überblick von der Struktur und den Zusammenhängen zwischen ausgewählten Capella-Modellen vermittelt.

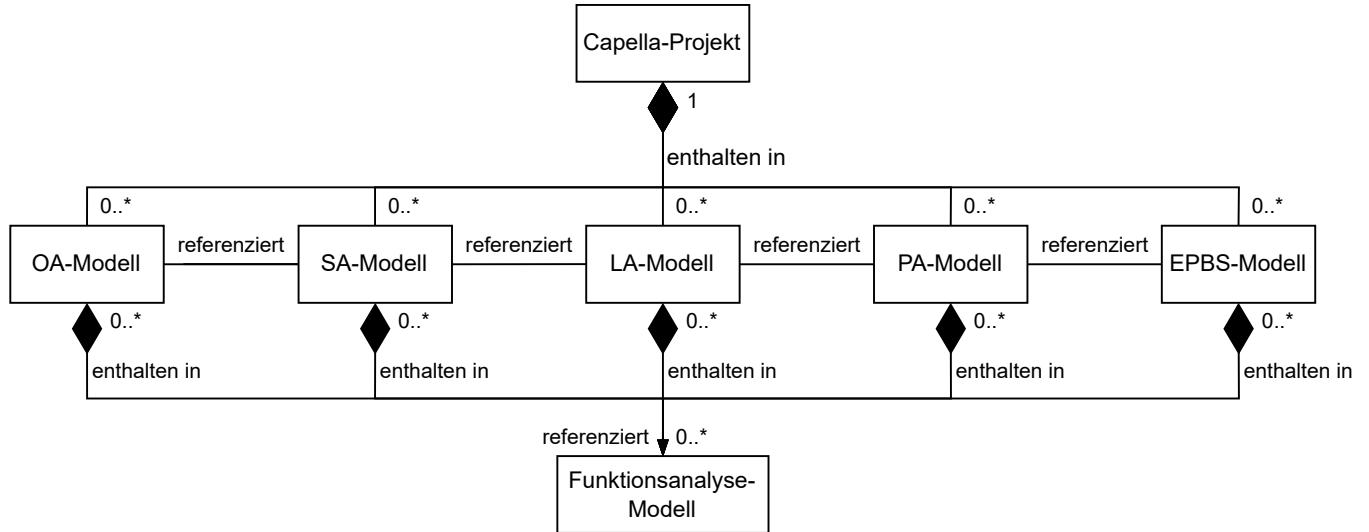
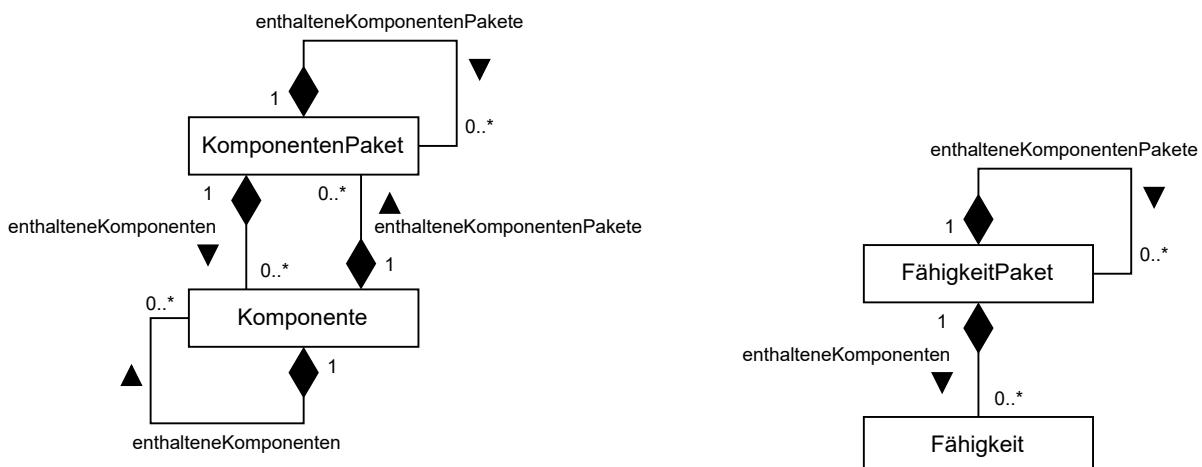


Abbildung 3.1.: Grober Überblick der Capella-Projekt-Struktur.

Capella-Modelle können selbst weitere Capella-Modelle beinhalten. Dies wird in Capella erreicht, indem ein Modellelement ein Capella-Modell repräsentieren und beinhalten kann. Unter anderem ist ein Capella-Projekt selbst ein Modell, dessen Struktur durch das *CapellaModeller*-Metamodell beschrieben und in Abbildung 3.1 dargestellt wird. Capella-Projekte können unbegrenzt viele *OperationalAnalysis*-, *SystemAnalysis*-, *LogicalArchitecture*-, *PhysicalArchitecture*- und *EPBSArchitecture*-Modellelemente beinhalten. Diese Modellelemente repräsentieren und beinhalten die Systemmodelle der entsprechenden ARCADIA-Schritte. Die Struktur dieser Modellelemente sind durch gleichnamige Metamodelle spezifiziert. Das *SystemAnalysis*-Modellelement bildet die alleinige Ausnahme, dessen Metamodell das *ContextArchitecture*-Metamodell ist. Zwar besitzen die OA-, SA-, LA-, PA- und EPBS-Modelle getrennte Metamodelle, aber diese benötigen und referenzieren einander. Dies ist dem Umstand geschuldet, dass jedes Modellelement seine Realisierungen im Systemmodell des nachfolgenden ARCADIA-Schritts direkt referenziert. Dies bedeutet, dass die System-Komponente **Nutzer** als Modellelement der *SystemAnalysis* aus Abbildung 2.25 die operationale Entität **Nutzer** als Modellelement der *OperationalAnalysis* aus Abbildung 2.23 referenziert. Dementsprechend benötigt das *ContextArchitecture*-Metamodell das *OperationalAnalysis*-Metamodell als Kontext, da es Referenzen auf Elemente besitzt, welche gar nicht im *ContextArchitecture*-Metamodell definiert sind. Des Weiteren sieht Capella vor, dass Modellelemente sich bewusst sind, von welchen Elementen sie referenziert werden. Konkret bedeutet dies, dass das Modellelement der operationalen Entität **Nutzer** sich bewusst ist, dass sie von dem Modellelement der System-Komponente **Nutzer** referenziert wird. Daraus folgt, dass auch das *OperationalAnalysis*-Metamodell das *ContextArchitecture*-Metamodell als Kontext benötigt. Dieses gegenseitige Referenzieren von Metamodellen und infolgedessen gegenseitige Referenzieren von Modellelementen, welche verschiedenen Capella-Modellen angehören, führt zu einer starken Verflechtung der Modelle. Dadurch sind Capella-Modelle nicht voneinander trennbar und es ist nicht trivial ein Capella-Modell ohne den Kontext der anderen Modelle eines Capella-Projekts zu betrachten. Dies wird nur dadurch weiter erschwert, dass Modellelemente eines Capella-Modells

nicht nur Modellelemente anderer Capella-Modelle referenzieren können, sondern selbst weitere Modelle repräsentieren und beinhalten können. Beispielsweise beinhaltet das Modell jedes ARCADIA-Schrittes Modellelemente von Funktionsanalyse-Modellen. Wie anhand von Abbildung 2.25 beispielhaft demonstriert wird, sind Funktionsanalysemodule ebenfalls durch eine Vielzahl von stark miteinander verflochtenen Metamodellen spezifiziert. Funktionsanalysemodule werden mit dem *FunctionalAnalysis*-Metamodell spezifiziert. Die in Abbildung 2.25 dargestellten System-Funktionen sind Teil eines solchen Funktionsanalysemodells. Allerdings sind diese System-Funktionen in den System-Komponenten enthalten. Nicht nur sind die System-Funktionen somit transitiv im SA-Systemmodell enthalten, sondern sie werden auch vom SA-Systemmetamodell spezifiziert. Nur die Systemmodelle der einzelnen ARCADIA-Schritte lassen sich bedingt voneinander trennen und einzeln betrachten, indem gegenseitige Referenzen ignoriert werden.

Die Systemmodelle der einzelnen ARCADIA-Schritte sind fast identisch strukturiert. Meist besteht der Unterschied zwischen diesen Modellen rein in der Benennung und der Interpretation der repräsentierten Information. Dementsprechend wird im Folgenden die Struktur dieser Modelle abstrakt erläutert und anschließend die Korrespondenzen zwischen ausgewählten Modellelementen tabellarisch aufgezeigt. In Capella lassen sich Gruppen von Elementen derselben Art zu Paketen zusammenfassen. Die Elemente und Pakete können selbst Elemente und Pakete beinhalten. Wie in Abbildung 3.2a anhand von Komponenten-Modellelementen dargestellt wird, entsteht dadurch eine rekursive Struktur. In der Regel kann ein Element unbegrenzt viele Elemente und Pakete derselben Art beinhalten. Ebenso können Pakete in der Regel ebenfalls unbegrenzt viele Elemente und Pakete derselben Art beinhalten.



(a) Paketstruktur von Komponenten in Capella-Modellen. (b) Paketstruktur von Fähigkeiten in Capella-Modellen.

Abbildung 3.2.: Paketstruktur von Capella-Modellelementen beispielhaft visualisiert. Schwarze Dreiecke geben die Leserichtung der Kompositionsbeziehungen an.

Die Hauptbestandteile der Capella-Systemmodelle der einzelnen ARCADIA-Schritte sind Komponenten, Fähigkeiten und Funktionen. Das Systemmodell eines ARCADIA-Schrittes wird mit einem Systemmodellelement dargestellt, welches jeweils ein Paket für alle enthaltenen Komponenten, Funktionen und Fähigkeiten enthält. Komponenten werden benutzt, um Bestandteile des Systems, mit dem System interagierende Personen oder System-externe Komponenten darzustellen. Funktionen modellieren Aktivitäten oder Aktionen, welche von Komponenten durchgeführt werden können. Meist ist die Ausführung einer Funktion mit einem Austausch von Komponenten oder Informationen verbunden. Komponenten und Funktionen weisen die zuvor erläuterte rekursive Struktur, bestehend aus Elementen und Paketen, vor. Fähigkeit-Modellelemente

stellen Fähigkeiten des Systems oder von Komponenten dar, welche durch Funktionen ermöglicht werden. Es ist zu beachten, dass Fähigkeiten eine eigene Struktur besitzen. Wie in Abbildung 3.2b dargestellt ist, können Fähigkeit-Modellelemente im Gegensatz zu Komponenten oder Funktionen keine weiteren Fähigkeiten oder Fähigkeit-Pakete beinhalten. In Abbildung 3.3 werden diese Zusammenhänge der Struktur eines Capella-Systemmodells zusammenfassend dargestellt.

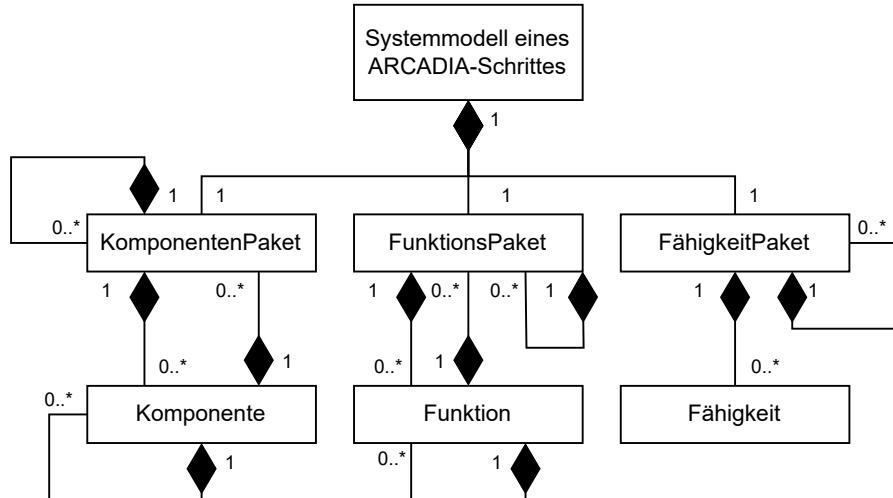


Abbildung 3.3.: Grobe Struktur eines Capella-Systemmodells als Resultat eines ARCADIA-Schrittes.  
Kantenbezeichnungen übersichtshalber ausgeblendet.

Komponenten, Fähigkeiten und Funktionen werden in jedem ARCADIA-Schritt unterschiedlich genannt. Tabelle 3.1 listet die Modellelementbezeichnungen der Systemmodelle, Komponenten, Funktionen und Fähigkeiten zugeordnet zu den verschiedenen ARCADIA-Schritten auf, damit diese Bezeichnungen in den folgenden Abschnitten verwendet werden können. Zu beachten ist, dass das EPBS-Modell anders als die anderen Modelle der ARCADIA-Schritte nicht primär das System modelliert, sondern Anforderungen an die Umsetzung des Systems darstellt. Capella sieht eine gesonderte Betrachtung des EPBS-Modells vor, da es das einzige Systemmodell ist, in welchem nicht alle Modellelemente des vorherigen ARCADIA-Schritt-Systemmodells, dem PA-Systemmodell, repräsentiert werden müssen. So ist es im EPBS-Modell unter anderem nicht möglich, Funktionen darzustellen. Stattdessen werden die Komponenten des PA-Schrittes, auch **PhysicalComponent** genannt, als ein **ConfigurationItem**-Modellelement dargestellt, welches Anforderungen an Komponenten und deren Funktionen repräsentiert.

	Systemmodell	Komponente	Funktion	Fähigkeit
OA	OperationalAnalysis	Entity	OperationalActivity	OperationalCapability
SA	SystemAnalysis	SystemComponent	SystemFunction	Capability
LA	LogicalArchitecture	LogicalComponent	LogicalFunction	CapabilityRealization
PA	PhysicalArchitecture	PhysicalComponent	PhysicalFunction	CapabilityRealization
EPBS	EPBSArchitecture	ConfigurationItem	-	CapabilityRealization

Tabelle 3.1.: Tabellarische Gegenüberstellung von Modellelementbezeichnung und Zuordnung zu einem ARCADIA-Schritt.

Im Folgenden wird ein Korrespondenzmodell spezifiziert, mit welchem in folgenden Abschnitten TGG-Regeln definiert und erläutert werden. Als das Ursprungsmodell wird das **SystemAnalysis**-Systemmodell (im Folgenden auch SA-Systemmodell genannt) gewählt. Das **LogicalArchitecture**-Systemmodell (im Folgenden auch LA-Systemmodell genannt) wird dementsprechend als das Zielmodell gewählt, da es sich um das Systemmodell des nachfolgenden ARCADIA-Schrittes handelt. Wie zuvor erläutert lässt sich die Wahl von SA und LA auf die Bedeutung des Übergangs von SA zu LA zurückführen. Die OA ist eine abstraktere Systemdarstellung, als die SA es ist. In beiden dieser Schritte werden keine implementierbaren Systemarchitekturen modelliert, sondern die Bedürfnisse und Anforderungen an das System. Die Systemmodelle der LA und PA stellen implementierbare Systemarchitektur dar. Somit ist der Übergang von SA zu LA ein Entwerfen und Definieren einer implementierbaren Systemarchitektur, welche die in der OA und SA modellierten Bedürfnisse erfüllen kann. Wie in Abbildung 3.1 abgebildet sind die Systemmodelle der verschiedenen ARCADIA-Schritte stark durch Realisierungsreferenzen miteinander verflochten. Es ist mittels TGGen nicht möglich, Referenzen zwischen Ursprungs- und Zielmodell auszudrücken. Stattdessen können die Realisierungsreferenzen als Entsprechungen im Korrespondenzmodell dargestellt werden. Dementsprechend lassen sich die benötigten Korrespondenzmodellelemente direkt von den Capella-Metamodellen ableiten, wie anhand von Komponenten im Folgenden demonstriert wird.

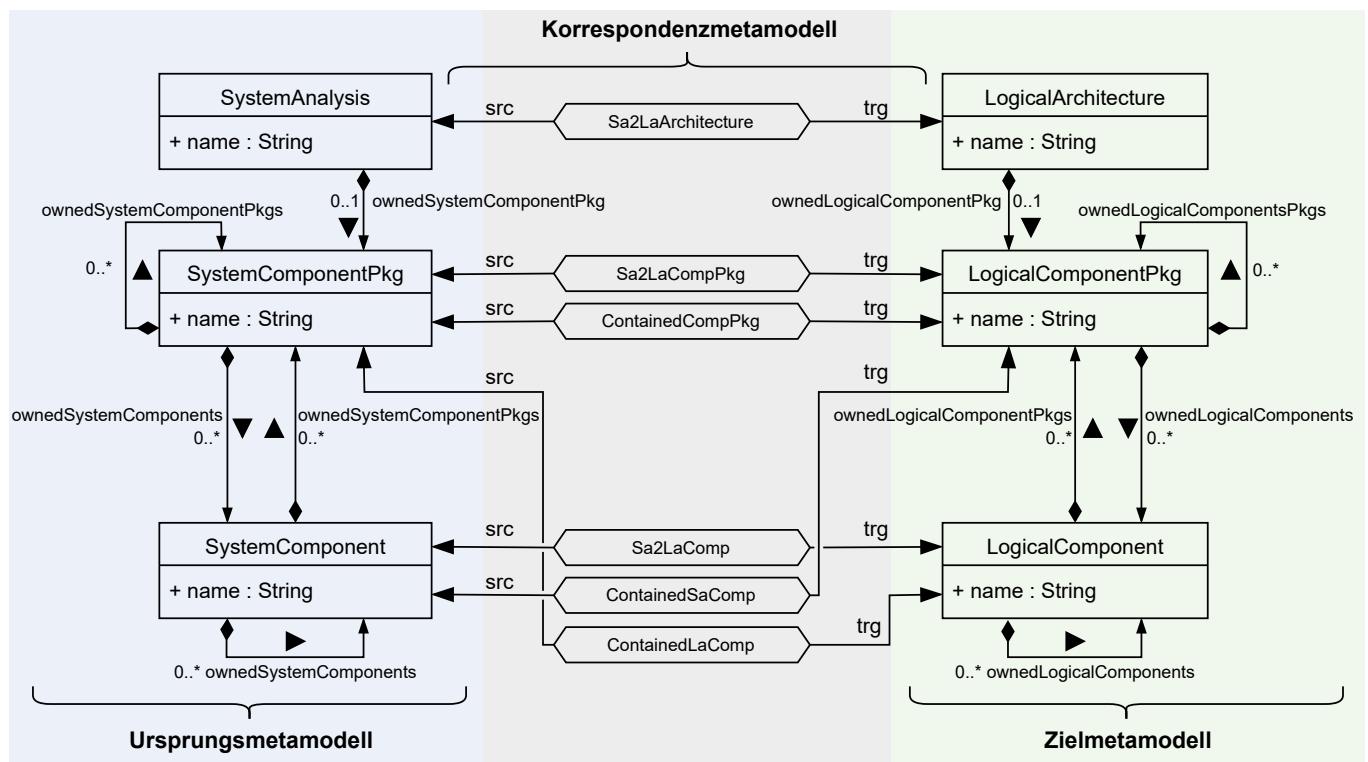


Abbildung 3.4.: Ausschnitt des ContextArchitecture-Metamodells (links), des LogicalArchitecture-Metamodells (rechts) und einem Korrespondenzmodell in der Mitte.

Abbildung 3.4 zeigt einen Ausschnitt des SA-Metamodells auf der linken Seite, einen Ausschnitt des LA-Metamodells auf der rechten Seite und ein Korrespondenzmetamodell in der Mitte. Das Ursprungsmodell besteht aus einem **SystemAnalysis**-Modellelement, welches das Systemmodell der SA repräsentiert. Dieses kann maximal ein **SystemComponentPkg**-Modellelement über die **ownedSystemComponentPkgs**-Referenz beinhalten. Analog zur vorgestellten Struktur eines ARCADIA-Systemmodells in Abbildung 3.3 kann ein **SystemComponentPkg**-Modellelement unbegrenzt viele **SystemComponent**-Modellelemente über die **ow-**

*nedSystemComponents*-Referenz und unbegrenzt viele **SystemComponentPkg**-Modellelemente über die *ownedSystemComponentPkgs*-Referenz beinhalten. Die in **SystemComponentPkg**-Modellelementen enthaltenen **SystemComponent**-Modellelemente können selbst wieder unbegrenzt viele **SystemComponent**- und **SystemComponentPkg**-Modellelemente enthalten. Das Zielmodell ist, wie zuvor erläutert, identisch zum Ursprungsmodell strukturiert. Allerdings tragen das Systemmodell, Komponenten, Komponenten-Pakete und Referenzen an die LA angepasste Namen, wie der Tabelle 3.1 entnommen werden kann. Die Entsprechung zwischen den Systemmodellen wird mit dem **Sa2LaArchitecture**-Korrespondenzmodellelement dargestellt, welches sich entsprechende **SystemAnalysis**- und **LogicalArchitecture**-Modellelemente referenziert. Entsprechungen zwischen **SystemComponentPkg**- und **LogicalComponentPkg**-Modellelementen werden mit der **Sa2LaCompPkg**-Korrespondenz modelliert. Auch dieses Korrespondenzmodellelement referenziert ein Paar von sich entsprechenden **SystemComponentPkg**- und **LogicalComponentPkg**-Modellelementen. Das **Sa2LaComp**-Korrespondenzelement repräsentiert die Entsprechung zwischen **SystemComponent**- und **LogicalComponent**-Modellelementen und verbindet diese durch Referenzen.

Wie in Abbildung 3.1 dargestellt, kann ein Capella-Projekt eine Vielzahl an Systemmodellen für jeden ARCADIA-Schritt beinhalten. Modellelemente eines SA-Systemmodells sollen allerdings nur Realisierungen in einem LA-Systemmodell enthalten, falls das LA-Systemmodell dem SA-Systemmodell entspricht. Damit wird erreicht, dass ein System keine Komponenten enthalten kann, welche für ein anderes System spezifiziert worden sind. Mit eMoflon::IBeX ist es derzeit nicht möglich, in einer TGG-Regel transitive Hüllen auszudrücken. Dementsprechend muss die Information, welches Systemmodell welches Modellelement transitiv enthält, modelliert werden. Das Ursprungs- und Zielmodell sind bereits mit Metamodellen spezifiziert. Diese Metamodelle sollen nicht modifiziert werden und daraus folgt, dass diese Information im Korrespondenzmodell und als eine Entsprechung dargestellt werden muss. Hierfür kann die Struktur der Systemmodelle ausgenutzt werden. Jedes Modellelement im SA-Systemmodell besitzt laut ARCADIA mindestens eine Entsprechung im LA-Systemmodell. Außerdem referenziert jedes Systemmodell im Capella-Projekt genau ein Komponenten-Paket. Das von jedem Systemmodell direkt referenzierte Komponenten-Paket besitzt nur eine Entsprechung im Systemmodell des vorherigen oder nachfolgenden ARCADIA-Schrittes, da alle Systemmodelle genau ein Komponenten-Paket referenzieren. Dieses Komponenten-Paket wird von nun an als das Wurzel-Komponenten-Paket bezeichnet, um auszudrücken, dass dieses alle anderen Komponenten und Komponenten-Pakete des Systemmodells transitiv enthält. Eine Entsprechung zwischen dem enthaltenen Modellelement und dem Wurzel-Komponenten-Paket kann verwendet werden, um die transitive Hülle aller Komponenten und Komponenten-Pakete im Wurzel-Komponenten-Paket darzustellen. Dies wird in Abbildung 3.5 beispielhaft visualisiert.

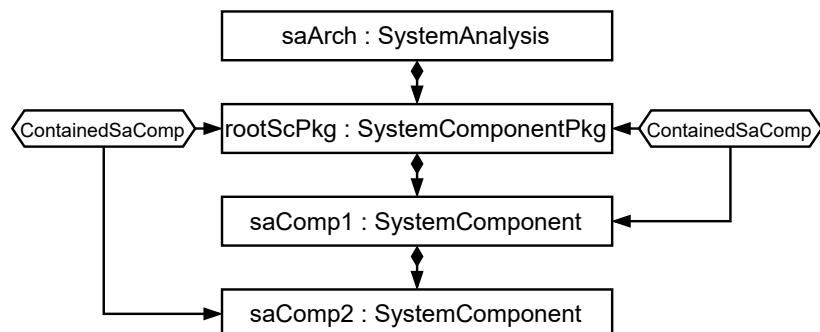


Abbildung 3.5.: Beispielhafte Nutzung einer Entsprechung zum Darstellen der transitiven Hülle, bestehend aus **SystemComponent**- und **SystemComponentPkg**-Modellelementen, welche transitiv im Wurzel-**SystemComponentPkg**-Modellelement enthalten sind.

Um zu überprüfen, ob eine Komponente oder ein Komponenten-Paket in einem Modell enthalten ist, muss lediglich untersucht werden, ob die Komponente oder das Komponenten-Paket dem Wurzel-Komponenten-Paket entspricht. Auf diese Art und Weise ist es nicht nötig, die transitive Hülle auszurechnen. Das Wurzel-Komponenten-Paket *rootScPkg* enthält das **SystemComponent**-Modellelement *saComp1*, welches das **SystemComponent**-Modellelement *saComp2* enthält. Die **ContainedSaComp**-Entsprechung setzt jeweils *saComp1* und *saComp2* mit *rootScPkg* zueinander in Bezug und stellt dar, dass sie in der transitiven Hülle enthalten sind. **ContainedSaComp**-Modellelemente sollen allerdings im Korrespondenzmodell enthalten sein. Die Modellelemente, welche die **ContainedSaComp**-Korrespondenz zueinander in Bezug setzt, sind beides Modellelemente des Ursprungsmodeells. Korrespondenzmodellelemente können aber nur ein Element im Ursprungs- und ein Element im Zielmodell referenzieren und nicht zwei Elemente desselben Modells. Dementsprechend ist das direkte Referenzieren der transitiv enthaltenden Modellelementen und dem Wurzel-Komponenten-Paket, wie in Abbildung 3.5, nicht möglich. Stattdessen kann der in Abbildung 3.6 visualisierte Ansatz gewählt werden.

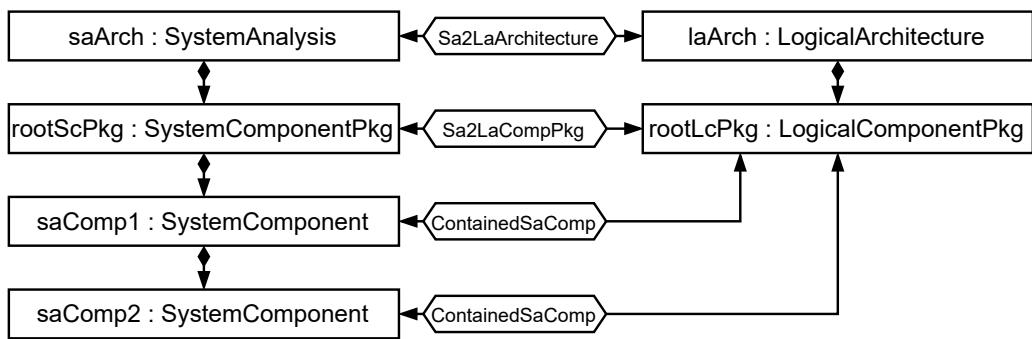


Abbildung 3.6.: Implementierung eines Korrespondenzmodelllements zum Darstellen der transitiven Hülle, bestehend aus **SystemComponent**- und **SystemComponentPkg**-Modelllementen, welche transitiv im Wurzel-**SystemComponentPkg**-Modelllement enthalten sind.

Anstatt *saComp1* und *saComp2* direkt mit *rootScPkg* zueinander in Bezug zusetzen, können diese mit dem Wurzel-**LogicalComponentPkg**-Modelllement *rootLcPkg* im korrespondierendem LA-Systemmodell zueinander in Bezug gesetzt werden. Die Darstellung der transitiven Hülle ist äquivalent zu der vorherigen, da Wurzel-Komponenten-Pakete in Capella-Modellen immer genau eine Entsprechung im Systemmodell des vorherigen und des nachfolgenden ARCADIA-Schrittes besitzen. Dementsprechend wird das Korrespondenzmodelllement **ContainedSaComponent** in Abbildung 3.4 verwendet, um das Wurzel-**LogicalComponentPkg**-Modelllement mit **SystemComponent**-Modelllementen zueinander in Bezug zu setzen. Dass **LogicalComponent**-Modelllemente transitiv im LA-Systemmodell enthalten sind, kann ebenso durch eine Korrespondenz im Korrespondenzmodell abgebildet werden. Hierfür referenziert das **ContainedLaComp**-Korrespondenzmodelllement jeweils das Wurzel-**SystemComponentPkg**-Modelllement und ein **LogicalComponent**-Modelllement. Die **ContainedCompPkg**-Korrespondenz wird zum Darstellen eingesetzt, dass Komponenten-Pakete transitiv im Systemmodell enthalten sind. Diese setzt **SystemComponentPkg**-Modelllemente mit dem Wurzel-**LogicalComponentPkg**-Modelllement oder **LogicalComponentPkg**-Modelllementen mit dem Wurzel-**SystemComponentPkg**-Modelllement zueinander in Bezug.

## 3.2. Erzeugen von Modellelementen

In diesem Abschnitt werden ausgewählte Regeln der in dieser Arbeit implementierten TGG zum Synchronisieren von Komponenten zwischen SA- und LA-Systemmodellen vereinfacht vorgestellt. Die vollständige TGG mit allen implementierten Regeln zum Synchronisieren von Komponenten, Funktionen und Fähigkeiten ist auf GitHub<sup>2</sup> archiviert und kann dort eingesehen werden. Damit die TGG tatsächlich konsistente Tripel beschreibt, muss während der Definition der TGG-Regeln besonderes Augenmerk auf die Anforderungen an die Modellvalidität geworfen werden. Wegen der begrenzten Bearbeitungszeit, Größe und Komplexität der ARCADIA-ML wird im Folgenden nur ein kleiner Satz an Anforderungen an die Modellvalidität definiert. Lediglich diese Anforderungen werden bei der Definition der TGG-Regeln berücksichtigt. Es folgt eine Auflistung und Erläuterung der für diese Arbeit gewählten Anforderungen:

**Anforderung 3.1:** **SystemComponent**-Modellelemente im SA-Systemmodell werden von mindestens einem **LogicalComponent**-Modellelement im korrespondierenden LA-Systemmodell realisiert.

**Anforderung 3.2:** **SystemComponentPkg**-Modellelemente im SA-Systemmodell werden von mindestens einem **LogicalComponentPkg**-Modellelement im korrespondierenden LA-Systemmodell realisiert.

**Anforderung 3.3:** Jedes **LogicalComponent**-Modellelement im LA-Systemmodell realisiert mindestens ein **SystemComponent**-Modellelement im korrespondierenden SA-Systemmodell.

**Anforderung 3.4:** Mindestens eine der Realisierungen im LA-Systemmodell eines **SystemComponent** oder eines **SystemComponentPkg**-Modellelements trägt denselben Namen wie das realisierte Modellelement.

**Anforderung 3.5:** Die Hierarchie, in welche **SystemComponent**- und **SystemComponentPkg**-Modellelemente im SA-Systemmodell gegliedert sind, wird im korrespondierenden LA-Systemmodell übernommen und erweitert.

**Anforderung 3.6:** Nicht jedes **LogicalComponentPkg**-Modellelement im LA-Systemmodell muss ein **SystemComponentPkg**-Modellelement im korrespondierenden SA-Systemmodell realisieren.

Das Ziel des Übergangs von SA zu LA ist das Entwerfen einer abstrakten Systemarchitektur, welche die in der SA modellierten Bedürfnisse decken kann. Dementsprechend ist es zielführend, jede in der SA spezifizierte Komponente in der LA zu realisieren, weswegen die Anforderung 3.1 an die Modellvalidität gestellt wird. Analog zur Anforderung 3.1 muss sichergestellt werden, dass jede in der SA spezifizierte Komponenten-Gruppe ebenfalls in der LA realisiert wird, woraus Anforderung 3.2 folgt. **SystemComponent**-Modellelemente repräsentieren die Anforderungen an die **LogicalComponent**-Modellelemente. Wird ein **LogicalComponent**-Modellelement zum Implementieren des Systems nicht benötigt, so realisiert es kein **SystemComponent**-Modellelemente. Im Umkehrschluss muss jede benötigte Komponente im LA-Systemmodell mindestens einer Anforderung im

<sup>2</sup><https://github.com/adrian-zwenger/CapellaTGG>

SA-Systemmodell genügen, oder genauer gesagt muss jede Komponente im LA-Systemmodell mindestens eine Komponente im SA-Systemmodell realisieren. Dementsprechend wird Anforderung 3.3 gestellt. Capella/ARCADIA stellt keine Anforderungen an die Namensgebung von realisierenden Komponenten. Für diese Arbeit ist entschieden worden, dass der Name eines **SystemComponent**- oder eines **SystemComponentPkg**-Modellelements auch der Name mindestens eines realisierenden **LogicalComponent**- oder eines **LogicalComponentPkg**-Modellelements sein soll. Dementsprechend wird Anforderung 3.4 gestellt. Die rekursive Struktur im SA-Systemmodell, bestehend aus **SystemComponent**- und **SystemComponentPkg**-Modellelementen, lässt sich als hierarchische Gliederung des Systems in Subsysteme und Systemkomponenten interpretieren. Dabei repräsentiert ein **SystemComponent**-Modellelement ein Systemelement und ein **SystemComponentPkg**-Modellelement ein Subsystem des Systems oder eines anderen Subsystems. Dementsprechend ist es sinnvoll, diese Hierarchie im LA-Systemmodell zu übernehmen und zu erweitern. Dies wird mit der Anforderung 3.5 festgehalten. In Capella ist es möglich, **LogicalComponentPkg**-Modellelemente im LA-Systemmodell anzulegen, ohne anzugeben, welches **SystemComponentPkg**-Modellelement im korrespondierenden SA-Systemmodell realisiert wird. Es werden keine Anforderungen der ARCADIA-Methode bezüglich der Modellkonsistenz dadurch verletzt. Damit dies ebenfalls von den TGG-Regeln berücksichtigt wird, wird Anforderung 3.6 gestellt.

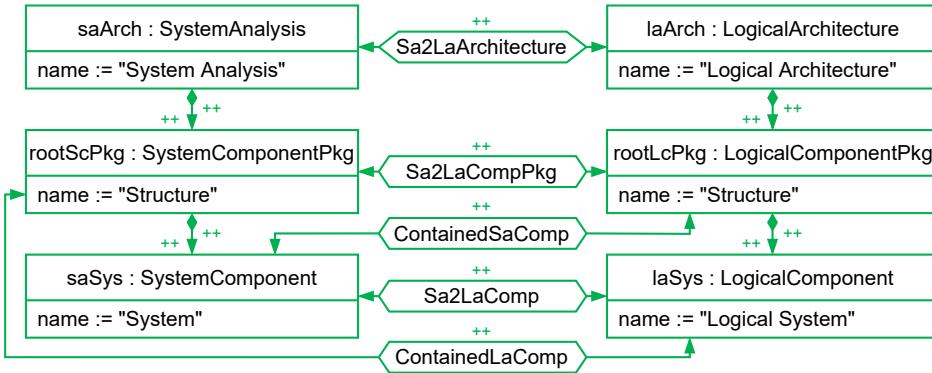


Abbildung 3.7.: TGG-Regel zum Erzeugen der SA- und LA-Systemmodelle.

Analog zu der vorgestellten TGG aus Abschnitt 2.6.2 wird auch im Regelsatz für Capella-Modelle eine axiomatische Regel benötigt, welche die Behälter des Ursprungs- und Zielmodells erzeugt. Hierfür wird die TGG-Regel in Abbildung 3.7 definiert. Zu beachten ist, dass in diesem gesamten Kapitel Kantenbezeichnungen und Instanznamen von Korrespondenzmodellelementen der Einfachheit halber weggelassen werden. Die Regel deklariert das Erzeugen eines **SystemAnalysis**-Modellelements *saArch*, welches ein Systemmodell der SA repräsentiert. **SystemAnalysis**-Modellelemente tragen in Capella standardmäßig den Namen „System Analysis“ und dementsprechend wird das *name*-Attribut gesetzt. *saArch* entspricht dem **LogicalArchitecture**-Modellelement *laArch*, welches das korrespondierende LA-Systemmodell darstellt. LA-Systemmodelle werden standardmäßig „Logical Architecture“ genannt, weswegen auch für *laArch* der *name*-Attributwert entsprechend gesetzt wird. Die Entsprechung zwischen *saArch* und *laArch* wird mit einem **Sa2LaArchitecture**-Korrespondenzelement, welches beide referenziert, dargestellt. In den folgenden Abschnitten werden Regeln für das Erzeugen von Komponenten erläutert. Des Weiteren sind alle Komponenten transitiv im Wurzel-Komponenten-Paket enthalten, von welchem Systemmodelle genau eines enthalten. Dementsprechend ist es sinnvoll, die Wurzel-Komponenten-Pakete in dieser Regel mit anzulegen. Dafür wird das **SystemComponentPkg**-Modellelement *rootScPkg* und dessen korrespondierendes **LogicalComponentPkg**-Modellelement *rootLcPkg* erzeugt. Ein **Sa2LaCompPkg**-Korrespondenzmodellelement stellt die Entsprechung zwischen *rootScPkg* und *rootLcPkg* dar und setzt diese mittels Referenzen in Bezug zueinander. Des Weiteren wird *rootScPkg* als Inhalt von *saArch* und *rootLcPkg* als Inhalt von *laArch* gesetzt. Wurzel-Komponenten-Pakete werden in Capella-Modellen in der

Regel „Structure“ (engl. für Struktur oder Aufbau) genannt. Angesichts dessen wird der Standardwert des *name*-Attributes von *rootScPkg* und *rootLcPkg* dementsprechend gesetzt. Beim Anlegen eines neuen Capella-Projekts wird von Capella in jedem ARCADIA-Schritt-Systemmodell, abgesehen vom OA-Systemmodell, eine das System repräsentierende Komponente angelegt. In der SA wird diese Komponente „System“ und in der LA „Logical System“ (engl. für logisches System) bezeichnet. Dementsprechend wird das **SystemComponent**-Modellelement *saSys* gemeinsam mit dem entsprechenden **LogicalComponent**-Modellelement *laSys* angelegt. Der Wert des *name*-Attributes wird wie zuvor erwähnt gesetzt. Die Entsprechung dieser beiden Systeme wird mit einem **Sa2LaComp**-Modellelement, welches *saSys* und *laSys* referenziert, im Korrespondenzmodell abgebildet. *saSys* und *laSys* sollen direkt in den jeweiligen Wurzel-Komponenten-Paketen enthalten sein, und somit werden sie als Inhalt von *rootScPkg* und *rootLcPkg* gesetzt. Damit im Korrespondenzmodell dargestellt wird, dass *saSys* und *laSys* transitiv in den Systemmodellen enthalten sind, werden die folgenden zusätzlichen Korrespondenzen mit angelegt. Das **ContainedSaComp**-Korrespondenzelement verbindet *saSys* mit *rootLcPkg* durch Referenzen und stellt dar, dass *saSys* transitiv in *saArch* enthalten ist. Analog setzt das **ContainedLaComp**-Korrespondenzelement *laSys* mit *rootScPkg* zueinander in Bezug und stellt dar, dass *laSys* transitiv in *laArch* enthalten ist. Die durch diese Regel erzeugte Struktur ist die minimale Menge an Modellelementen, welche zum Darstellen von Komponenten in den SA- und LA-Systemmodellen benötigt werden.

Capella ist sensiv gegenüber der Speicherreihenfolge von Modellelementen. Zum Beispiel wird die Komponente an erster Stelle im Wurzel-Komponenten-Paket als die Repräsentation des Systems interpretiert, ohne den Kontext der Komponente zu berücksichtigen. Mit TGGGen ist es allerdings derzeit nicht möglich zu deklarieren, an welcher Stelle ein Modellelement in einer sortierten Datenstruktur abzulegen ist. Dementsprechend kann nicht garantiert werden, dass *saSys* und *laSys* an der korrekten Position im Modell gespeichert sind. In Capella werden die Komponenten, welche das System darstellen, wie jede andere Komponenten behandelt. Somit stellt die Einschränkung keine Speicherreihenfolgen deklarieren zu können kein Problem dar. Es muss lediglich nach Zusammenfügen der Capella-Systemmodelle zu einem Capella-Projekt die Position von *saSys* und *laSys* korrigiert werden.

### 3.2.1. Komponenten mit einer Realisierung

Nach Anforderungen 3.1 und 3.2 soll es möglich sein, dass **SystemComponent**- und **SystemComponentPkg**-Modellelemente mehr als eine Entsprechung im korrespondierenden LA-Systemmodell besitzen. Des Weiteren verlangt Anforderung 3.5, dass die Hierarchie bestehend aus **SystemComponent**- und **SystemComponentPkg**-Modellelementen im LA-Systemmodell mit entsprechenden **LogicalComponent**- und **LogicalComponentPkg**-Modellelementen realisiert wird. Zum Implementieren dieser Anforderungen bietet es sich an, zwischen zwei verschiedenen Regel-Gruppen zu unterscheiden. Die erste Gruppe besteht aus Regeln, welche das konsistente simultane Generieren einer Komponenten- und Komponenten-Paket-Hierarchie deklarieren. Die zweite Gruppe besteht aus Regeln, welche zusätzliche Realisierungen im LA-Systemmodell erzeugen, ohne dass weitere SA-Systemmodellelemente angelegt werden. In diesem Unterabschnitt wird die erste TGG-Regel-Gruppe erläutert, während Abschnitt 3.2.2 sich mit der zweiten Gruppe beschäftigt. Anforderung 3.4 ist mit Attributbedingungen, welche die Gleichheit der *name*-Attributwerte der zu erzeugenden Modellelemente sicherstellen, implementierbar. Anforderung 3.5 lässt sich durch das gleichmäßige Anlegen von sich entsprechenden Modellelementpaaren implementieren. Dabei muss lediglich sichergestellt werden, dass das neu anzulegende Modellelementpaar in sich entsprechenden Modellelementen erzeugt wird. Wird dies wiederholt, so entsteht im Ursprungs- und im Zielmodell dieselbe Hierarchie. Komponenten oder Komponenten-Pakete können direkt im Wurzel-Komponenten-Paket, transitiv in einer Komponente oder transitiv in einem Komponenten-Paket enthalten sein. Daher sind hierbei insgesamt sechs Fälle, drei Fälle für das Erzeugen von Komponenten und ebenso drei für Komponenten-Pakete, zu beachten. Die Regeln für

das Erzeugen von Komponenten-Paketen unterscheiden sich von den TGG-Regeln für Komponenten lediglich darin, dass Komponenten-Pakete und nicht Komponenten angelegt werden. Daher werden im Folgenden nur die TGG-Regeln zum Erzeugen der Komponenten-Hierarchie erläutert, während nicht auf die Regeln zum Erzeugen der Komponenten-Paket-Hierarchie eingegangen wird. Die vollständige Regel-Gruppe zum Erzeugen beider Hierarchien ist in Anhang A.1 abgebildet.

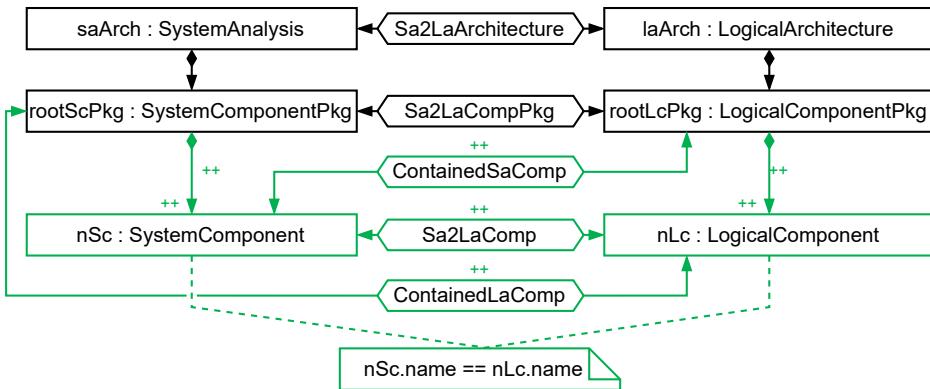


Abbildung 3.8.: TGG-Regel zum Erzeugen von Komponenten im Wurzel-Komponenten-Paket.

Das Erzeugen von **SystemComponent**-Modellelementen im SA-Systemmodell und deren korrespondierenden **LogicalComponent**-Modellelementen im LA-Systemmodell setzt die Existenz beider Systemmodelle voraus. Dementsprechend ist der Kontext für alle Regeln in diesem Unterabschnitt größtenteils identisch und wird im Folgenden exemplarisch anhand von Abbildung 3.8 erläutert. Das **SystemAnalysis**-Modellelement *saArch* repräsentiert das SA-Systemmodell als ein Capella-Modellelement und ist Teil des Kontexts. Dieses Systemmodell entspricht einem LA-Systemmodell, welches mit dem **LogicalArchitecture**-Modellelement *laArch* repräsentiert und ebenfalls als Kontext vorausgesetzt wird. Die Entsprechung zwischen *saArch* und *laArch* wird in Form eines **Sa2LaArchitecture**-Korrespondenzelements, welches beide referenziert, im Kontext vorausgesetzt. Es werden auch das Wurzel-**SystemComponentPkg**-Modellelement *rootScPkg* und das entsprechende Wurzel-**LogicalComponentPkg**-Modellelement *rootLcPkg*, miteinander verbunden durch ein **Sa2LaCompPkg**-Korrespondenzelement, als Kontext vorausgesetzt. *rootScPkg* ist in *saArch* enthalten. Ebenso ist *rootLcPkg* in *laArch* enthalten. Die TGG-Regel in Abbildung 3.8 deklariert das Erzeugen von Komponenten im Wurzel-Komponenten-Paket. Dabei wird das **SystemComponent**-Modellelement *nSc*, das **LogicalComponent**-Modellelement *nLc* und ein **Sa2LaComp**-Korrespondenzelement, welches *nSc* und *nLc* referenziert, neu erzeugt. *nSc* wird in *rootScPkg* und *nLc* wird in *rootLcPkg* angelegt. Eine Attributbedingung stellt sicher, dass der *name*-Attributwert von *nSc* und *nLc* identisch sind. Damit im Korrespondenzmodell abgebildet wird, dass *nSc* transitiv in *saArch* und *nLc* transitiv in *laArch* enthalten sind, werden die in Abschnitt 3.1 beschriebenen zusätzlichen Korrespondenzen erzeugt. Dafür wird eine **ContainedSaComp**-Korrespondenz angelegt, welche *nSc* und *rootLcPkg* durch Referenzen miteinander verbindet. Ebenso werden *nLc* und *rootScPkg* mit einer **ContainedLaComp**-Korrespondenz in Bezug zueinander gesetzt.

Die Regel, abgebildet in Abbildung 3.9, deklariert das Erzeugen einer Komponente, welche in einer anderen Komponente enthalten ist. In dieser Regel werden, wie in Abbildung 3.8, die Existenz von sich entsprechenden **SystemAnalysis**- und **LogicalArchitecture**-Modellelementen mit deren zugehörigen Wurzel-**SystemComponentPkg**- und Wurzel-**LogicalComponentPkg**-Modellelementen vorausgesetzt. Da dieses Mal die neu zu erzeugenden Elemente Inhalt von bereits existierenden Komponenten sein sollen, wird vorausgesetzt, dass ein **SystemComponent**-Modellelement *sc* und das entsprechende **LogicalComponent**-Modellelement *lc*, welche transitiv in *saArch* und *laArch* enthalten sind, existieren. Dadurch, dass im Kontext nicht deklariert ist, in

welchen Modellelementen  $sc$  und  $lc$  enthalten sind, beschreibt der Kontext jegliche  $sc$  und  $lc$  Paare in den korrespondierenden Systemmodellen. Analog zur vorherigen Regel werden das **SystemComponent**-Modellelement  $nSc$ , das korrespondierende **LogicalComponent**-Modellelement  $nLc$  und Korrespondenzmodellelemente erzeugt. Allerdings wird  $nSc$  nicht als Inhalt des Wurzel-**SystemComponentPkg**-Modellelements gesetzt, sondern als Inhalt von  $sc$ . Dementsprechend wird  $nLc$  als Inhalt von  $lc$  gesetzt.

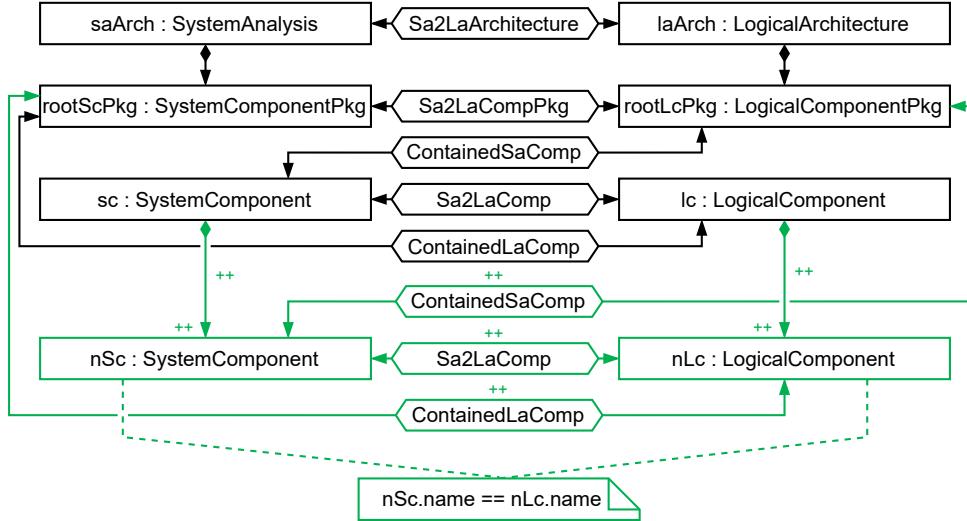


Abbildung 3.9.: TGG-Regel zum Erzeugen von Komponenten in Komponenten.

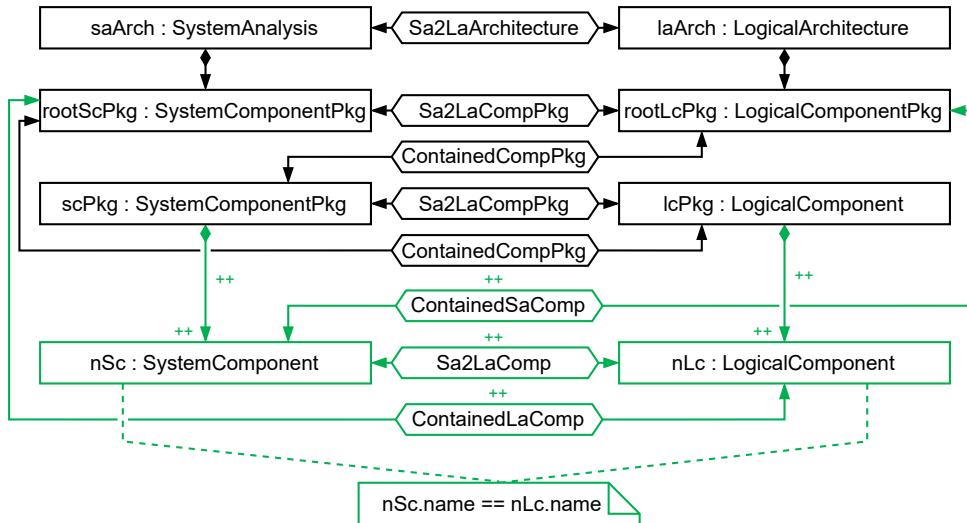


Abbildung 3.10.: TGG-Regel zum Erzeugen von Komponenten in Komponenten-Paketen.

Das Erzeugen von Komponenten, welche transitiv in Komponenten-Paketen enthalten sind, wird analog zur Regel in Abbildung 3.9 erreicht. Wie zuvor deklariert die TGG-Regel in Abbildung 3.10 das Erzeugen eines **SystemComponent**-Modellelements  $nSc$  und des korrespondierenden **LogicalComponent**-Modellelements  $nLc$ . Anstatt dass  $nSc$  und  $nLc$  in sich entsprechenden Komponenten erzeugt werden, werden diese in sich entsprechenden Komponenten-Paketen erzeugt.  $nSc$  wird nun im **SystemComponentPkg**-Modellelement  $scPkg$  und  $nLc$  im **LogicalComponentPkg**-Modellelement  $lcPkg$  erzeugt.

### 3.2.2. Komponenten mit mehreren Realisierungen

Zum Erzeugen von Komponenten und Komponenten-Paketen im SA-Systemmodell mit einer Realisierung im LA-Systemmodell mussten insgesamt sechs Regeln implementiert werden. Das Anlegen von zusätzlichen Realisierungen für SA-Systemmodellelemente im LA-Systemmodell benötigte allerdings insgesamt 16 TGG-Regeln. Wie im Folgenden beispielhaft skizziert wird, ist dies dem Umstand geschuldet, dass es viele verschiedene Einordnungsmöglichkeiten in die Komponenten- und Komponenten-Paket-Hierarchie beim Erzeugen der zusätzlichen Realisierung geben kann. Das **SystemComponent**-Modellelement *sc* wird im LA-Systemmodell von dem **LogicalComponent**-Modellelement *lc* realisiert. Es soll nun zusätzlich das **LogicalComponent**-Modellelement *nLc* erzeugt werden, welches ebenfalls *sc* realisiert. Das **LogicalComponent**-Modellelement *nLc* kann nun als Inhalt von *lc* angelegt werden, aber es kann auch in derselben Hierarchieebene erzeugt und somit demselben Modellelement angefügt werden, in welchem *lc* ebenfalls enthalten ist. Dementsprechend existieren die folgenden Möglichkeiten, wo *nLc* erzeugt werden kann:

1. Das **LogicalComponent**-Modellelement *nLc* wird im **LogicalComponent**-Modellelement *lc* erzeugt.
2. Das **LogicalComponent**-Modellelement *nLc* wird auf derselben Hierarchieebene erzeugt und ist somit gemeinsam mit *lc* Inhalt...
  - a) des Wurzel-**LogicalComponentPkg**-Modellelements.
  - b) eines transitiv im Systemmodell enthaltenem **LogicalComponent**-Modellelements.
  - c) eines transitiv im Systemmodell enthaltenem **LogicalComponentPkg**-Modellelements.

Möglichkeit 2c muss erneut in zwei Fälle unterteilt werden. Zum einen kann das **LogicalComponentPkg**-Modellelement eine Realisierung eines **SystemComponentPkg**-Modellelements sein. Wie der Anforderung 3.6 entnommen werden kann, ist es zum anderen möglich, dass das **LogicalComponentPkg**-Modellelement kein **SystemComponentPkg**-Modellelement realisiert. Im Folgenden wird diese TGG-Regel-Gruppe zum Erzeugen von zusätzlichen Komponenten-Realisierungen erläutert. Auch wird eine Regel zum Erzeugen von **LogicalComponentPkg**-Modellelementen, welche kein **SystemComponentPkg**-Modellelement realisieren, in **LogicalComponent**-Modellelementen vorgestellt. Die restlichen TGG-Regeln sind dem Anhang A.2 zu entnehmen.

Eine Regel, für den in Möglichkeit 1 beschriebenen Fall, lässt sich wie folgt implementieren und ist in Abbildung 3.11 dargestellt. Ein **SystemAnalysis**-Modelllement *saArch*, ein korrespondierendes **LogicalArchitecture**-Modelllement *laArch*, zueinander in Bezug gesetzt durch ein **Sa2LaArchitecture**-Korrespondenzmodellelement, sind erneut Teil des Kontexts. Auch werden die in diesen Modellelementen enthaltenen Wurzel-Komponenten-Pakete *rootScPkg* und das korrespondierende *rootLcPkg* vorausgesetzt. Die Entsprechung zwischen diesem **SystemComponentPkg**- und dem **LogicalComponentPkg**-Modellelement wird mit einem **Sa2LaCompPkg**-Korrespondenzmodellelement im Kontext vorausgesetzt. Es soll mit dieser Regel das Anlegen einer zusätzlichen Realisierung erreicht werden. Dementsprechend sind das **SystemComponent**-Modellelement *sc* und dessen Realisierung, das **LogicalComponent**-Modellelement *lc*, Teil des Kontexts. Ein **Sa2LaComp**-Korrespondenzmodellelement stellt die Entsprechung zwischen *sc* und *lc* dar. Auch sind jeweils ein **ContainedSaComp**- und **ContainedLaComp**-Korrespondenzelement Teil des Kontexts, welche im Korrespondenzmodell abbilden, dass *sc* und *lc* transitiv in den Systemmodellen enthalten sind. Die neue Realisierung, das **LogicalComponent**-Modellelement *nLc*, wird als Inhalt von *lc* erzeugt. Die Entsprechung zwischen der neuen Realisierung *nLc* und *sc* wird mit dem neuen **Sa2LaComp**-Korrespondenzelement dargestellt. Damit im Korrespondenzmodell dargestellt wird, dass *nLc* transitiv im LA-Systemmodell enthalten ist, wird ein zusätzliches

**ContainedLaComp**-Korrespondenzmodellelement angelegt. Dafür setzt das Korrespondenzmodellelement *nLc* mit *rootScPkg* in Bezug zueinander, indem es diese referenziert.

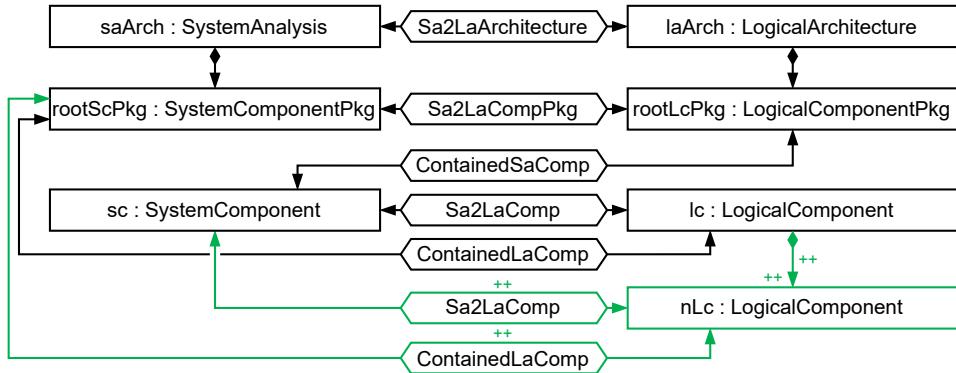


Abbildung 3.11.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung in einer existierenden Realisierung.

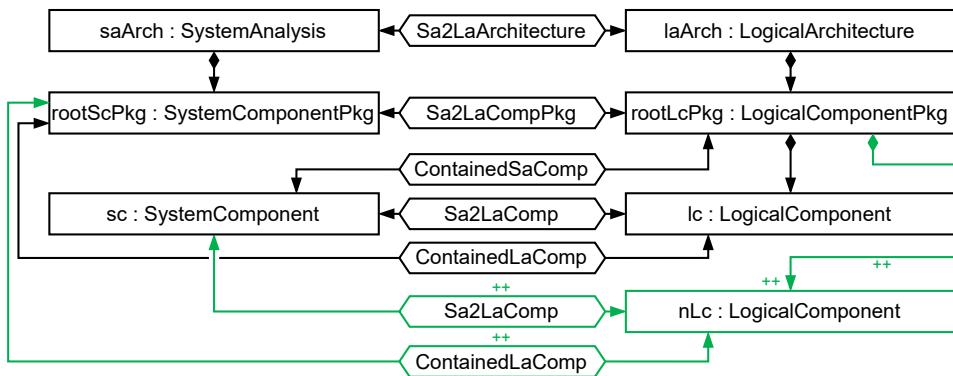


Abbildung 3.12.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung im Wurzel-**LogicalComponentPkg**-Modellelement enthalten ist.

Eine TGG-Regel für den in Möglichkeit 2a beschriebenen Fall lässt sich durch leichtes Modifizieren der soeben beschriebenen Regel implementieren. Das Ergebnis ist in Abbildung 3.12 abgebildet. In dieser TGG-Regel wird *nLc* nicht *lc* als Inhalt angefügt, sondern als Inhalt von *rootLcPkg*. Ebenfalls wird im Kontext vorausgesetzt, dass *lc* ebenfalls in *rootLcPkg* enthalten ist. Durch diese zwei Änderungen wird erreicht, dass die TGG-Regel das Erzeugen einer neuen Realisierung im Wurzel-**LogicalComponentPkg**-Modellelement deklariert, welche dasselbe **SystemComponent**-Modellelement realisiert, wie ein anderes **LogicalComponent**-Modellelement im Wurzel-**LogicalComponentPkg**-Modellelement.

Eine Regel zum Abdecken der Möglichkeit 2b kann von der TGG-Regel in Abbildung 3.12 wie folgt abgeleitet werden und ist in Abbildung 3.13 abgebildet. Das **LogicalComponent**-Modellelement *nLc* wird in dem **LogicalComponent**-Modellelement erzeugt, in welchem *lc* ebenfalls enthalten ist. Dabei realisiert *nLc* das **SystemComponent**-Modellelement *sc*, welches *lc* ebenfalls realisiert. Dementsprechend wird eine neue **Sa2LaComp**-Korrespondenz angelegt, welche *nLc* und *sc* referenziert. Das **SystemComponent**-Modellelement *saContain* und dessen realisierendes **LogicalComponent**-Modellelement *laContain* werden im Kontext vorausgesetzt. Korrespondenzelemente setzen *saContain* und *laContain* zueinander in Bezug und bilden im

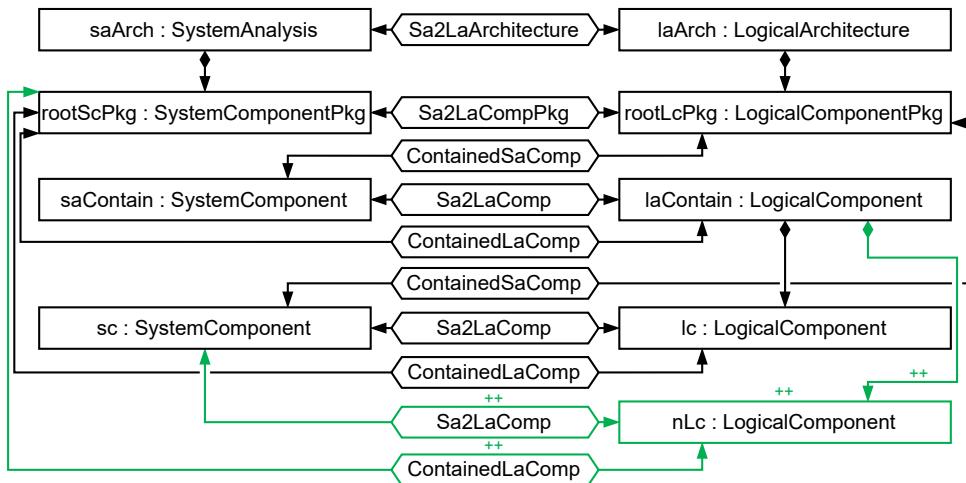


Abbildung 3.13.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponent**-Modellelement enthalten ist.

Korrespondenzmodell ab, dass beide transiv jeweils in *saArch* und *laArch* enthalten sind. Es wird geprüft, dass *lc* Inhalt von *laContain* ist. Ebenfalls wird *nLc* als Inhalt von *laContain* gesetzt.

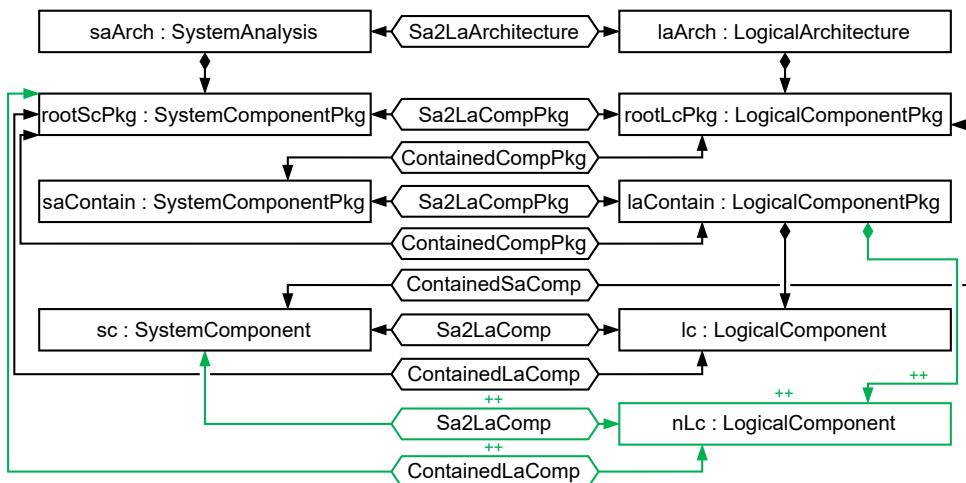


Abbildung 3.14.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponentPkg**-Modellelement enthalten ist.

Eine TGG-Regel für den Fall aus Möglichkeit 2c kann implementiert werden, indem in Abbildung 3.13 der Modellelementtyp von *saContain* zu **SystemComponentPkg** und von *laContain* zu **LogicalComponentPkg** geändert wird. Nach Anpassen der Korrespondenzelementtypen ergibt sich die TGG-Regel in Abbildung 3.14, welche das Erzeugen einer zusätzlichen Realisierung in einem **LogicalComponentPkg**-Modellelement, worin bereits eine Realisierung enthalten ist, deklariert.

Die vorherige TGG-Regel deckt nur den Fall ab, dass das **LogicalComponentPkg**-Modellelement *laContain* ein **SystemComponentPkg**-Modellelement *saContain* realisiert. Allerdings soll es auch möglich sein, zusätzliche Realisierungen in **LogicalComponentPkg**-Modellelementen zu erzeugen, welche kein **SystemComponentPkg**-

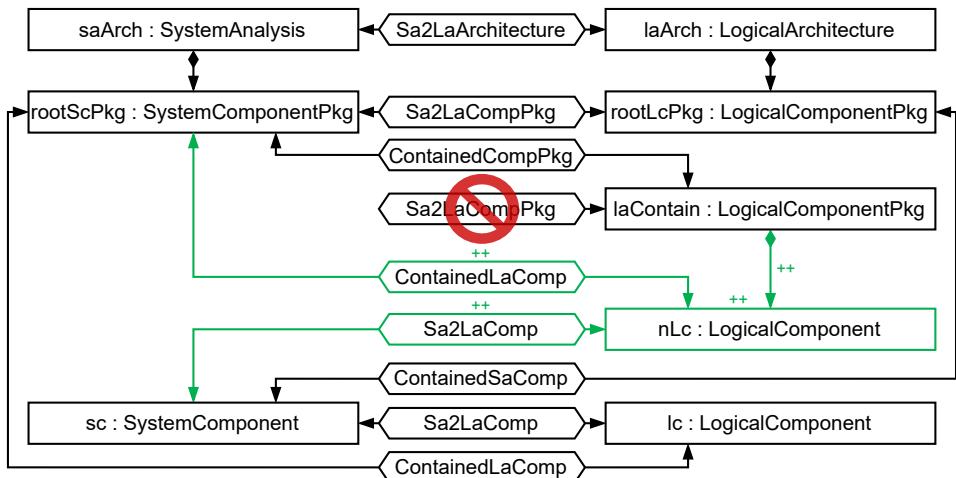


Abbildung 3.15.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung in einem **LogicalComponentPkg**-Modellelement, welches keinem **SystemComponentPkg**-Modellelement entspricht.

Modellelement realisieren. Dafür wird eine TGG-Regel, wie jene aus Abbildung 3.15, benötigt. Mit einer TGG-Regel zu beschreiben, welchem **SystemComponent**-Modellelement das zu erzeugende **LogicalComponent**-Modellelement sinngemäß entsprechen soll, ist nicht trivial. Dementsprechend ist die Regel so deklariert worden, damit alle Konstellation bestehend aus sich entsprechenden Komponenten und einem Komponenten-Paket abgedeckt werden können. Allerdings ist es möglich, dass eine Vielzahl dieser Konstellationen im Kontext der Systemarchitektur-Modellierung nicht sinnvoll sind. Deswegen muss ein Nutzer die Ausführung dieser Regel überwachen und eine Regelanwendungsmöglichkeit und gegebenenfalls manuell auswählen. Der Kontext dieser Regel besteht ebenfalls aus **saArch**, welches das Wurzel-**SystemComponentPkg**-Modellelement **rootScPkg** enthält, und **laArch**, welches das Wurzel-**LogicalComponentPkg**-Modellelement **rootLcPkg** enthält. Auch ist das zu realisierende **SystemComponent**-Modellelement **sc** und dessen existierende Realisierung **lc** Teil des Kontexts. Es wird ebenfalls mit den zusätzlichen **ContainedSaComp**- und **ContainedLaComp**-Korrespondenzmodellelementen im Kontext sichergestellt, dass **sc** transitiv in **saArch** und **lc** transitiv in **laArch** enthalten ist. Dabei ist zu beachten, dass nicht deklariert wird, wo sich **sc** und **lc** genau in der Komponenten- und Komponenten-Paket-Hierarchie im Kontext befinden. Es wird nur deren Existenz vorausgesetzt. Des Weiteren ist das **LogicalComponentPkg**-Modellelement **laContain** Teil des Kontexts. Hierbei wird allerdings mit einer Negative Application Condition (NAC) geprüft, dass kein **Sa2LaCompPkg**-Korrespondenzelement existiert, welches **laContain** referenziert. Dadurch wird sichergestellt, dass **laContain** kein **SystemComponentPkg**-Modellelement realisiert. Diese NAC ist in Abbildung 3.15 mit einem roten Verbots-Symbol über dem **Sa2LaCompPkg**-Korrespondenzelement dargestellt. Nach dieser Regelanwendung ist das neue **LogicalComponent**-Modellelement **nLc** Inhalt von **laContain**. Ein **ContainedLaComp**-Korrespondenzelement setzt **nLc** mit **rootScPkg** in Bezug zueinander. Die Entsprechung zwischen **sc**, dem realisierten Modellelement, und **nLc**, der neuen Realisierung, wird mit dem neu angelegten **Sa2LaComp**-Korrespondenzmodellelement im Korrespondenzmodell dargestellt.

**LogicalComponentPkg**-Modellelemente, welche kein **SystemComponentPkg**-Modellelement realisieren, können entweder im Wurzel-**LogicalComponentPkg**-, in einem **LogicalComponent**- oder in einem transitiv enthaltenem **LogicalComponentPkg**-Modellelement enthalten sein. Sind sie in einem transitiv enthaltenem **LogicalComponentPkg**-Modellelement enthalten, kann erneut zwischen Komponenten-Paketen jeweils mit oder ohne Entsprechung im SA-Systemmodell unterschieden werden. Dementsprechend werden ins-

gesamt vier weitere Regeln benötigt. Diese TGG-Regeln unterscheiden sich allerdings wenig voneinander, weswegen im folgenden nur die Regel vorgestellt wird, welche das Anlegen von **LogicalComponentPkg**-Modellelementen ohne Entsprechung im Wurzel-**LogicalComponentPkg**-Modellelement deklariert. Diese Regel ist in Abbildung 3.16 dargestellt. Die restlichen TGG-Regeln sind dem Anhang A.2 zu entnehmen. *saArch*, welches das Wurzel-**SystemComponentPkg**-Modellelement *rootScPkg*, und *laArch*, welches das Wurzel-**LogicalComponentPkg**-Modellelement *rootLcPkg* enthält, werden im Kontext vorausgesetzt. Das **LogicalComponentPkg**-Modellelement *nLcPkg* wird in *rootLcPkg* erzeugt. Eine **ContainedCompPkg**-Korrespondenz setzt *nLcPkg* und *rootScPkg* zueinander in Bezug und stellt dar, dass *nLcPkg* in *laArch* transitiv enthalten ist. Die Anwendung dieser Regeln zum Erzeugen von **LogicalComponentPkg**-Modellelementen, welche kein **SystemComponentPkg**-Modellelement realisieren, muss ebenfalls von einem Nutzer überwacht werden. Durch das Überwachen wird ermöglicht, dass nur dann ein **LogicalComponentPkg**-Modellelement ohne Entsprechung im SA-Systemmodell erzeugt wird, wenn der Nutzer es für sinnvoll hält. Dementsprechend muss während der Modellsynchronisierung für jedes zu übersetzende **LogicalComponentPkg**-Modellelement entschieden werden, wie mit diesem Element umgegangen und somit welche TGG-Regel ausgeführt werden soll. In anderen Worten muss entschieden werden, ob eine Entsprechung im SA-Systemmodell vorhanden sein soll oder nicht.

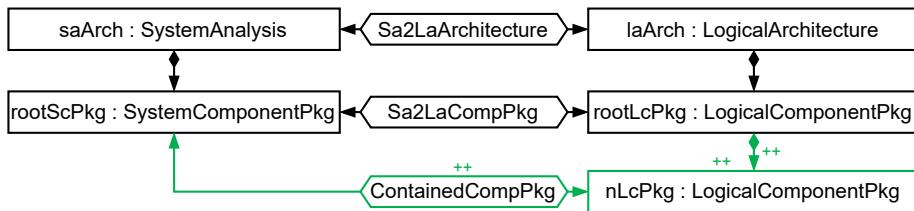


Abbildung 3.16.: TGG-Regel zum Erzeugen eines **LogicalComponentPkg**-Modellelements, welches keinem **SystemComponentPkg**-Modellelement entspricht, im Wurzel-**LogicalComponentPkg**-Modellelement.

### 3.2.3. Erzeugen von Realisierungsreferenzen

Die vorherigen Regelsätze deklarieren das simultane Anlegen von **SystemComponent**-Modellelementen mit ihren realisierenden **LogicalComponent**-Modellelementen und das Erzeugen zusätzlicher **LogicalComponent**-Modellelementen, welche bereits vorhandene **SystemComponent**-Modellelemente realisieren. Diese Regeln erfüllen Anforderungen 3.1 und 3.2, da jedes **SystemComponent**- und jedes **SystemComponentPkg**-Modellelement in einem konsistenten Tripel mindestens einmal realisiert wird. Dabei genügen die TGG-Regeln aus Abschnitt 3.2.1 den Anforderungen 3.4 und 3.5, da mit diesen Regeln die Komponenten-Hierarchie zwischen SA- und LA-Systemmodell synchronisiert werden kann. Des Weiteren wird ebenfalls sicherstellt, dass einer der Realisierungen eines **SystemComponent**-Modellelements den Namen des **SystemComponent**-Modellelements trägt. Auch Anforderung 3.6 ist erfüllt, da mit Regeln wie jener aus Abbildung 3.16 **LogicalComponentPkg**-Modellelemente erzeugt werden können, welche kein **SystemComponentPkg**-Modellelement realisieren. Allerdings können die bisherigen TGG-Regeln nicht der Anforderung 3.3 genügen, da es mit den bisherigen Regeln nicht möglich ist ein Tripel zu erzeugen, in dem ein **LogicalComponent**-Modellelement mehr als nur ein einziges **SystemComponent**-Modellelement realisiert.

Dementsprechend werden TGG-Regeln, wie die Regel in Abbildung 3.17, benötigt, welche Entsprechungen zwischen **SystemComponent**- und **LogicalComponent**-Modellelementen erzeugt. Diese Entsprechungen

repräsentieren die Realisierungsreferenzen im Korrespondenzmodell, welche die Modellelemente im Capella-Projekt direkt in Bezug zueinander setzen. Dabei muss sichergestellt werden, dass das **SystemComponent**- und **LogicalComponent**-Modellelement sich zuvor noch nicht entsprochen haben, damit Entsprechungen nicht redundant im Korrespondenzmodell abgebildet werden. Der Kontext dieser Regel beinhaltet ein **SystemAnalysis**-Modellelement *saArch*, welches dem **LogicalArchitecture**-Modellelement *laArch* entspricht. Diese Entsprechung wird im Kontext mit dem **Sa2LaArchitecture**-Korrespondenzmodellelement, welches *saArch* und *laArch* referenziert, gefordert. Die Wurzel-Komponenten-Pakete beider Systemmodelle werden ebenfalls im Kontext vorausgesetzt, da diese transitiv die Komponenten enthalten. Dementsprechend ist das **SystemComponentPkg**-Modellelement *rootScPkg* in *saArch* und das **LogicalComponentPkg**-Modellelement *rootLcPkg* in *laArch* enthalten. Das **Sa2LaCompPkg**-Korrespondenzelement stellt die Entsprechung zwischen *rootScPkg* und *rootLcPkg* durch das Referenzieren beider Modellelemente dar. Ein **SystemComponent**-Modellelement *sc* und ein **LogicalComponent**-Modellelement *lc* sind ebenso Kontext. Ein **ContainedSaComp**-Korrespondenzmodellelement setzt *sc* und *rootLcPkg* zueinander in Bezug und stellt dar, dass *sc* transitiv in *saArch* enthalten ist. Ebenfalls setzt ein **ContainedLaComp**-Korrespondenzmodellelement *lc* und *rootScPkg* zueinander in Bezug und modelliert, dass *lc* transitiv in *laArch* enthalten ist. Des Weiteren wird im Kontext mit einer NAC gefordert, dass sich *sc* und *lc* nicht entsprechen und somit noch kein **Sa2LaComp**-Korrespondenzmodellelement existiert, welches *sc* und *lc* referenziert. Die NAC ist in Abbildung 3.17 mit einem roten Verbots-Symbol über dem **Sa2LaComp**-Kontextmodellelement dargestellt. Dadurch, dass im Kontext nicht deklariert wird, welche Modellelemente *sc* und *lc* enthalten, werden jegliche sich nicht entsprechende *sc*- und *lc*-Modellelementpaare beschrieben. Durch Anwendung dieser Regel werden *sc* und *lc* durch ein neu angelegtes **Sa2LaComp**-Korrespondenzmodellelement zueinander in Bezug gesetzt. Eine TGG-Regel, welche Realisierungsreferenzen zwischen **SystemComponentPkg**- und **LogicalComponentPkg**-Modellelementen repräsentiert als ein Korrespondenzmodellelement erzeugt, kann dem Anhang A.2 entnommen werden. Diese Regeln müssen ebenfalls von einem Nutzer überwacht werden, damit nicht willkürlich Modellelemente zueinander in Bezug gesetzt werden und dies nur geschehen kann, wenn der Nutzer es für sinnvoll empfindet.

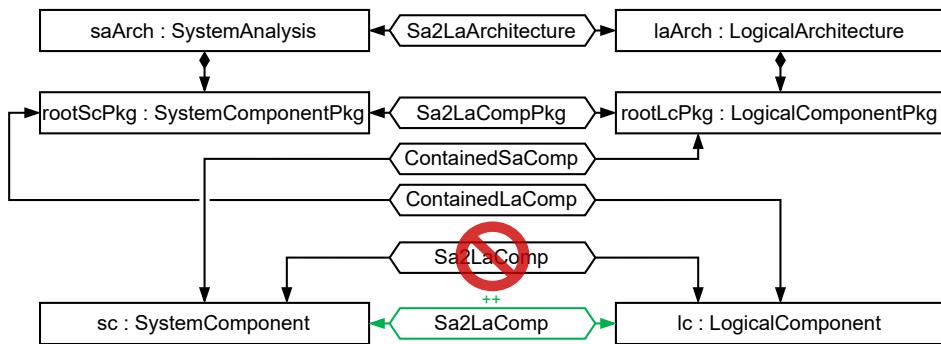


Abbildung 3.17.: TGG-Regel zum Erzeugen einer Realisierungsbeziehung zwischen **SystemComponent**- und **LogicalComponent**-Modellelementen, welche sich zuvor nicht entsprochen haben.

### 3.3. Erzeugen von Beziehungen zwischen Modellelementen

Mit TGG-Regeln, wie jene zuvor vorgestellten, ist es möglich, Modellelemente in einem SA- und LA-Systemmodellpaar zu erzeugen und zu synchronisieren. Ein Großteil der in den Systemmodellen enthaltenen Informationen werden mit Beziehungen zwischen Modellelementen dargestellt. In diesem Abschnitt wird

beispielhaft vorgeführt, wie TGG-Regeln zum Synchronisieren der Beziehungen definiert werden können. In Capella werden jegliche Beziehungen mit Modellelementen dargestellt, welche die in Bezug zueinander stehenden Modellelemente referenzieren. Dies wird im Folgenden anhand von Generalisierungsbeziehungen zwischen Komponenten erläutert und ist in Abbildung 3.18 visualisiert. Das **SystemComponent**-Modellelement *scSuper* ist der übergeordnete Komponententyp des **SystemComponent**-Modellelements *scSub*. Wie in Abbildung 3.18a dargestellt wird, werden in UML Generalisierungsbeziehungen mit einem Pfeil ausgehend vom untergeordneten zum übergeordneten Typen, in diesem Fall von *scSub* zu *scSuper*, dargestellt.



Abbildung 3.18.: **SystemComponent**-Modellelement *scSuper* ist der übergeordnete Komponententyp des **SystemComponent**-Modellelements *scSub*. Generalisierungsbeziehung links in UML und rechts in Capella dargestellt.

Wie in Abbildung 3.18b abgebildet ist, wird in Capella diese Beziehung mit einem **Generalization**-Modellelement dargestellt. Das **Generalization**-Modellelement *gen* referenziert die **SystemComponent**-Modellelemente und setzt diese zueinander in Bezug. Auf den übergeordneten Typen *scSuper* wird mit der *super*-Referenz und auf den untergeordneten Typen *scSub* wird mit der *sub*-Referenz verwiesen. Zum Darstellen anderer Beziehungstypen werden lediglich andere Knotentypen eingesetzt, welche auf die in Beziehung zueinander stehenden Modellelemente mit entsprechend benannten Referenzen verweisen. Daraus folgt, dass die TGG-Regeln zum Synchronisieren von Beziehungen in der Regel strukturell identisch sind. Es ist somit für das Verständnis ausreichend, den Umgang mit Beziehungen zwischen Modellelementen an der TGG-Regel für Generalisierungsbeziehungen vorzuführen.

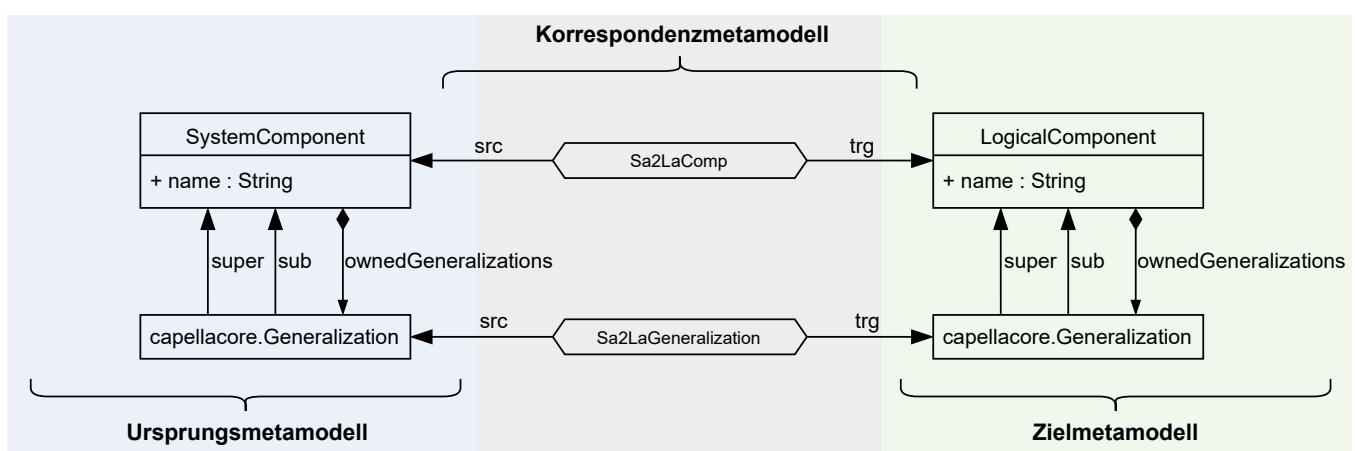


Abbildung 3.19.: Erweiterung des Korrespondenzmetamodells aus Abbildung 3.4 um Generalisierungsbeziehungen zwischen Komponenten.

Es ist zu beachten, dass die Beziehungsmodellelemente meist in anderen Metamodellen als das Metamodell des Systemmodells spezifiziert sind. Das **Generalization**-Modellelement ist unter anderem im *capellacore*-Metamodell spezifiziert und erhält deswegen im Folgenden den Präfix „*capellacore*“. Damit die Entsprechung zwischen Generalisierungsbeziehungen im SA- und LA-Systemmodell im Korrespondenzmodell abgebildet werden können, muss zuvor das Korrespondenzmetamodell aus Abbildung 3.4 erweitert werden. Generalisierungsbeziehungen werden sowohl im SA- als auch im LA-Systemmodell mit **Generalization**-Modellelementen dargestellt, wie in der Erweiterung des Korrespondenzmetamodells in Abbildung 3.19 zu sehen ist. Die jeweils untergeordnete Komponente enthält das **Generalization**-Modellelement, welches die untergeordnete Komponente über die *sub*-Referenz mit der übergeordneten Komponente über die *super*-Referenz in Beziehung zueinander setzt. Eine TGG-Regel, welche zum Synchronisieren solcher Generalisierungsbeziehungen genutzt werden kann, lässt sich wie folgt definieren und ist in Abbildung 3.20 dargestellt.

Das **SystemAnalysis**-Modellelement *saArch*, welches das Wurzel-**SystemComponentPkg**-Modellelement *rootScPkg* enthält, und dessen korrespondierendes **LogicalArchitecture**-Modellelement *laArch*, welches das Wurzel-**LogicalComponentPkg**-Modellelement *rootLcPkg* enthält, sind Teil des Kontexts dieser Regel. Das **Sa2LaArchitecture**-Korrespondenzelement stellt die Entsprechung zwischen *saArch* und *laArch* dar, während ein **Sa2LaCompPkg**-Korrespondenzmodellelement *rootScPkg* und *rootLcPkg* zueinander in Bezug setzt. Des Weiteren sind je Systemmodell zwei Komponenten Teil des Kontexts. Im SA-Systemmodell sind die **SystemComponent**-Modellelemente *scSuper* und *scSub* transitiv enthalten. Dies wird durch zwei **ContainedSaComp**-Korrespondenzmodellelemente, welche *scSuper* und *scSub* mit *rootLcPkg* in Bezug zueinander setzen, im Korrespondenzmodell dargestellt. Die beiden **SystemComponent**-Modellelemente besitzen Realisierungen im LA-Systemmodell, mit welchen sie über **Sa2LaComp**-Korrespondenzmodellelemente verbunden sind. *scSuper* entspricht *lcSuper* und *scSub* entspricht *lcSub*. *lcSuper* und *lcSub* werden durch **ContainedLaComp**-Korrespondenzelemente in Bezug zu *rootScPkg* gesetzt, womit dargestellt wird, dass diese Komponenten transitiv in *laArch* enthalten sind. Die Regel deklariert das Erzeugen eines neuen **Generalization**-Modellelements im SA- und LA-Systemmodell. Im SA-Systemmodell wird das **Generalization**-Modellelement *scGen* innerhalb von *scSub* erzeugt. Auch werden Referenzen von *scGen* zu *scSub* und von *scGen* zu *scSuper* angelegt. Im LA-Systemmodell wird das **Generalization**-Modellelement *lcGen* als Inhalt von *lcSub* angelegt. Des Weiteren referenziert *lcGen* *lcSub* und *lcSuper*. Das zu erzeugende **Sa2LaGeneralization**-Korrespondenzmodellelement referenziert *scGen* und *lcGen* und stellt deren Entsprechung dar.

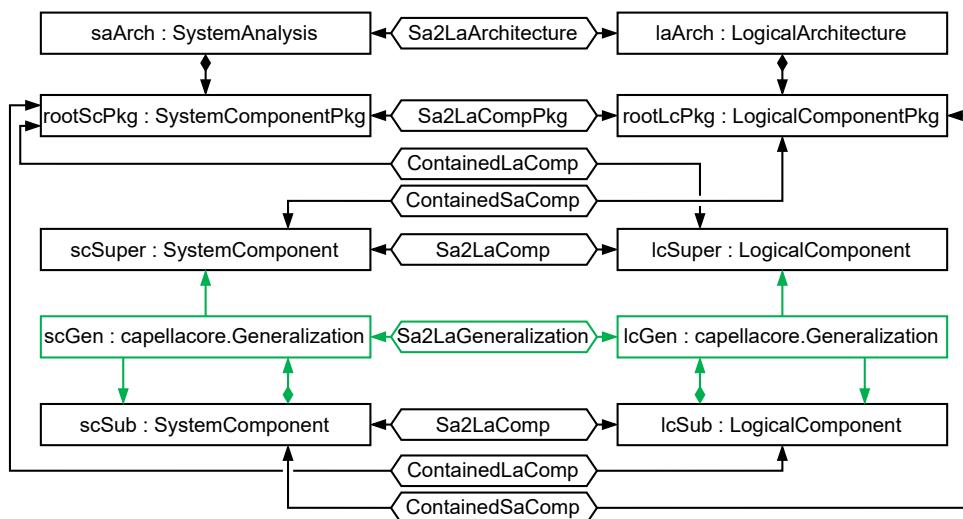


Abbildung 3.20.: TGG-Regel zum Erzeugen von Generalisierungsbeziehungen zwischen Komponenten.

Die Struktur für Regeln zum Erzeugen weiterer Beziehungen zwischen Modellelementen ist identisch zur vorgestellten Regel. Unterschiede liegen lediglich im Knotentypen der referenzierten Modellelemente und der Knotentyp des Modellelements, welche die Beziehung darstellt. Die implementierte TGG enthält des Weiteren Regeln zum Erzeugen von Modellelementen, welche folgende Beziehungen in den Systemmodellen darstellen:

- Generalisierungsbeziehungen zwischen Fähigkeiten.
- Komponentenaustausch zwischen Komponenten.
- Physische Verbindungen oder Kommunikationswege zwischen Komponenten.
- Zuordnung von Fähigkeiten zu Komponenten.
- Enthält-Beziehungen von Fähigkeiten.
- Erweiterungsbeziehungen von Fähigkeiten.
- Zuordnung von Funktionen zu Komponenten.

## 4. Evaluation

Infolge dieser Arbeit ist ein Ansatz zum Synchronisieren von Capella-Modellen mittels TGGen beispielhaft vorgestellt und implementiert worden. Die vollständige implementierte TGG besteht aus insgesamt 69 Regeln zum Synchronisieren von ausgewählten Modellelementbeziehungen, Komponenten-, Funktion- und Fähigkeit-Strukturen zwischen SA- und LA-Systemmodellen. Mit diesen Regeln sind Übersetzungs- und Synchronisationsprototypen mit eMoflon::IBeX erstellt worden. Hiermit konnte gezeigt werden, dass das Synchronisieren von Capella-Modellen mittels TGGen nicht nur theoretisch möglich sein sollte, sondern auch technisch realisierbar ist. Es ist allerdings zu beachten, dass die TGG nur einen kleinen Teil der gesamten Capella-ML und somit nur einen kleinen Teil aller Capella-Modelle abdeckt. Als Folge dessen ist das Ergebnis dieser Arbeit, dass das Synchronisieren von Capella-Modellen mittels TGGen möglich und realisierbar ist, nur bedingt gültig. In zukünftigen Arbeiten sollte die TGG erweitert werden, um weitere Capella-Modellelemente und Capella-Modelle synchronisieren zu können. In diesem Kapitel wird der implementierte Ansatz anhand des Laufzeitverhaltens evaluiert. Die Analyse des Laufzeitverhaltens soll Auskunft über die Skalierbarkeit der mit eMoflon::IBeX erstellten Übersetzungs- und Synchronisationsprototypen geben.

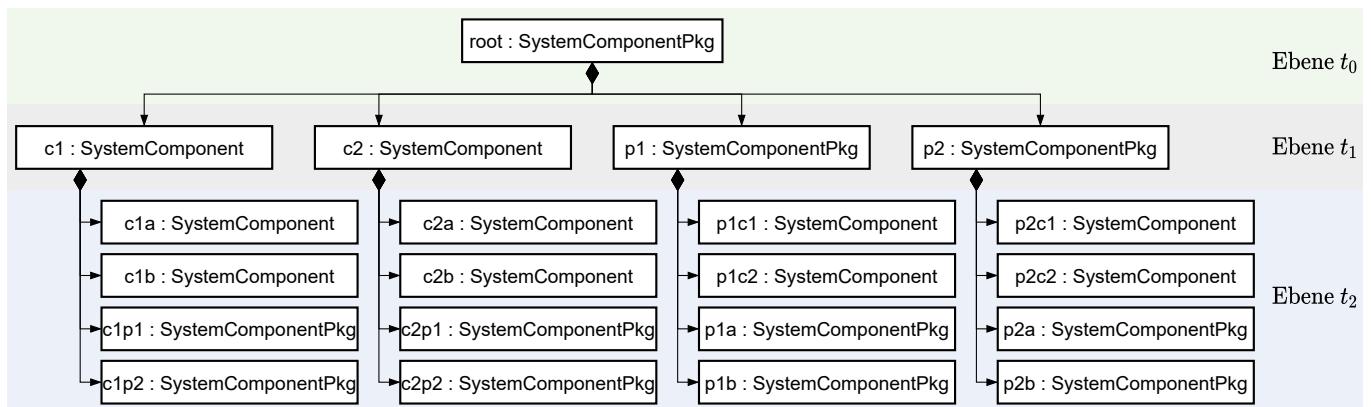


Abbildung 4.1.: Visualisierung einer rekursiven Komponenten-Hierarchie mit  $t = 2$ .

Als Testumgebung wurde eine virtuelle Maschine mit Debian 11 als Betriebssystem eingesetzt. Dieser wurden acht Prozessorkerne eines AMD Ryzen 2700x<sup>1</sup> mit einer Taktgeschwindigkeit von 3,6 GHz und 16 GB Random Access Memory (engl. für Arbeitsspeicher) (RAM) zur Verfügung gestellt. Zum Untersuchen des Laufzeitverhaltens von Synchronisierungs- und Batch-Übersetzungsprozessen werden Laufzeit und RAM-Verbrauch gemessen. Dafür werden SA-Systemmodelle der Größe  $t$  generiert. Die Größe  $t$  gibt hierbei die Menge an Hierarchie-Ebenen der rekursiven Komponenten-, Funktion- und Fähigkeit-Hierarchien im Systemmodell an. Des Weiteren enthält jedes Element einer Hierarchie-Ebene, abgesehen von denen auf der letzten Ebene, genau  $t$  Modellelemente von jeden Modellelementtypen, welche enthalten sein können. Abbildung 4.1 stellt

<sup>1</sup><https://www.amd.com/en/products/cpu/amd-ryzen-7-2700x>

dies anhand der rekursiven Komponenten-Hierarchie für  $t = 2$  dar. Das Wurzel-**SystemComponentPkg**-Modellelement *root* ist Teil der Hierarchieebene  $t_0$ . Die Elemente, welche in *root* enthalten sind, sind Teil der Hierarchieebene  $t_1$ . In **SystemComponent**- und **SystemComponentPkg**-Modellelementen können **SystemComponent**- und **SystemComponentPkg**-Modellelemente enthalten sein. Dementsprechend enthält *root* die zwei **SystemComponent**-Modellelemente *c1* und *c2* und die zwei **SystemComponentPkg**-Modellelemente *p1* und *p2*. Die Modellelemente *c1*, *c2*, *p1* und *p2* enthalten jeweils wiederum zwei **SystemComponent**- und **SystemComponentPkg**-Modellelemente, welche der Hierarchieebene  $t_2$  angehören. Diese befinden sich auf der untersten Hierarchieebene und enthalten somit keine weiteren Modellelemente. Damit das SA-Systemmodell nicht ausschließlich aus den rekursiven Komponenten-, Fähigkeit- und Funktion-Hierarchien besteht, werden des Weiteren jeweils  $t$  Beziehungen jeder von der implementierten TGG unterstützten Beziehungsart angelegt. In Tabelle 4.1 werden die Größen der generierten SA-Systemmodelle, welche zum Testen verwendet worden sind, in Abhängigkeit von  $t$  aufgelistet.

	$t = 1$	$t = 2$	$t = 3$	$t = 4$
<b>Modellgröße</b>	sehr klein	klein	groß	sehr groß
<b>Knotenanzahl</b>	26	93	766	12435

Tabelle 4.1.: Größe des generierten SA-Systemmodells in Abhängigkeit von  $t$ .

Es ist zu beachten, dass die Modellgröße exponentiell mit steigendem  $t$  zunimmt. Modelle mit  $t = 4$  sind mit 12435 Modellelementen sehr groß und beinhalten mehr als das 16-fache an Modellelementen als Modelle mit  $t = 3$  und fast das 480-fache an Modellelementen als Modelle mit  $t = 1$ . Dementsprechend sind die Messergebnisse bei  $t = 4$  nicht direkt mit den anderen vergleichbar. Die Messergebnisse bei  $t = 4$  sollen Auskunft über das Verhalten bei sehr großen Modellen geben und möglicherweise Grenzen der technischen Implementierung aufweisen.

## 4.1. Laufzeitverhalten von Batch-Übersetzungen

Zum Ermitteln des Laufzeitverhaltens von Batch-Übersetzungen wurde jeweils ein SA-Systemmodell der Größe  $t$  generiert und anschließend der Übersetzungsprozess gestartet. Dies wurde für  $t = 1$  bis  $t = 4$  200-mal wiederholt, um einen Mittelwert für die Übersetzungsdauer und für den RAM-Verbrauch zu erhalten. Die Messergebnisse der Übersetzungsdauer sind tabellarisch und als Balkendiagramm in Abbildung 4.2 dargestellt. Die erste Spalte der Tabelle gibt die verschiedenen Werte für  $t$  an. Die sich daneben befindende Knoten-Spalte beinhaltet die Anzahl der Modellelemente im generierten SA-Modell. Die gemittelte Übersetzungsdauer ist mit dessen Standardabweichung hinter dem  $\pm$ -Zeichen in der Laufzeit-Spalte angegeben. Die vierte und fünfte Spalte geben die relative Anzahl an Knoten und die gemittelte Laufzeit einer Zeile in Bezug zur ersten Zeile an. Die mittlere Laufzeit ist für sehr kleine bis große Modelle,  $t$  kleiner als vier, deutlich unter einer Sekunde. Erst große Modelle mit  $t = 3$  benötigen im Schnitt mehr als eine halbe Sekunde. Dabei werden ca. 0,6209 Sekunden, mit einer Standardabweichung von 0,0490 Sekunden, zum vollständigen Übersetzen des Modells benötigt. Beim Vergleichen der relativen Größe und der relativen Laufzeit ist bei diesen Messwerten bemerkbar, dass Laufzeit und Modellgröße direkt korrelieren. Modelle mit  $t = 2$  besitzen ca. 358 % der Modellelemente als Modelle mit  $t = 1$ . Ebenso werden ungefähr 365 % der Laufzeit zum Übersetzen des SA-Systemmodells mit  $t = 2$  benötigt. Die relative Modellgröße ist hier beinahe identisch zur relativen Laufzeit, wie auch bei Modellen höheren  $t$ -Werten zu erkennen ist. Bei sehr großen Modellen mit  $t = 4$ , welche 12435 Modellelemente und somit fast das 480-fache an Elementen als Modelle mit  $t = 1$  beinhalten, sind relative

Laufzeit und relative Größe in derselben Größenordnung. Die relative Größe beträgt ca. 47827 % und die relative Laufzeit beträgt ca. 42524 %. Es liegt eine Differenz von ungefähr 5303 % zwischen diesen Werten vor, was darauf deuten könnte, dass das Wachstum der relativen Laufzeit bei sehr großen Modellen weniger stark als das Wachstum der Modellgröße ausfallen könnte. Dieses Wachstumsverhalten der relativen Laufzeit deutet darauf hin, dass Laufzeit mit der Modellgröße ungefähr linear skalieren könnte.

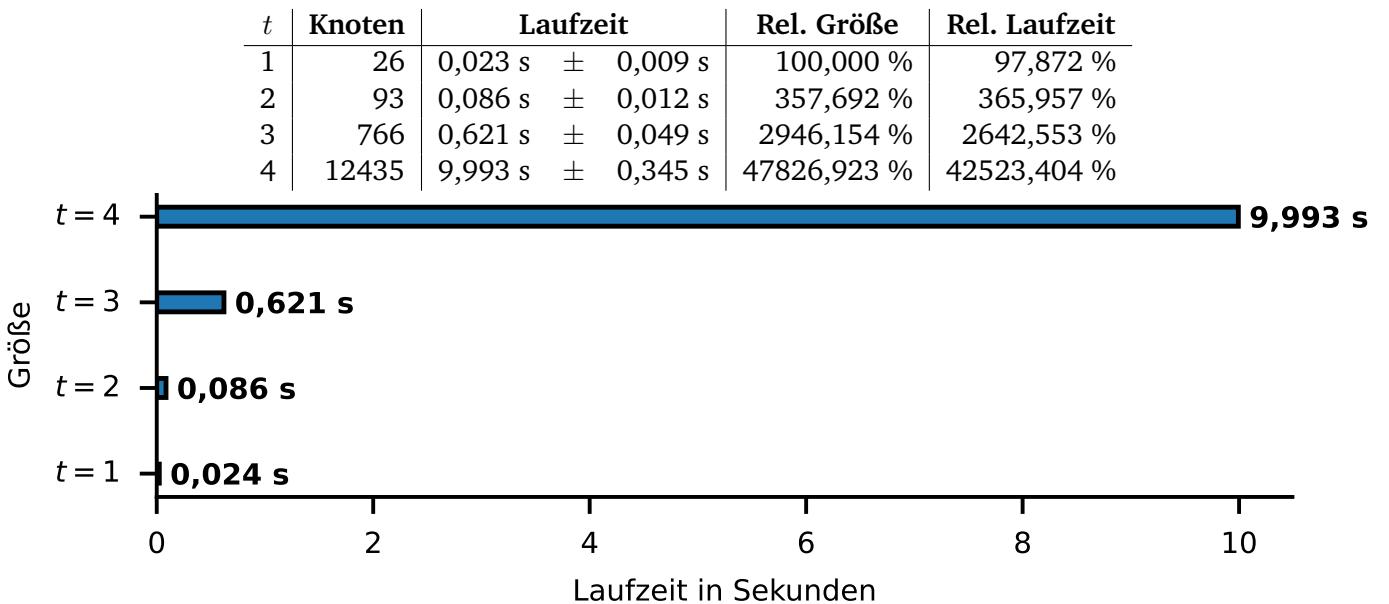


Abbildung 4.2.: Laufzeiten der Batch-Übersetzungen von SA-Systemmodellen der Größe  $t$  tabellarisch und als Balkendiagramm dargestellt.

Ein Zusammenhang zwischen Modellgröße und RAM-Verbrauch ließ sich auch mit den Messwerten des RAM-Verbrauchs feststellen. Diese Werte sind durch das Aufrufen des Garbage-Kollektors und Messen der anschließend freigegebenen Menge an RAM erhalten worden. Es ist zu bedenken, dass solche Messung nicht zuverlässig sind, diese jedoch für eine grobe Beurteilung des RAM-Verbrauchs eines Übersetzungsprozesses ausreichend sind. Die Ergebnisse der Testreihe sind in Abbildung 4.3 tabellarisch und als Balkendiagramm dargestellt. Die erste, zweite und vierte Spalte sind identisch zu denen aus der Tabelle in Abbildung 4.2. Die dritte Spalte gibt den durchschnittlichen RAM-Verbrauch mit deren Standardabweichung in Megabyte (MB) an. Die letzte Spalte gibt den relativen durchschnittlichen Verbrauch einer Zeile in Bezug zum durchschnittlichen Verbrauch der ersten Zeile an. Während relative Laufzeit und relative Modellgröße stets dieselbe Größenordnung teilen, ist dies mit dem relativen RAM-Verbrauch nicht der Fall. Es scheint keinen linearen Zusammenhang zwischen Modellgröße und RAM-Verbrauch zu bestehen, dennoch steigt der RAM-Verbrauch erwartungsgemäß ebenfalls mit der Modellgröße. Für Modelle mit  $t = 2$  betrug der RAM-Verbrauch durchschnittlich 68,21 MB, mit einer Standardabweichung von 0,6449 MB, was 106 % des durchschnittlichen RAM-Verbrauchs zum Übersetzen von Modellen mit  $t = 1$  sind. Das Übersetzen von Modellen mit  $t = 3$  benötigte ca. 150 % und Modelle  $t = 4$  benötigten 871 % des RAM-Verbrauchs. Obwohl Modelle mit  $t = 4$  fast 480-mal so viele Knoten wie Modelle mit  $t = 1$  enthalten, steigt der RAM-Verbrauch nur auf knapp das Achtfache an. Die Laufzeit bei  $t = 4$  steigt allerdings auf ungefähr das 43-fache der Laufzeit zum Übersetzen von Modellen mit  $t = 1$ . Der Anstieg an benötigter Laufzeit in Relation zur Modellgröße ist viel größer als der Anstieg an RAM-Verbrauch. Daraus kann geschlossen werden, dass bei der Übersetzung von den synthetischen Modellen die Laufzeit mit der Modellgröße schlechter skalieren kann als der RAM-Verbrauch. Es könnte somit sein, dass die Dauer einer Übersetzung oder einer Synchronisierung der synthetischen Modelle ab einer

gewissen Modellgröße zu hoch wird. Dies geht auch aus den Messwerten der Synchronisierungsprozesse hervor, welche im Folgenden vorgestellt werden.

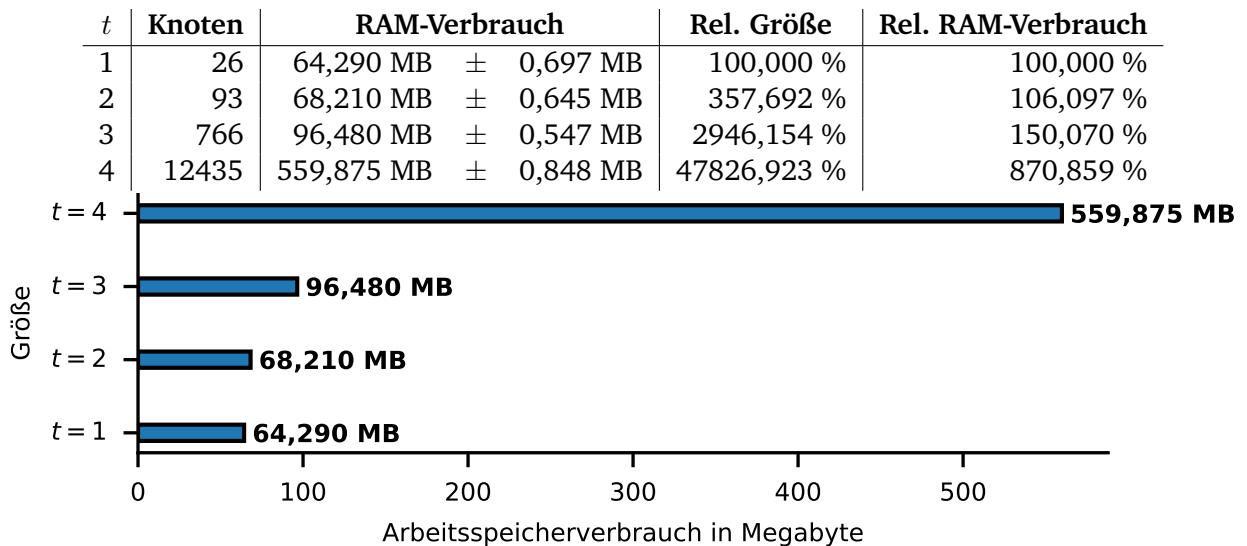


Abbildung 4.3.: RAM-Verbrauch der Batch-Übersetzungen von SA-Systemmodellen der Größe  $t$  tabellarisch und als Balkendiagramm dargestellt.

## 4.2. Laufzeitverhalten der Synchronisierung

Zum Beurteilen des Laufzeitverhaltens vom implementierten Synchronisationsprototyps werden SA-Systemmodelle der Größe  $t$  generiert und diese in LA-Systemmodelle übersetzt. Anschließend wird das SA-Systemmodell modifiziert, womit die Modelle nicht mehr konsistent zueinander sind. Es folgt der Synchronisierungsprozess, bei welchem analog zum vorherigen Abschnitt die durchschnittliche Laufzeit und der mittlere RAM-Verbrauch ermittelt wird. Damit der Einfluss von Modellgröße und Art der Änderung beurteilt werden kann, werden drei verschiedene Szenarien betrachtet. Mit dem ersten Szenario wird die Auswirkung der Modellgröße auf das Laufzeitverhalten des Synchronisierungsprozesses untersucht. Dabei wird ein Modellelement dem SA-Modell hinzugefügt oder entfernt, wonach die Modelle synchronisiert werden. Dies wird für Modelle der Größe  $t = 1$  bis  $t = 4$  wiederholt. Mit dem zweiten Szenario wird betrachtet, wie sich eine steigende Anzahl an Änderungen bei gleichbleibender initialer Modellgröße auf das Laufzeitverhalten auswirkt. Dafür werden in 250-er Schritten 250 bis 2000 Modellelemente an zufälligen Stellen im SA-Systemmodell neu erzeugt und anschließend die Modelle synchronisiert. In beiden dieser Szenarien wird lediglich das Hinzufügen oder das Entfernen von Modellelementen betrachtet. Deswegen soll mit dem dritten Szenario untersucht werden, wie sich das Umhängen einer Modellelementstruktur von einer zu einer anderen Stelle im Modell auf das Laufzeitverhalten bei Modellen unterschiedlicher Größen auswirkt. Dies wird wie im ersten Szenario für Modelle der Größe  $t = 1$  bis  $t = 4$  untersucht.

Zum Untersuchen des ersten Szenarios wurde ein **SystemComponent**-Modellelement im Wurzel-**SystemComponentPkg**-Modellelement erzeugt und anschließend das SA- mit dem LA-Systemmodell synchronisiert. Dies wurde jeweils 200-mal für Modelle mit  $t = 1$  bis  $t = 4$  wiederholt und Messwerte für die Laufzeiten und den RAM-Verbrauch erhoben. Die Messwerte der Laufzeiten sind in Abbildung 4.4 und die Messwerte des RAM-Verbrauchs sind in Abbildung 4.5 tabellarisch und als Balkendiagramm dargestellt.

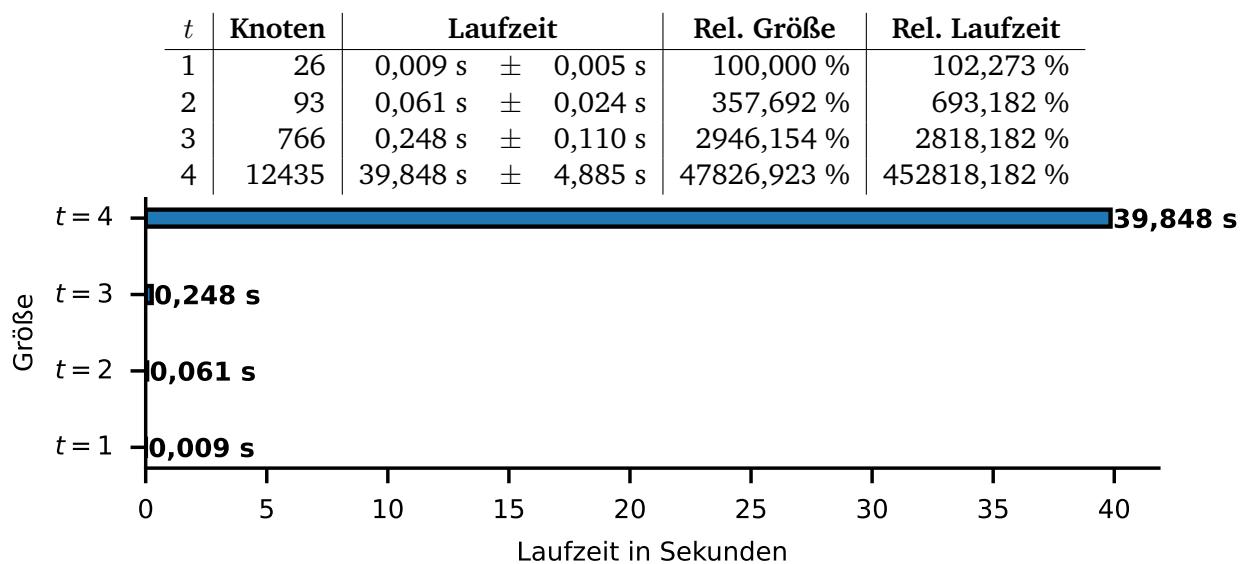


Abbildung 4.4.: Laufzeiten einer Synchronisierung von Modellen der Größe *t* nach Hinzufügen einer Komponente. Als Tabelle und als Balkendiagramm dargestellt.

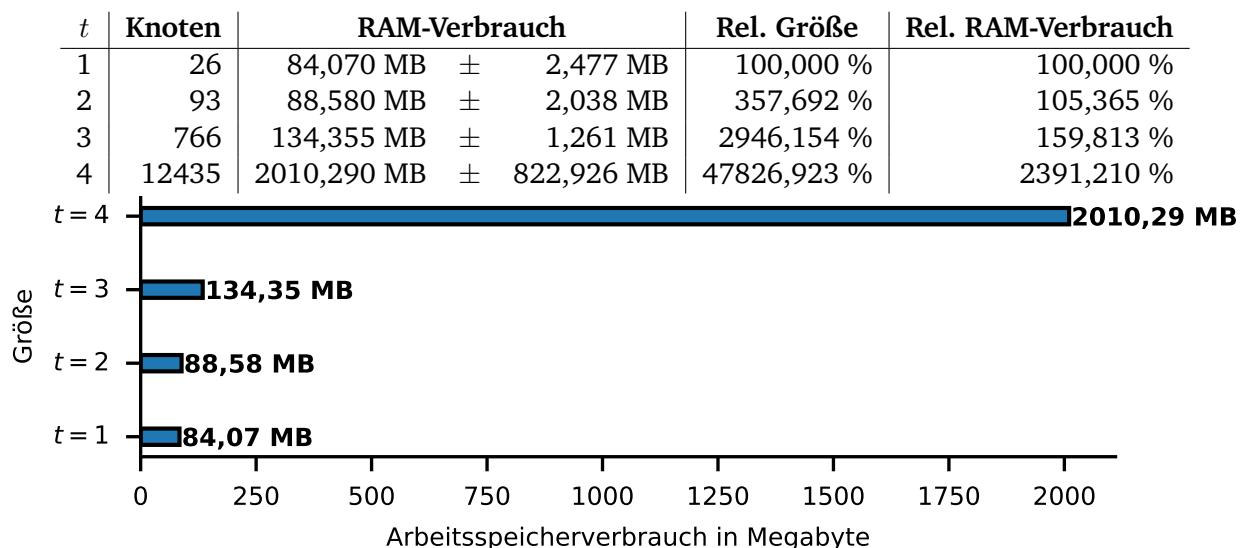


Abbildung 4.5.: RAM-Verbrauch einer Synchronisierung von Modellen der Größe *t* nach Hinzufügen einer Komponente. Als Tabelle und als Balkendiagramm dargestellt.

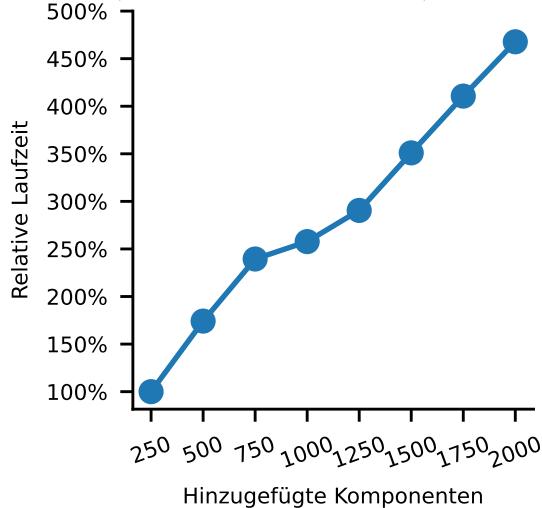
Ein direkter Vergleich der Laufzeiten, mit denen der Batch-Übersetzungen aus Abbildung 4.2 ist streng genommen nicht möglich, da das zu synchronisierende SA-Systemmodell durch das Hinzufügen des **System-Component**-Modellelementes ein Element mehr beinhaltet, als das Modell, welches für die Batch-Übersetzung verwendet worden ist. Diese Differenz ist allerdings im Vergleich zu den Modellgrößen vernachlässigbar, weshalb im folgenden dennoch mit den Werten aus Abschnitt 4.1 verglichen wird. Dabei ist bemerkbar, dass der Synchronisierungsprozess selbst für große Modelle mit *t* = 3 deutlich schneller ist als eine Batch-Übersetzung ähnlicher Modelle. Beispielsweise wird zur Synchronisierung von Modellen mit *t* = 1 nur 0,009 Sekunden im Schnitt benötigt, während die Übersetzung eines ähnlich großen Modells mit 0,023 Sekunden mehr als doppelt so lange dauert. Werden allerdings die Messwerte zum Synchronisieren von sehr großen Modellen

mit  $t = 4$  und ungefähr 480-mal so vielen Modellelementen wie in Modellen mit  $t = 1$  betrachtet, so fällt auf, dass mit durchschnittlich beinahe 40 Sekunden Laufzeit ungefähr viermal länger zum Synchronisieren benötigt wird, als die Batch-Übersetzung eines ähnlich großen Modells. Dass die Laufzeit mit exponentiell steigender Modellgröße auch die Laufzeit der Synchronisierung drastisch ansteigt, ist erwartungsgemäß. Allerdings wurde nicht erwartet, dass die Laufzeit der Synchronisierung viermal länger als die einer Batch-Übersetzung von ähnlichen Modellen ist. Der RAM-Verbrauch ist bei Modellen mit  $t$  kleiner als vier zwar stets höher als bei der Batch-Übersetzung ähnlicher Modelle, aber der relative RAM-Verbrauch ist stets in derselben Größenordnung. Beispielsweise beträgt dieser bei  $t = 2$  ungefähr 105,36 % des Verbrauchs eines Synchronisierungsprozesses mit Modellen der Größe  $t = 1$  ist. Im Vergleich dazu kann Abbildung 4.3 entnommen werden, dass bei Batch-Übersetzungen ähnlicher Modelle mit  $t = 2$  der durchschnittliche relative Verbrauch 106,1 % betrug. Die Differenz des relativen Verbrauchs ist hier somit unter einem Prozent. Bei diesen Messergebnissen ist auffällig, dass Synchronisierungsprozesse von Modellen mit  $t = 4$  nicht nur länger als eine Batch-Übersetzung von ähnlichen Modellen dauert, sondern ebenfalls sehr viel mehr RAM benötigen. Es wurden zum Synchronisieren im Schnitt ungefähr zwei Gigabyte an RAM gebraucht. Ähnliches Verhalten von Laufzeit und RAM-Verbrauch konnten auch beim Wiederholen dieser Testreihe bemerkt werden und die mittlere Laufzeit und der mittlere RAM-Verbrauch sind stets in derselben Größenordnung. Dabei wurde allerdings ein **SystemComponent**-Modellelement aus dem Modell entfernt, anstatt ein neues hinzuzufügen. Die Messergebnisse sind dem Anhang B beigelegt. Des Weiteren wurden zusätzliche Testreihen durchgeführt, in welchen eine Komponenten-Generalisierungsbeziehungen neu erzeugt oder eine bestehende entfernt wurde. Die Darstellung der Messergebnisse ist ebenfalls dem Anhang B zu entnehmen. Bei diesen ist ebenfalls ähnliches Laufzeitverhalten bemerkbar gewesen.

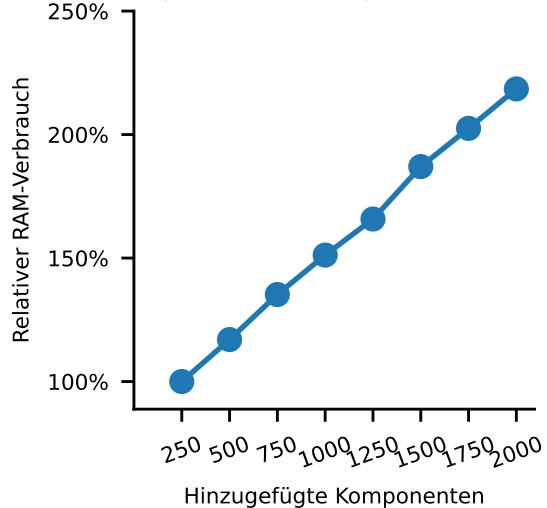
Zum Untersuchen des zweiten Szenarios wurden zuerst zueinander konsistente SA- und LA-Systemmodelle der Größe  $t = 3$  erzeugt und anschließend  $n$  **SystemComponent**-Modellelemente nacheinander an eine zufällige Stelle in der Komponenten-Hierarchie des SA-Modells eingefügt. Anschließend wurde der Synchronisierungsprozess gestartet und die Laufzeit und der RAM-Verbrauch gemessen. Dies wurde jeweils 200-mal für  $n = 250$  bis  $n = 2000$  mit einer Schrittweite von 250 wiederholt. Die Ergebnisse sind in Abbildung 4.6 tabellarisch und mit zwei Graphen dargestellt. Die erste Spalte der Tabelle gibt  $n$ , und somit die Menge an hinzugefügten **SystemComponent**-Modellelementen, an. In der zweiten Spalte ist die gemittelte Laufzeit mit dessen Standardabweichungen in Sekunden angegeben. Der durchschnittliche RAM-Verbrauch ist mit dessen Standardabweichung der dritten Spalte zu entnehmen. Die relative Laufzeit in der vierten Spalte gibt die durchschnittliche Laufzeit einer Zeile in Relation zur durchschnittlichen Laufzeit der ersten Zeile an. Die fünfte Spalte gibt den durchschnittlichen RAM-Verbrauch einer Zeile in Relation zum Verbrauch der ersten Zeile an. Abbildung 4.6a visualisiert die relativen Laufzeiten in Bezug zur Menge der hinzugefügten Modellelemente. Die Abszisse gibt die relative Laufzeit und die Ordinate die Menge der hinzugefügten Modellelemente an. In Abbildung 4.6b wird der relative RAM-Verbrauch ebenfalls in Bezug zur Menge der hinzugefügten Modellelemente dargestellt. Der relative RAM-Verbrauch ist über die Abszisse und die Menge der hinzugefügten **SystemComponent**-Modellelemente ist über die Ordinate aufgetragen. Es ist in diesen Abbildungen erkennbar, dass Laufzeit und RAM-Verbrauch in diesem Szenario zum linearen Skalieren mit der Menge an hinzugefügten Modellelementen tendieren. Des Weiteren ist der Anstieg der relativen Laufzeit mit steigender Menge an hinzugefügten Modellelementen in dieser Messreihe steiler als der Anstieg des relativen RAM-Verbrauchs. Dies deutet darauf hin, dass mit einer steigenden Anzahl an zu synchronisierenden Änderungen, wie ebenfalls zuvor bemerkt werden konnte, die Laufzeit schlechter als der RAM-Verbrauch zu skalieren scheint.

Damit Messwerte für das dritte Szenario gesammelt werden können, wurden zueinander konsistente SA- und LA-Systemmodelle der Größe  $t$  synthetisch erzeugt, ein **SystemComponent**-Modellelement aus dem Wurzel-**SystemComponentPkg**-Modellelement samt dessen Inhalt an eine andere Stelle in der Komponenten-

<b><math>n</math></b>	<b>Laufzeit</b>		<b>RAM-Verbrauch</b>		<b>Rel. Laufzeit</b>	<b>Rel. RAM-Verbrauch</b>
250	0,631 s	± 0,112 s	145,745 MB	± 2,583 MB	100,000 %	100,000 %
500	1,098 s	± 0,073 s	170,520 MB	± 6,616 MB	174,010 %	116,999 %
750	1,510 s	± 0,039 s	197,100 MB	± 1,204 MB	239,303 %	135,236 %
1000	1,625 s	± 0,052 s	220,370 MB	± 1,365 MB	257,528 %	151,202 %
1250	1,832 s	± 0,057 s	241,645 MB	± 4,267 MB	290,333 %	165,800 %
1500	2,213 s	± 0,065 s	272,595 MB	± 5,136 MB	350,713 %	187,036 %
1750	2,589 s	± 0,086 s	295,160 MB	± 3,424 MB	410,301 %	202,518 %
2000	2,949 s	± 0,110 s	318,440 MB	± 10,159 MB	467,353 %	218,491 %



(a) Relative Laufzeit in Bezug zur Menge hinzugefügter Komponenten.



(b) Relativer RAM-Verbrauch in Bezug zur Menge hinzugefügter Komponenten.

Abbildung 4.6.: Messwerte der Synchronisierung von Modellen der Größe  $t = 3$  nach dem Hinzufügen von  $n$  Komponenten.

Hierarchie verschoben und anschließend der Synchronisierungsprozess gestartet. Dies wurde für Modelle der Größe  $t = 1$  bis  $t = 4$  in der Regel 200-mal wiederholt. Bei dieser Testreihe ist zu bedenken, dass die Anzahl an insgesamt verschobenen Modellelementen exponentiell in Bezug zu  $t$  steigt. Dies liegt daran, wie die synthetischen Modelle erzeugt worden sind. Ein Modell der Größe  $t$  besitzt eine Komponenten-Hierarchie mit  $t$  Ebenen. Jedes Modellelement einer Ebene, abgesehen von der letzten Ebene, enthält  $t$  Komponenten und  $t$  Komponenten-Pakete. Wird also ein **SystemComponent**-Modellelement aus dem Wurzel-**SystemComponentPkg** samt dessen Inhalt an eine andere Stelle im Modell verschoben, so werden tatsächlich insgesamt  $v = \sum_{i=0}^{t-1} (2 \cdot t)^i$  **SystemComponent**- und **SystemComponentPkg**-Modellelemente verschoben. Somit wird bei Modellen mit  $t = 1$  ein einziges Modellelement verschoben. Bei  $t = 2$  sind es insgesamt fünf, bei  $t = 3$  insgesamt 43 und bei  $t = 4$  insgesamt 585 Modellelemente. Das Verschieben der Modellelemente wurde jeweils 200-mal für Modelle mit  $t = 1$ ,  $t = 2$  und  $t = 3$  wiederholt. Die Messergebnisse der Laufzeiten sind in Abbildung 4.7 und die Messergebnisse des RAM-Verbrauchs sind in Abbildung 4.8 tabellarisch und als Balkendiagramm abgebildet. Die durchschnittlichen Laufzeiten und der mittlere RAM-Verbrauch sind hier ähnlich zu denen aus dem ersten Szenario.

Für sehr große Modelle mit  $t = 4$ , welche ungefähr 480-mal so groß wie Modelle mit  $t = 1$  sind, konnten wegen instabilen Laufverhaltens allerdings nur fünf und nicht 200 Testergebnisse gesammelt werden. Dies lag daran, dass der Synchronisierungsprozess wegen oft nicht abgeschlossen werden konnte. In der Regel schlug

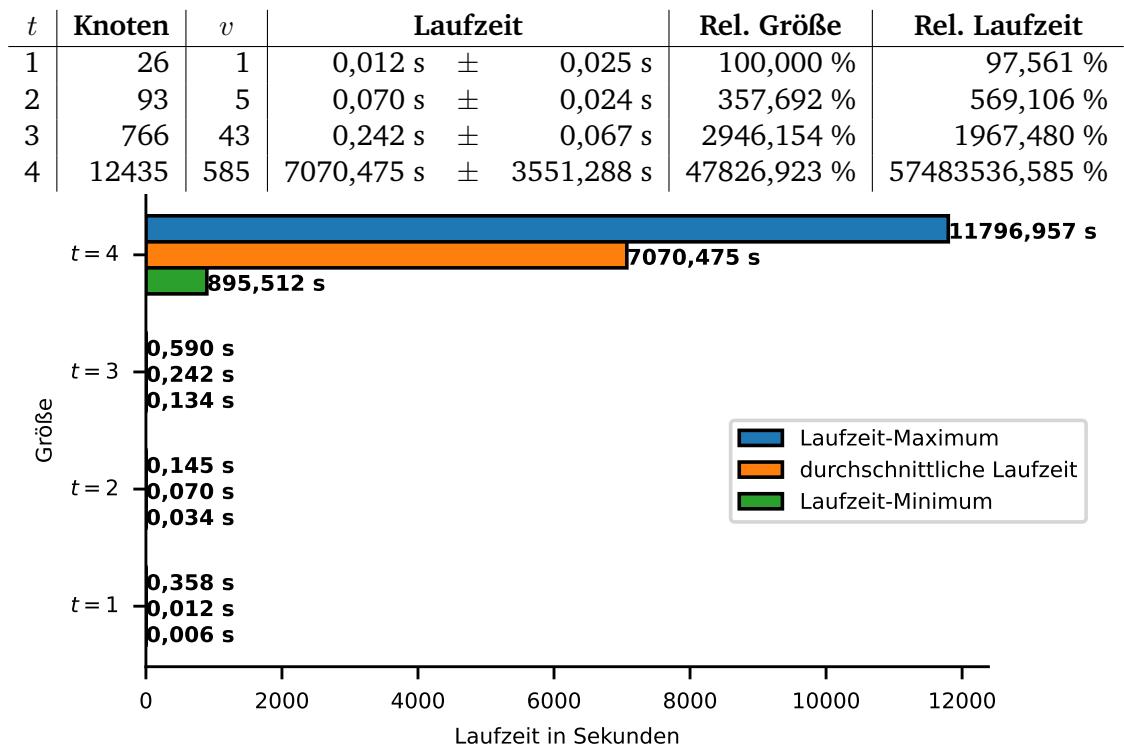


Abbildung 4.7.: Laufzeiten einer Synchronisierung von Modellen der Größe  $t$  nach Umhängen von insgesamt  $v$  Modellelementen als Tabelle und als Balkendiagramm dargestellt.

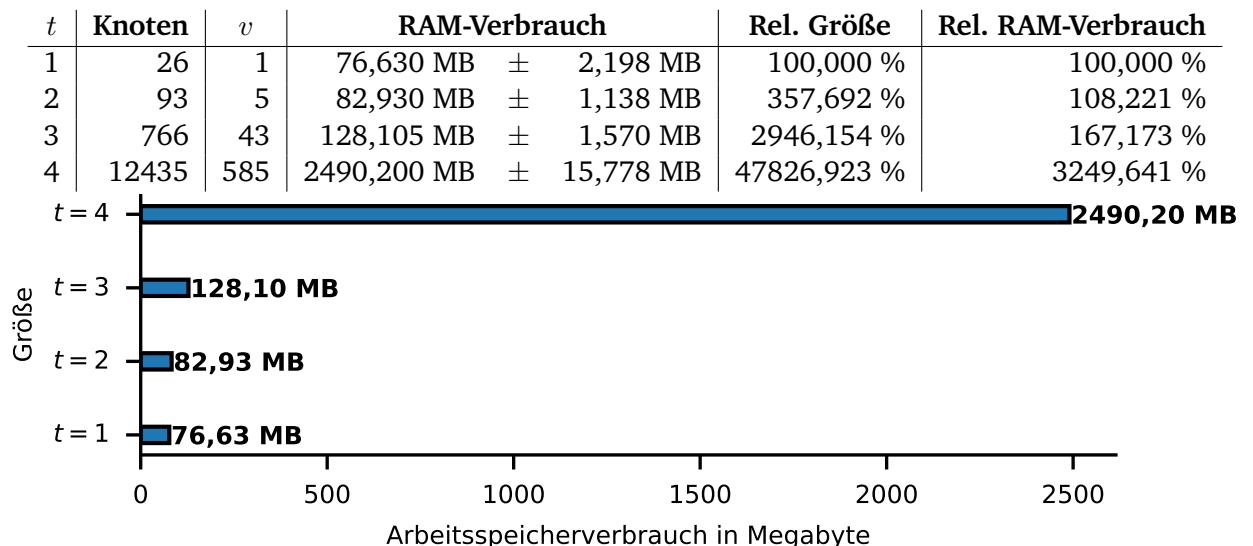


Abbildung 4.8.: RAM-Verbrauch einer Synchronisierung von Modellen der Größe  $t$  nach Umhängen von insgesamt  $v$  Modellelementen als Tabelle und als Balkendiagramm dargestellt.

die Synchronisierung wegen einer Überschreitung der maximalen Java-Heap-Größe vorzeitig fehl. In einigen Fällen brach der Prozess sogar ohne jeglicher Fehlermeldung ab oder die virtuelle Maschine reagierte nicht mehr auf Interaktionen und musste auf einen vorherigen Zustand zurückgesetzt werden. Die durchschnittliche Laufzeit der fünf validen Testergebnisse liegt bei 7070,475 Sekunden (fast zwei Stunden), während der

längste erfolgreiche Synchronisierungsprozess ganze 11796,957 Sekunden (ungefähr drei Stunden und 16 Minuten) lang dauerte. Es ist erwartungsgemäß, dass die Laufzeit mit exponentiell steigender Modellgröße und exponentiell steigender Menge an verschobenen Modellelementen drastisch ansteigt. Allerdings sind zwei Stunden durchschnittliche Laufzeit sehr lange und häufiges Abbrechen des Synchronisierungsprozesses unerwartet. Dieses Verhalten könnte womöglich mit der begrenzten Java-Heap-Größe zusammenhängen. Damit lässt sich allerdings die große Spanne zwischen einer minimalen Laufzeit von knapp unter 15 Minuten und einer maximalen Laufzeit von über 3 Stunden nicht direkt erklären. Es sollte in einer zukünftigen Arbeit untersucht werden, wieso und woran der Synchronisierungsprozess in diesem Kontext scheitert und ob die langen Laufzeiten damit zusammenhängen.

Zusammenfassend konnte folgendes Verhalten des Synchronisierungsprozesses beobachtet werden. Das Synchronisieren von kleinen Modellen der Größe  $t = 1$  bis hin zu großen Modellen der Größe  $t = 3$  tendierte dazu, etwas weniger Laufzeit als eine Batch-Übersetzung von ähnlichen Modellen zu benötigen. Bei sehr großen synthetischen Modellen der Größe  $t = 4$ , mit ungefähr 480-mal so vielen Modellelementen als Modelle der Größe  $t = 1$ , scheint eine Batch-Übersetzung ähnlicher Modelle ungefähr viermal schneller als eine Synchronisierung zu sein. Wurden allerdings sehr viele Modellelemente in Modellen der Größe  $t = 4$  verschoben und anschließend synchronisiert, so dauert dies äußerst lange und war in der Regel auch nicht erfolgreich möglich. Das Synchronisieren von synthetischen Modellen der Größe  $t = 3$  mit einer steigenden Anzahl von hinzugefügten Modellelementen hat ergeben, dass Laufzeit und RAM-Verbrauch dazu tendieren, beinahe linear mit der Anzahl an hinzugefügten Modellelementen zu steigen. Aus diesen Ergebnissen kann geschlossen werden, dass der implementierte Synchronisationsprototyp vermutlich performant genug sein könnte, um tatsächliche und nicht synthetische Systemmodelle bis zu einer gewissen Größe miteinander synchron halten zu können.

## 5. Verwandte Arbeiten

---

Während der Recherche konnten keine verwandten Arbeiten, welche ebenfalls das miteinander Synchronisieren von Capella-Modellen behandeln, gefunden werden. Es konnten allerdings andere Arbeiten entdeckt werden, in welchen Capella-Modelle entweder in andere Modelle übersetzt oder mit diesen synchronisiert werden. Diese werden im Folgenden zusammengefasst vorgestellt.

Die Arbeit von Badache et al. [29] beschreibt die Implementation eines Modellübersetzungsprototyps, welcher das Übersetzen von Capella-Modellen in OMGs MDA-ML *SysML*, welche auf UML basiert, ermögliche. Der Prototyp wurde mit dem Open-Source-Modelltransformationswerkzeug für EMF-basierte Modelle Kitalpha Transposer<sup>1</sup> als einen inkrementellen unidirektionalen Modellübersetzer implementiert. Dieser Übersetzer erzeugt ein SysML-Systemmodell aus einem Capella-LA-Systemmodell. Zur Beschreibung der Implementation listen Badache et al. Korrespondenzen zwischen Capella- und SysML-Modellelementen auf, für welche Transformationsregeln implementiert worden seien. Unter anderem entspreche ein **LogicalComponent**-Modellelement eines LA-Systemmodells einem UML-Klassenmodellelement in einem SysML-Systemmodell. Es ist allerdings zu beachten, dass dieser Übersetzer für die Capella-Modelle der Version 1.1 erstellt worden ist, welche nicht mehr auf den offiziellen Webseiten erhältlich ist. Zurzeit ist die Capella-Version 5.2 aktuell und die älteste offiziell erhältliche Version ist Capella 1.3<sup>2</sup>. Dementsprechend könnten die vorgestellten Entsprechungen nicht mehr zutreffen, da die Modelle der Version 1.1 veraltet sind. Es stellt sich des Weiteren die Frage, dass wenn eine SysML-Repräsentation einer mit Capella erstellten Systemarchitektur benötigt wird, ob mit der SysML-Repräsentation auch gearbeitet und diese modifiziert werden soll. Das Capella-Modell und das daraus gewonnene SysML-Modell überlappen sich im Informationsgehalt und dementsprechend können hier Synchronisierungsprobleme auftreten. Diese können allerdings nicht mit dem von Badache et al. implementierten Übersetzer behoben werden. Wie in dieser Arbeit könnten dafür TGGen als einen Ansatz zum Synchronisieren von Capella-LA- mit SysML-Systemmodellen gewählt werden. Dementsprechend bietet es sich an, in einer zukünftigen Arbeit an das Synchronisieren von Capella- mit SysML-Modellen zu untersuchen.

Das von Batteux et al. [30] vorgeschlagene Framework-Konzept solle in der Lage sein, Capella-PA- mit *AltaRica*<sup>3</sup>-Systemmodellen synchronisieren zu können. Im Gegensatz zu Capella, womit Modelle hauptsächlich zum Entwerfen und Darstellen der System-Architektur erstellt werden, werden AltaRica-Systemmodelle zum Untersuchen der Betriebssicherheit eingesetzt. Die Capella- und AltaRica-Modelle könnten nicht direkt miteinander synchronisiert werden, da diese Modelle nicht direkt vergleichbar seien. Stattdessen sollen abstrakte Systemmodelle, welche mit System Structure Modeling Language (S2ML) spezifiziert sind, erzeugt und diese miteinander synchronisiert werden. [30] S2ML ist eine von Batteaux et al. [31] entwickelte ML, welche das Vergleichen von Systemmodellen in verschiedenen MLs durch Übersetzen zu S2ML ermöglichen solle. S2ML biete eine minimale Menge an Sprachkonstrukten an, mit welchen Systemmodellen anderer MLs abstrakt in einer einheitlichen Art und Weise dargestellt werden könnten. Durch diese Abstraktion und vereinheitlichte Darstellung des Capella- und AltaRica-Modells als S2ML-Modelle solle ein Vergleich der in

<sup>1</sup><https://github.com/eclipse/kitalpha>

<sup>2</sup><https://github.com/eclipse/capella/releases/tag/v1.3.0>

<sup>3</sup><https://www.altarica-association.org/>

den Modellen enthaltenen Informationen möglich sein. Wie in Abbildung 5.1 dargestellt wird, besteht der vorgeschlagene Synchronisierungsprozess aus den drei Schritten Abstrahieren, Vergleichen und Konkretisieren.

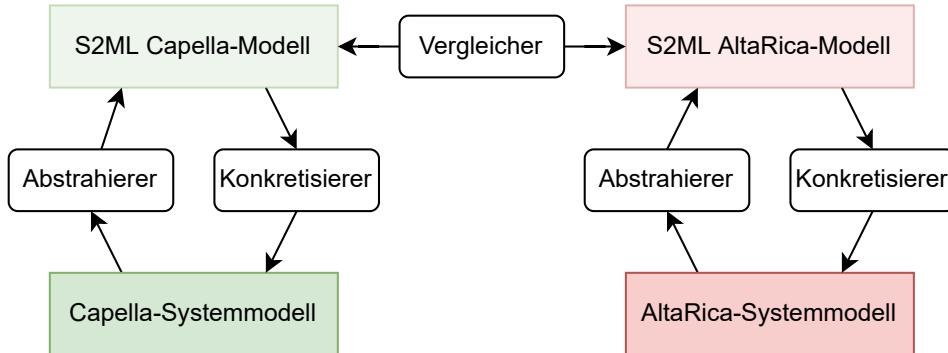


Abbildung 5.1.: Framework-Konzept zum Synchronisieren von Capella- mit AltaRica-Systemmodellen von Batteux et al. [30].

„Abstrahierer“ sollen eingesetzt werden, um das Capella-Systemmodell und das AltaRica-Systemmodell zu S2ML-Modellen zu übersetzen, wobei Information verloren gehen kann. Ein „Vergleicher“ solle anschließend diese S2ML-Systemmodelle miteinander synchronisieren. Die durch den Synchronisierungsprozess vorgenommene Änderungen an den S2ML-Systemmodellen sollen anschließend durch einen „Konkretisierer“ in die Capella- und AltaRica-Systemmodelle propagiert werden, um diese synchron zueinanderzuhalten und womöglich verloren gegangene Informationen wiederherzustellen. Batteux et al. beschreiben eine beispielhafte Implementierung der „Vergleicher“-Komponente und der Modell-Synchronisierung als den im Folgenden beschriebenen iterativen Prozess. Der „Vergleicher“ durchsuche die S2ML-Systemmodelle nach Entsprechungen. Die gefundenen Entsprechungen, und die Modellelemente, für welche keine Entsprechung entdeckt werden konnten, werden in einem Bericht zusammengefasst. Dieser Bericht solle manuell von einer Gruppe an Capella- und AltaRica-Systemmodell-Entwicklern untersucht werden. Dabei sollen fälschlicherweise gefundene Entsprechungen aufgelöst und fehlende Entsprechungen dem Bericht hinzugefügt werden, indem die Entwickler den Bericht editieren. Anschließend wird dem „Vergleicher“ den editierten Bericht überreicht, mit welchem nach Inkonsistenzen zwischen den S2ML-Modellen gesucht werden. Ein Bericht über Inkonsistenzen wird erzeugt, welche von den Entwicklern in den Modellen behoben werden sollen. Wie diese Inkonsistenzen vom „Vergleicher“ entdeckt werden, wird nicht erwähnt. Dieser Synchronisierungsprozess ähnelt stark einem manuellen Synchronisierungsprozess, in welchem Nutzer selbst die Modelle vergleichen und Inkonsistenzen beheben. Dabei wird bei Nutzung des „Vergleichers“ lediglich das Finden von Entsprechungen halb-automatisiert und das Finden von Inkonsistenzen werkzeuggestützt ermöglicht. Da allerdings ganze Entwickler-Gruppen in diesem Synchronisierungsprozess beteiligt sind, kann dieser möglicherweise aufwendig und teuer sein. Es ist zu untersuchen, ob dieses Framework-Konzept von der Nutzung von TGG profitieren kann, indem „Abstrahierer“, „Vergleicher“ und „Konkretisierer“ durch TGG-basierte Modell-Synchronisatoren ersetzt werden. Dafür könnte eine TGG definiert werden, mit welcher Capella-Modelle als Ursprungsmodele und S2ML-Modelle als Zielmodell miteinander synchronisiert werden können. Diese TGG könnte den „Abstrahierer“ und den „Konkretisierer“ für Capella-Modelle ersetzen. Ebenso könnte eine TGG mit einem AltaRica-Modell als Ursprungsmodele und einem S2ML-Modell als Zielmodell den „Abstrahierer“ und den „Konkretisierer“ für AltaRica-Modelle ersetzen. Eine TGG zum Synchronisieren von S2ML- mit S2ML-Modellen könnte als Ersatz für den von Batteux et al. implementierten „Vergleicher“ eingesetzt werden. Alternativ könnte aber auch eine TGG implementiert werden, welche Capella-Modelle direkt mit AltaRica-Modellen

synchronisiert, anstatt eine Kette von TGGen zum Synchronisieren der Modelle zu verwenden. Nicht nur könnte so das Finden von Entsprechungen voll-automatisiert mit Werkzeugen wie eMoflon::IBeX implementiert werden, sondern auch Konflikte möglicherweise automatisch erkannt und aufgelöst werden [2].

Während der Recherche sind einige Arbeiten, welche das Synchronisieren von SysML-Modellen mit Modellen andere MLs mittels TGGen behandeln, aufgefallen. Die ARCADIA-Methode und Capella sind direkt durch SysML inspiriert worden. Es bestehen des Weiteren Ähnlichkeiten zwischen einigen Capella-Sichten und SysML-Diagrammtypen<sup>4</sup>. Unter anderem wirbt Capella damit, dass Capella-Breakdown- und Capella-Interface-Diagramme SysML-Block-Diagrammen entsprechen. Auch seien Capella-Zustandsautomaten nahe mit SysML-Zustandsautomaten verwandt. Dementsprechend könnte es möglich sein, von einer bereits implementierten TGG zum Synchronisieren von SysML-Modellen mit Modellen einer anderen ML eine TGG abzuleiten, welche diese Modelle mit Capella-Modellen anstatt SysML-Modellen synchronisieren kann. Dies könnte in zukünftigen Arbeiten untersucht und womöglich umgesetzt werden. Es folgt eine kurze Vorstellung einiger Arbeiten, in welchen eine TGG für SysML-Modelle erstellt worden ist.

Giese et al. [32] haben einen Synchronisationsprototypen, welcher Systemarchitekturen in Form von SysML-Blockdiagrammen mit AUTOSAR-Softwarearchitekturmodellen synchron zueinander halten könne, mittels TGGen implementiert. **AUTomotive Open System ARchitecture (AUTOSAR)**<sup>5</sup> ist eine Entwicklungspartnerschaft in der Automobilindustrie, welche versucht ihre Software- und System-Architekturstandarte im Automobilbereich weiterzuentwickeln und auf dem Markt als einen einheitlichen Standard durchzusetzen. Laut Giese et al. werden im Automobilbereich SysML-Modelle zur groben Modellierung der Systemarchitektur verwendet. Basierend auf dieser würden AUTOSAR-Modelle der System-Implementierung entworfen werden. Die Verwendung des Synchronisationsprototypen ermögliche das Zusammenspiel verschiedener Entwicklungstätigkeiten und das synchron zueinander halten der Systemmodelle.

Johnson et al. [33] untersuchten das Synchronisieren von SysML-Blockdiagrammen mit Modellica<sup>6</sup>-Modellen mittels TGGen. Die Modellica-ML<sup>7</sup> kann zum Modellieren des physikalischen Verhaltens eines komplexen Systems verwendet werden. Zum Anbieten abstrakter System-Repräsentation eigne sich SysML, Modellica allerdings nicht. Modellica sei zum Darstellen des dynamischen Systemverhaltens geeignet, aber SysML nicht. Dies motivierte Johnson et al. dazu, einen Synchronisationsprototypen mit MOFLON, ein Vorgänger des eMoflon::IBeX-Werkzeugs, zu implementieren, um eine komplementäre Nutzung von SysML und Modellica zur Systembeschreibung zu ermöglichen.

Weidemann et al. [34] untersuchten das Synchronisieren von SysML- und Event-B<sup>8</sup>-Zustandsautomat-Modellen mittels TGGen für die Nutzung im Eisenbahnwesen. Event-B ist eine MBSE-Methode zur Systemmodellierung und Systemanalyse, welche zum Überprüfen von Sicherheitsanforderungen an das System in allen möglichen Konfigurationsmöglichkeiten des Systems verwendet werde [34]. Die mit SysML erstellten Zustandsautomaten könnten zwar mit Simulationen auf Fehler geprüft werden, allerdings sei das Beweisen, dass das beschriebene Systemverhalten den Anforderungen genüge, nicht möglich. Das Beweisen sei allerdings mit Event-B-Modellen möglich, weshalb Entwickler nach dem Entwerfen der Systemarchitektur in SysML ein Event-B-Modell erzeugen müssten. Damit dies nicht mehr manuell vorgenommen werden müsse, wurde ein Synchronisationsprototyp mit eMoflon::IBeX implementiert.

<sup>4</sup>[https://www.eclipse.org/capella/arcadia\\_capella\\_sysml\\_tool.html](https://www.eclipse.org/capella/arcadia_capella_sysml_tool.html)

<sup>5</sup><https://www.autosar.org/>

<sup>6</sup><https://modelica.org/modelicalanguage.html>

<sup>7</sup><https://modelica.org/modelicalanguage.html>

<sup>8</sup>[https://wiki.event-b.org/index.php/Main\\_Page](https://wiki.event-b.org/index.php/Main_Page)

## 6. Zusammenfassung

---

Das Entwerfen einer Systemarchitektur ist wegen der steigenden Komplexität von Systemen nicht trivial. Das Systems Engineering und das Model-Based Systems Engineering bieten Methoden, wie die ARCADIA-Methode, mit welchen Systemarchitekturen so geplant und umgesetzt werden sollen, dass diese der steigenden Komplexität und den steigenden Anforderungen gerecht werden können. Es sind in der Regel viele verschiedene Interessengruppen mit unterschiedlichen Kompetenzen, Zielen und Anforderungen an das System am Entwurf der Systemarchitektur beteiligt. Damit diese Interessengruppen eine für sie relevante Repräsentation des Systems nutzen können, werden im MBSE verschiedene Systemsichten modelliert. Diese Systemsichten können sich im Informationsgehalt überlappen, weswegen diese synchron zueinander gehalten werden müssen, damit das System widerspruchsfrei und vollständig dargestellt wird. Mit dem MBSE-Werkzeug Capella, welches die ARCADIA-Methode implementiert, ist es zwar möglich verschiedene Systemsichten anzulegen, allerdings existieren derzeit keine Möglichkeiten verschiedenen Systemsichten werkzeuggestützt miteinander zu synchronisieren. Dementsprechend kann das Entwerfen einer Systemarchitektur mit Capella aufwendig sein, wenn verschiedene Systemsichten simultan bearbeitet und diese anschließend manuell miteinander synchronisiert werden müssen.

In dieser Arbeit wurde der Ansatz zum Synchronisieren von Capella-Systemmodellen mittels Tripel-Graph-Grammatiken untersucht. Wegen der begrenzten Bearbeitungszeit, Größe und Komplexität der Capella-ML wurde sich dabei auf das miteinander Synchronisieren von Systemmodellen der System Need Analysis mit Systemmodellen der Logical Architecture beschränkt. Es sind TGG-Regeln zum Synchronisieren von Komponenten- und Komponenten-Paket-Modellelementen, welche in Capella-Modellen die Systemelemente und Subsysteme eines Systems repräsentieren, vorgestellt und definiert worden. Die verschiedenen Konstellationen, in welchen Komponenten- und Komponenten-Paket-Modellelemente konsistent zueinander in einem SA- und LA-Systemmodell angeordnet sein können, werden von diesen TGG-Regeln berücksichtigt. Unter anderem wird sichergestellt, dass **SystemComponent**- und **SystemComponentPkg**-Modellelemente im SA-Systemmodell von mindestens einem **LogicalComponent**- oder **LogicalComponentPkg**-Modellelement im LA-Systemmodell realisiert wird. Auch der Sonderfall, dass ein **LogicalComponentPkg**-Modellelement keine Entsprechung im SA-Systemmodell besitzt, wurde berücksichtigt. Ebenfalls wurde das Synchronisieren von Beziehungen zwischen Modellelementen exemplarisch an einer TGG-Regel für Komponenten-Generalisierungsbeziehungen vorgeführt. Die vollständige TGG, welche in Folge dieser Arbeit implementiert worden ist, enthielt des Weiteren Regeln zum Synchronisieren von Funktionen, Funktion-Paketen, Fähigkeiten, Fähigkeit-Paketen und weiteren Beziehungen zwischen diesen Modellelementen.

Basierend auf den definierten TGG-Regeln wurde ein Übersetzungs- und Synchronisationsprototyp mit eMoflon::IBeX implementiert. Mit diesen konnte gezeigt werden, dass das Synchronisieren von Capella-Systemmodellen möglich ist. Da allerdings in dieser Arbeit nicht alle Konzepte der Capella-ML mit dieser TGG abgedeckt wurden, ist dieses Ergebnis nur eingeschränkt gültig. Es bietet sich dementsprechend an, diese TGG in einer zukünftigen Arbeit zu erweitern. Dabei wäre zu überprüfen, ob die Ausdrucksmächtigkeit von TGGen für alle Konzepte der Capella-ML ausreichend ist.

Zur Evaluation des Übersetzungs- und Synchronisationsprototyps wurden synthetische Systemmodelle verwendet, um die Laufzeiten und den RAM-Verbrauch von Batch-Übersetzungs- und Synchronisierungsprozessen zu beurteilen. Dabei ist aufgefallen, dass bei Batch-Übersetzungen Laufzeit und RAM-Verbrauch erwartungsgemäß mit steigender Modellgröße zunehmen. Bei Synchronisierungsprozessen konnte ebenfalls ein Zusammenhang zwischen Modellgröße und Menge an Änderungen mit den Laufzeiten und dem RAM-Verbrauch festgestellt werden. Auffällig war, dass die Synchronisierung von sehr großen Modellen mit über 12000 Modellelementen nach Hinzufügen oder Entfernen eines Elements ca. viermal länger als eine Batch-Übersetzung ähnlicher Modelle benötigte. Des Weiteren schlug eine Synchronisierung von sehr großen Modellen mit über 12000 Modellelementen, nachdem über 500 Modellelemente verschoben wurden, in der Regel wegen unerwartetem Verhalten fehl. Dies könnte vermutlich durch die unzureichende Menge an verfügbaren Arbeitsspeicher der Testumgebung ausgelöst worden sein. Es bietet sich an, dieses Szenario in einer zukünftigen Arbeit zu untersuchen und die Ursachen für dieses Verhalten herauszufinden.

Zuletzt wurde ein Überblick zu themenbezogenen Arbeiten und Ausblicke auf mögliche zukünftigen Arbeiten gegeben. Unter anderem implementierte Badache et al. [29] einen Übersetzungsprototyp, mit welchem ein SysML- aus einem Capella-LA-Systemmodellen erzeugt werden könne. In einer zukünftigen Arbeit könnte eine TGG zum Synchronisieren von LA- mit SysML-Systemmodellen, basierend auf den von Badache et al. vorgestellten Korrespondenzen, implementiert werden. Dadurch könnten des Weiteren Konflikte zwischen Capella- und SysML-Systemmodellen mittels den TGGen erkannt und womöglich werkzeuggestützt behoben werden [2]. Des Weiteren könnte dieser Ansatz zum Synchronisieren von SysML- mit Capella-Systemmodellen das Zusammenspiel von Capella mit verschiedenen SysML-Werkzeugen ermöglichen.

## A. Anhang: Vorgestellte TGG-Regeln zum Synchronisieren von SA- mit LA-Systemmodellen

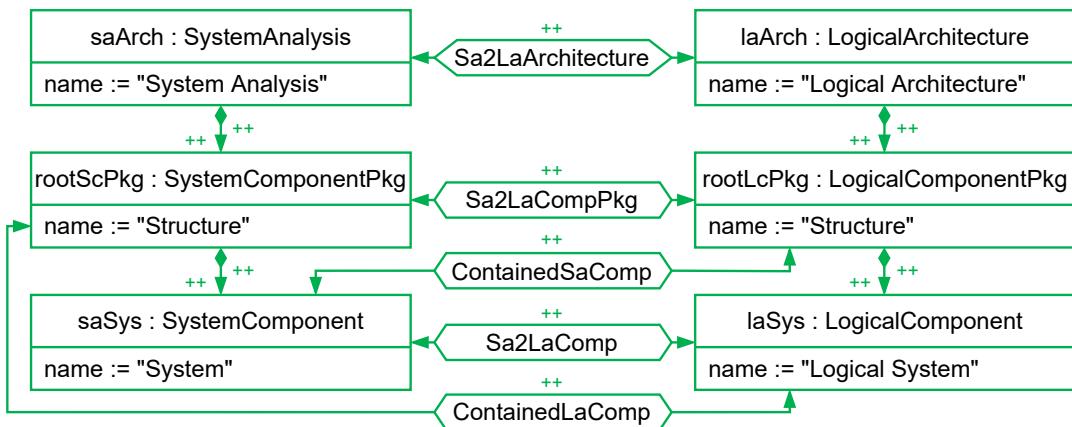


Abbildung A.1.: TGG-Regel zum Erzeugen der SA- und LA-Systemmodelle.

### A.1. Erzeugen von Elementen mit einer Realisierung

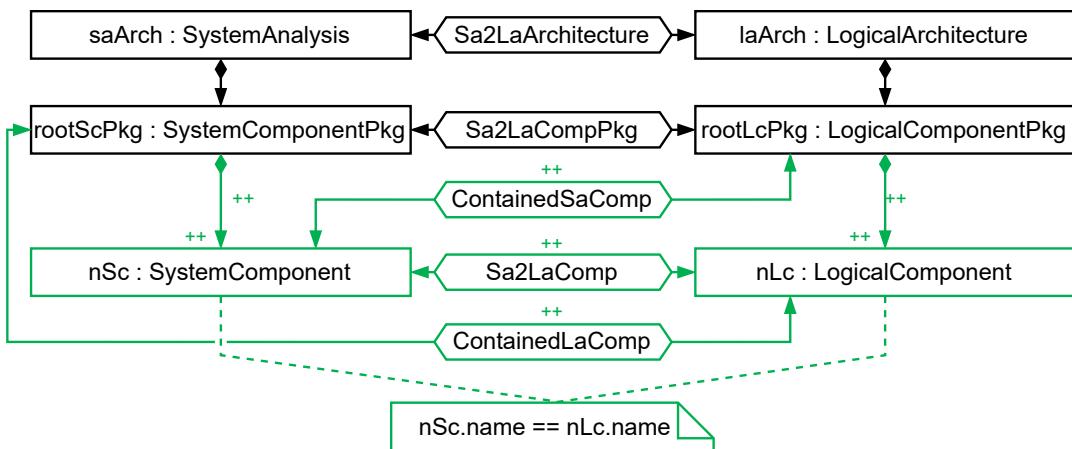


Abbildung A.2.: TGG-Regel zum Erzeugen von Komponenten im Wurzel-Komponenten-Paket.

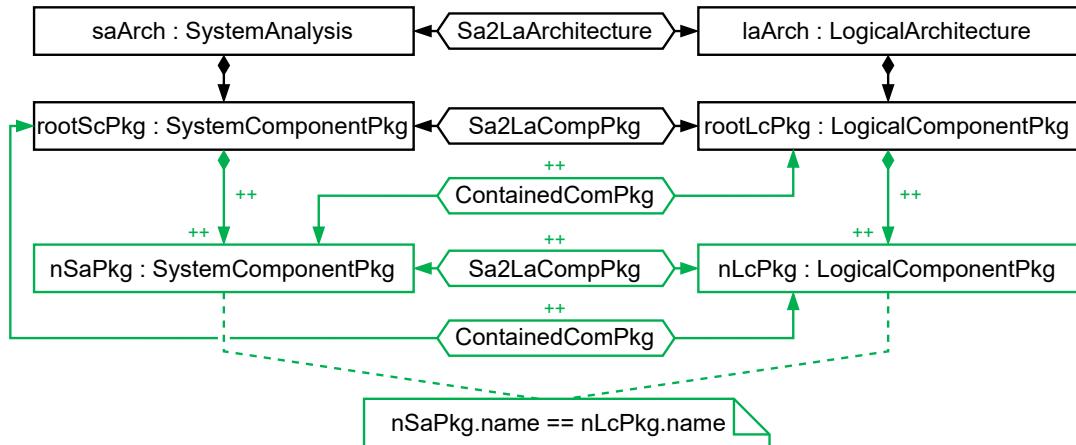


Abbildung A.3.: TGG-Regel zum Erzeugen von Komponenten-Paketen im Wurzel-Komponenten-Paket.

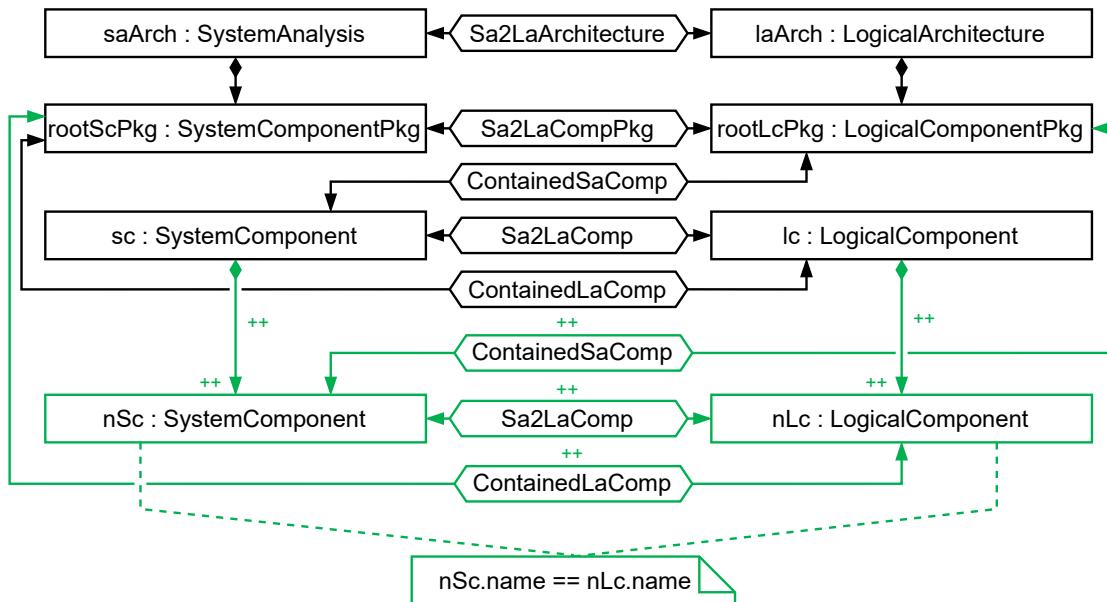


Abbildung A.4.: TGG-Regel zum Erzeugen von Komponenten in Komponenten.

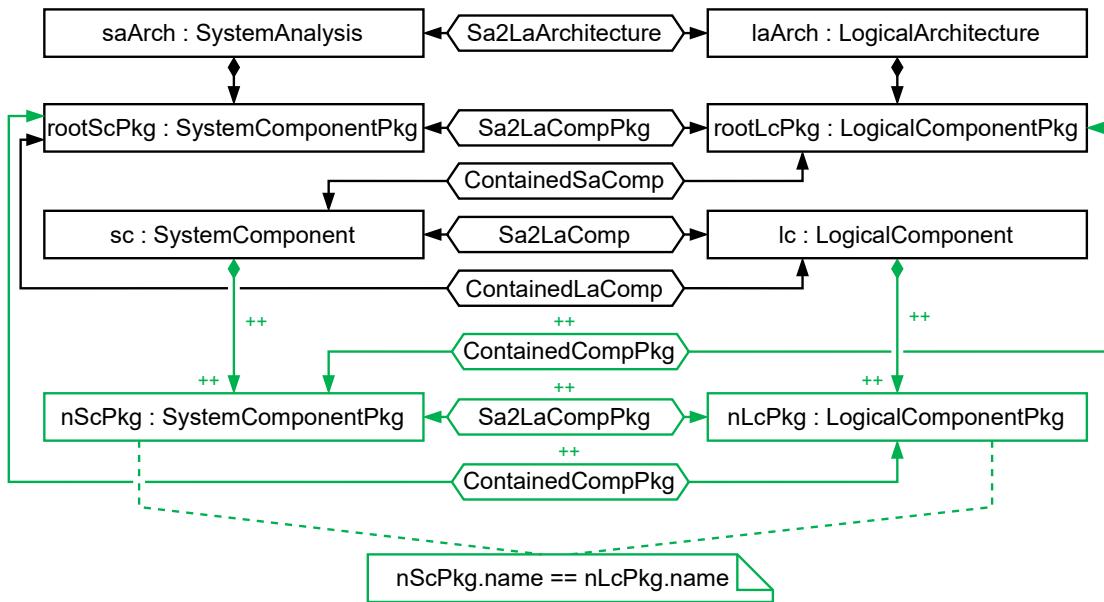


Abbildung A.5.: TGG-Regel zum Erzeugen von Komponenten-Paketen in Komponenten.

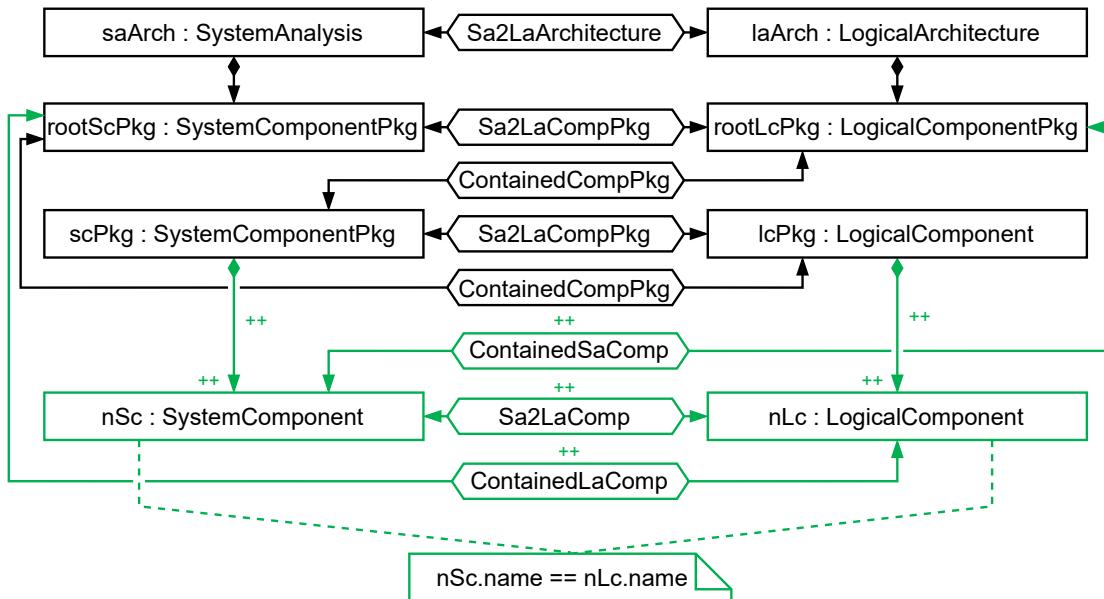


Abbildung A.6.: TGG-Regel zum Erzeugen von Komponenten in Komponenten-Paketen.

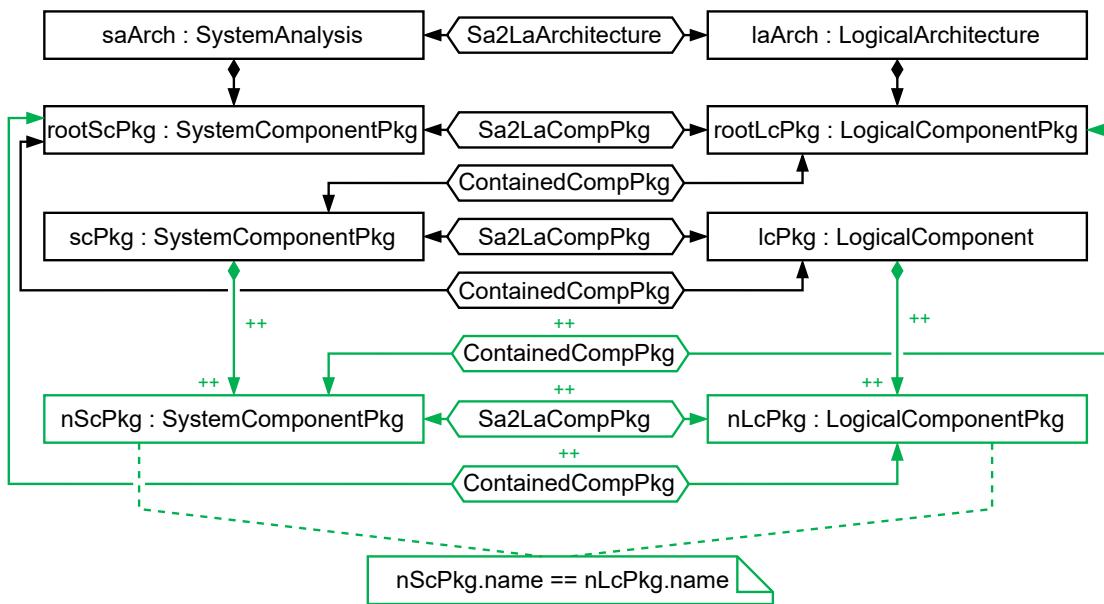


Abbildung A.7.: TGG-Regel zum Erzeugen von Komponenten-Paketen in Komponenten-Paketen.

## A.2. Erzeugen von zusätzlichen Realisierungen

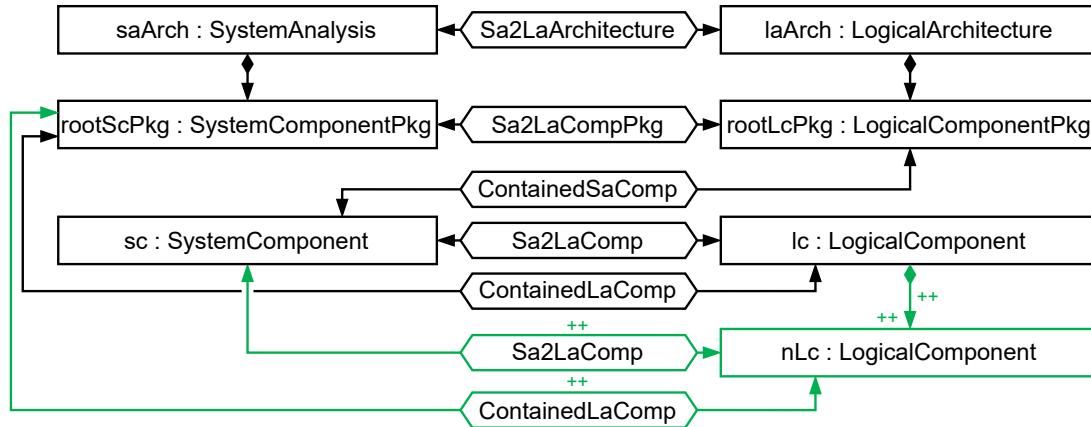


Abbildung A.8.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung in einer existierenden Realisierung.

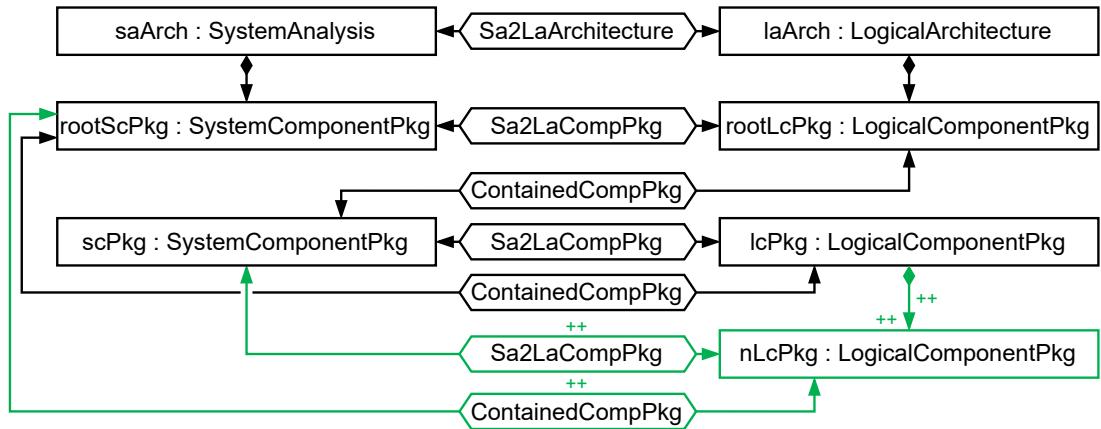


Abbildung A.9.: TGG-Regel zum Erzeugen einer **SystemComponentPkg**-Realisierung in einer existierenden Realisierung.

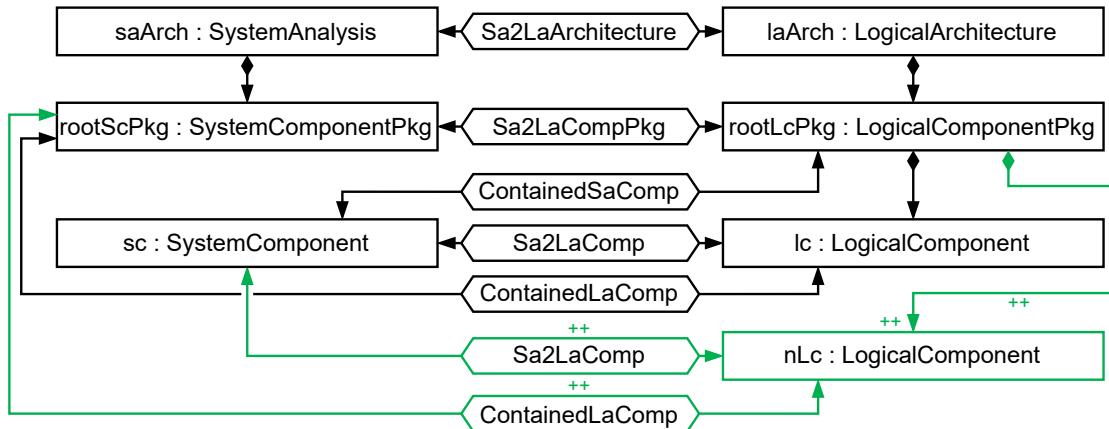


Abbildung A.10.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung im Wurzel-**LogicalComponentPkg**-Modellelement enthalten ist.

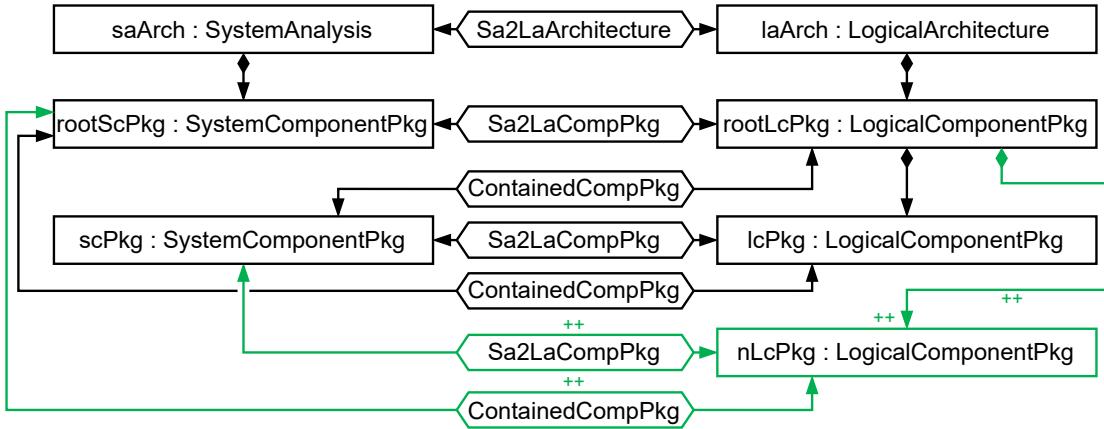


Abbildung A.11.: TGG-Regel zum Erzeugen einer **SystemComponentPkg**-Realisierung, welche gemeinsam mit einer existierenden Realisierung im Wurzel-**LogicalComponentPkg**-Modellelement enthalten ist.

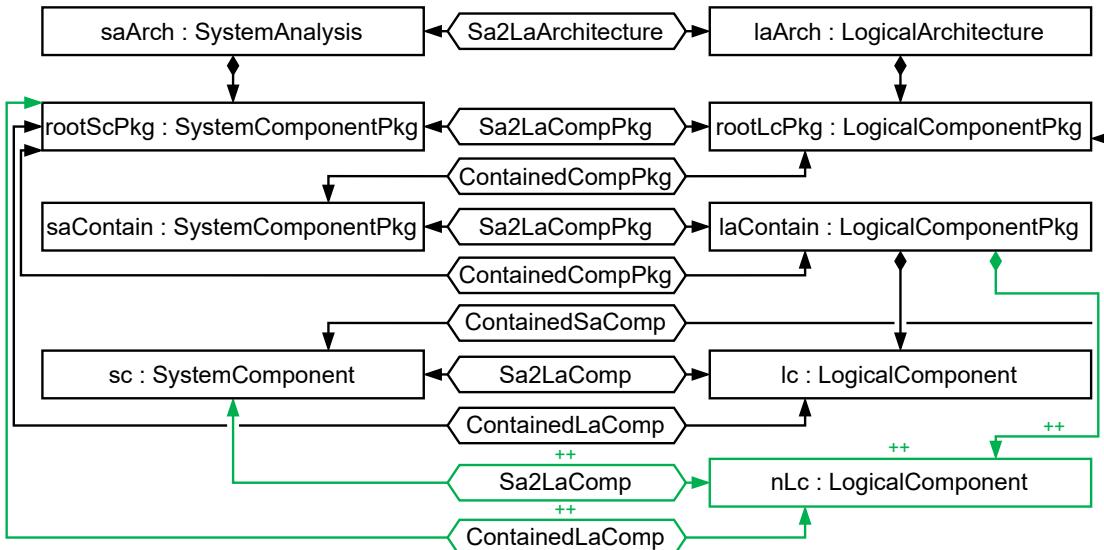


Abbildung A.12.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponentPkg**-Modellelement enthalten ist.

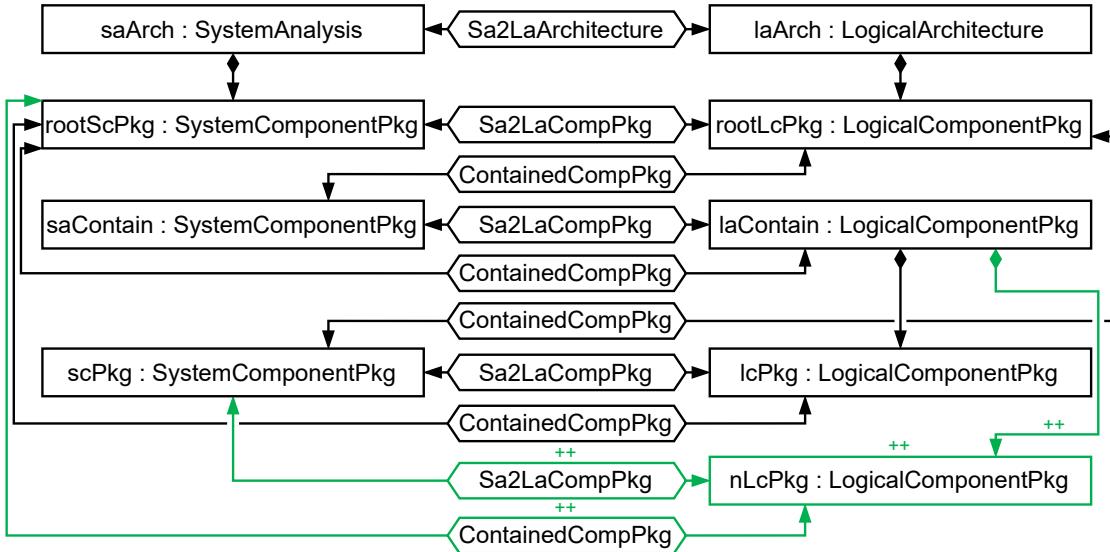


Abbildung A.13.: TGG-Regel zum Erzeugen einer **SystemComponentPkg**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponentPkg**-Modellelement enthalten ist.

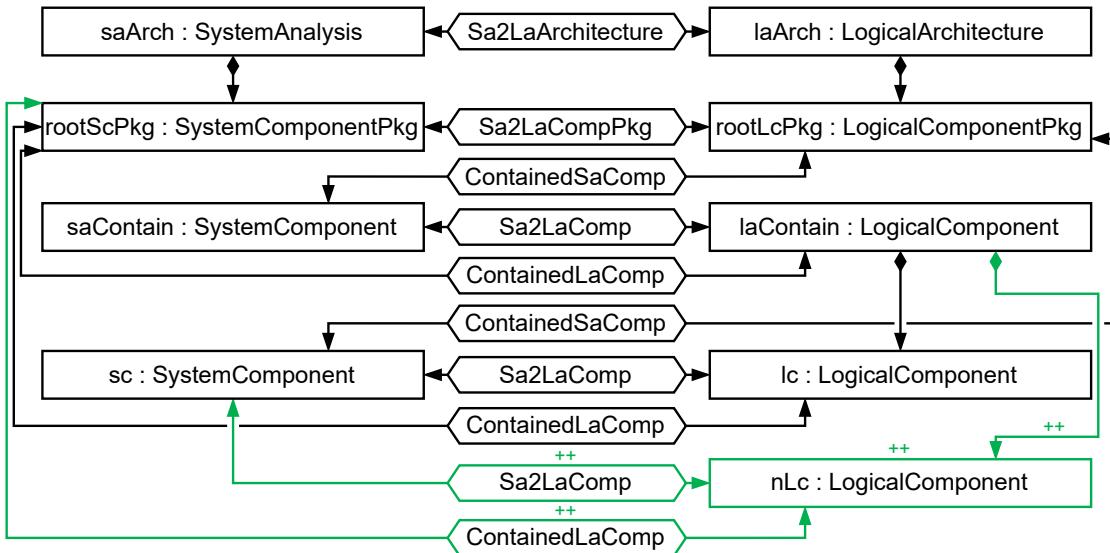


Abbildung A.14.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponent**-Modellelement enthalten ist.

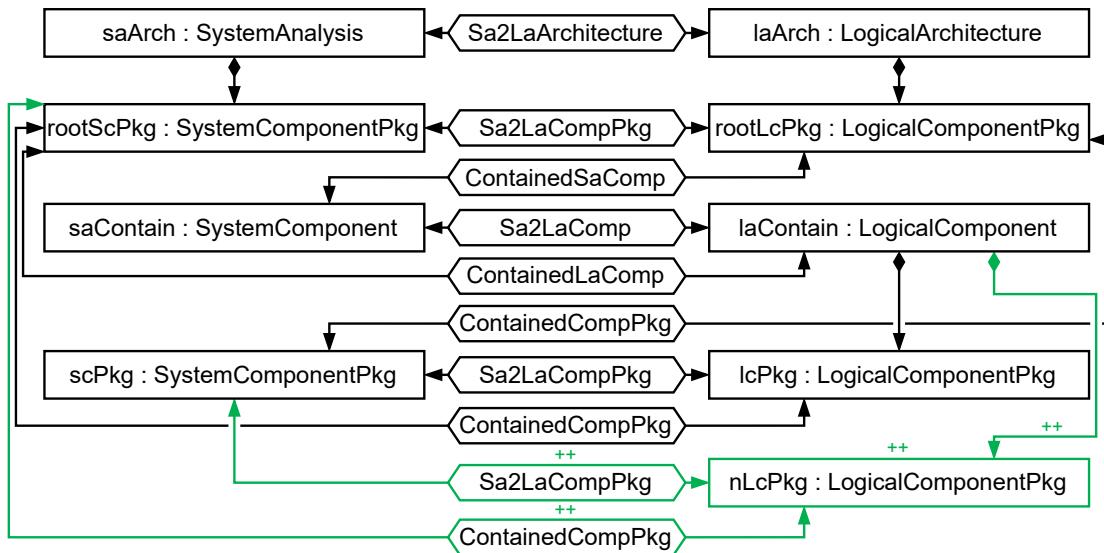


Abbildung A.15.: TGG-Regel zum Erzeugen einer **SystemComponentPkg**-Realisierung, welche gemeinsam mit einer existierenden Realisierung in einem **LogicalComponent**-Modellelement enthalten ist.

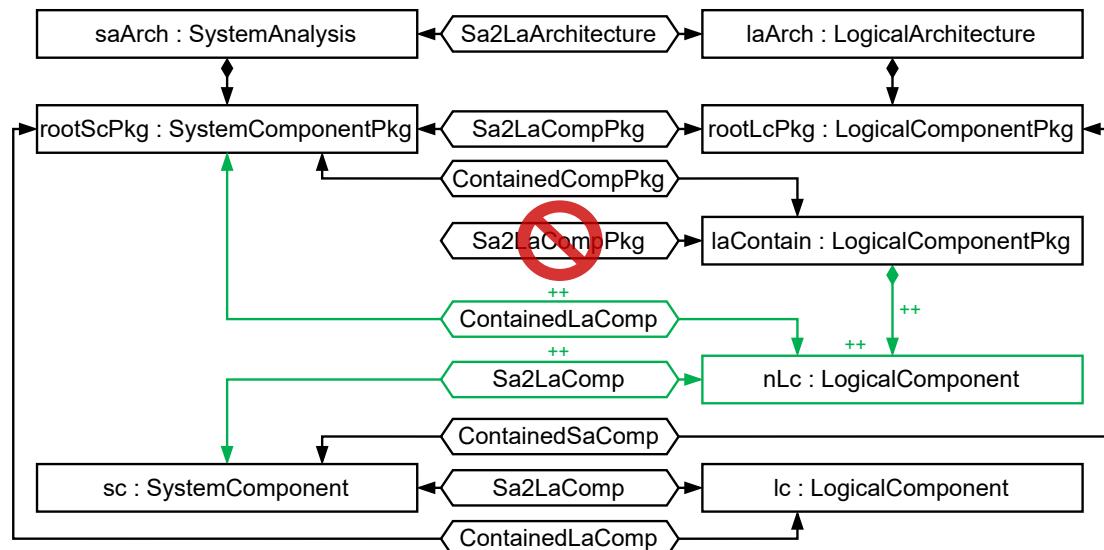


Abbildung A.16.: TGG-Regel zum Erzeugen einer **SystemComponent**-Realisierung in einem **LogicalComponentPkg**-Modellelement, welches keinem **SystemComponentPkg**-Modellelement entspricht.

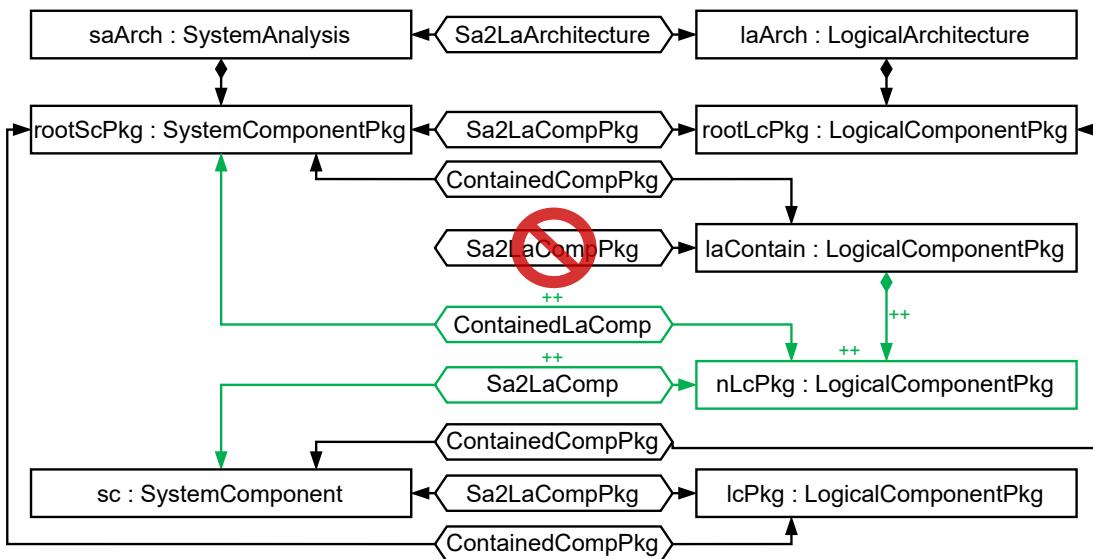


Abbildung A.17.: TGG-Regel zum Erzeugen einer **SystemComponentPkg**-Realisierung in einem **LogicalComponentPkg**-Modellelement, welches keinem **SystemComponentPkg**-Modellelement entspricht.

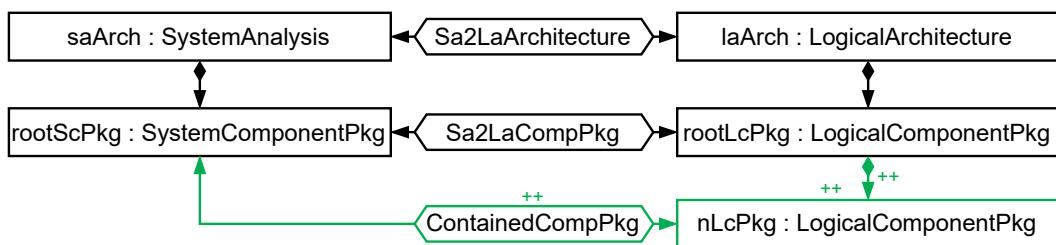


Abbildung A.18.: TGG-Regel zum Erzeugen eines **LogicalComponentPkg**-Modellelements, welches keinem **SystemComponentPkg**-Modellelement entspricht, im Wurzel-**LogicalComponentPkg**.

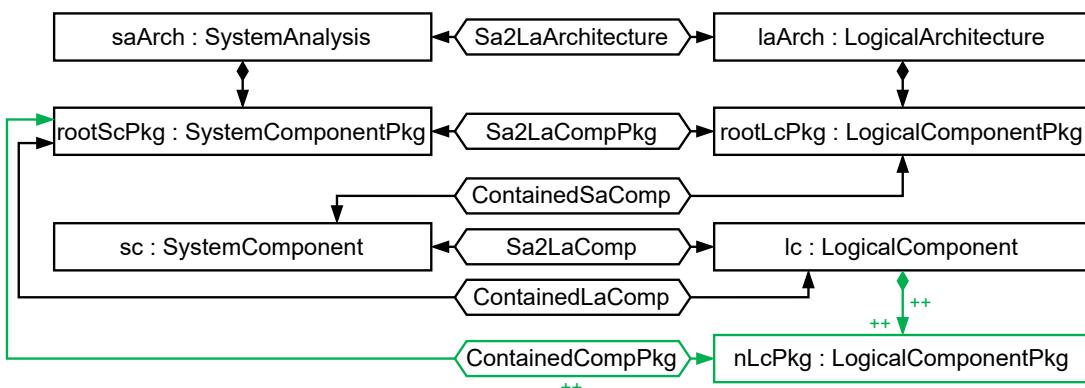


Abbildung A.19.: TGG-Regel zum Erzeugen eines **LogicalComponentPkg**-Modellelements, welches keinem **SystemComponentPkg**-Modellelement entspricht, in einer **SystemComponent**-Realisierung.

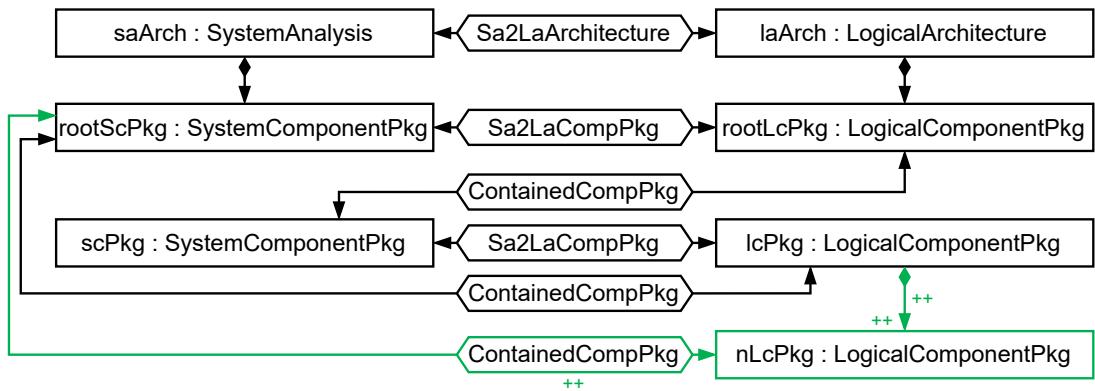


Abbildung A.20.: TGG-Regel zum Erzeugen eines **LogicalComponentPkg**-Modellelementes, welches keinem **SystemComponentPkg**-Modellelement entspricht, in einer **SystemComponentPkg**-Realisierung.

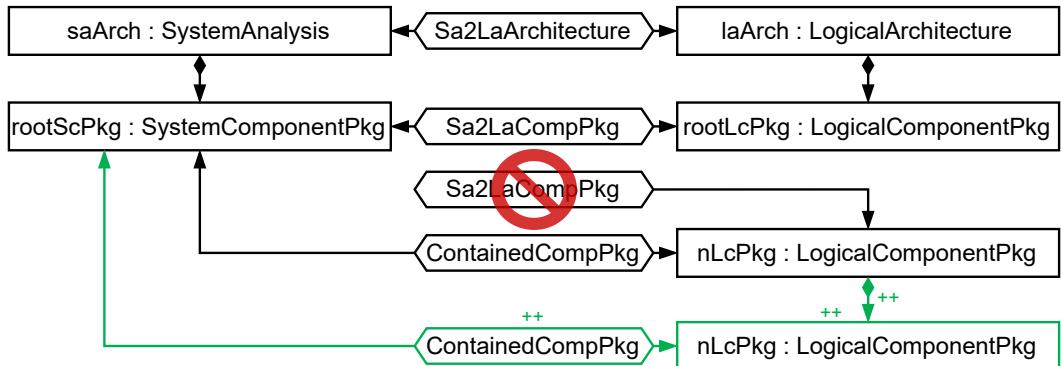


Abbildung A.21.: TGG-Regel zum Erzeugen eines **LogicalComponentPkg**-Modellelementes, welches keinem **SystemComponentPkg**-Modellelement entspricht, in einem **LogicalComponentPkg**-Modellelement, welches ebenfalls keinem **SystemComponentPkg**-Modellelement entspricht.

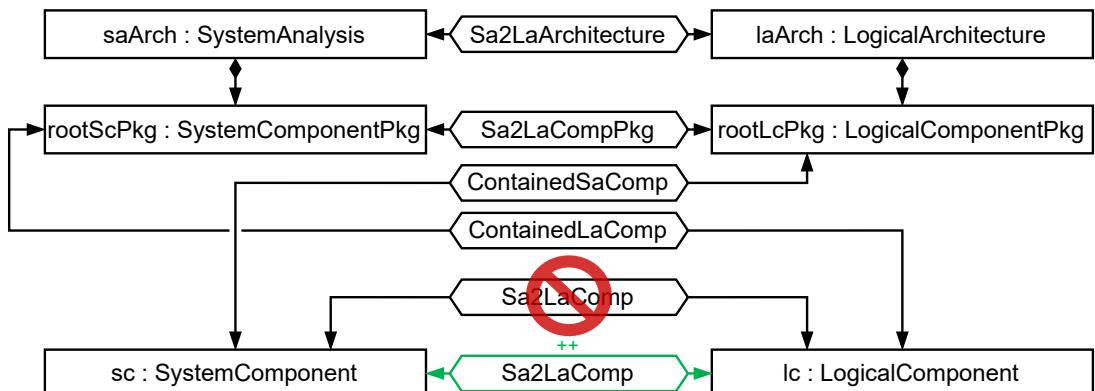


Abbildung A.22.: TGG-Regel zum Erzeugen einer Realisierungsbeziehung zwischen einem **SystemComponent**- und einem **LogicalComponent**-Modellelement, welche sich zuvor nicht entsprochen haben.

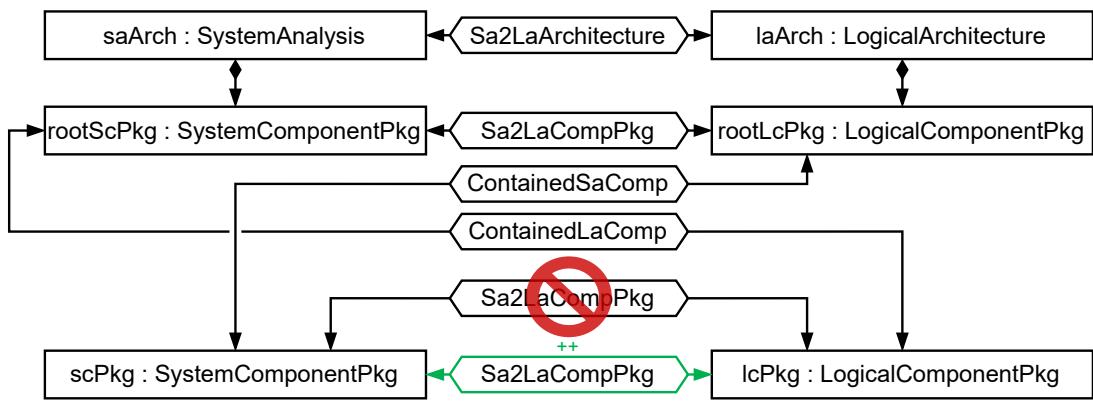


Abbildung A.23.: TGG-Regel zum Erzeugen einer Realisierungsbeziehung zwischen einem **SystemComponentPkg**- und einem **LogicalComponentPkg**-Modellelement, welche sich zuvor nicht entsprochen haben.

### A.3. Erzeugen von Beziehungen zwischen Modellelementen

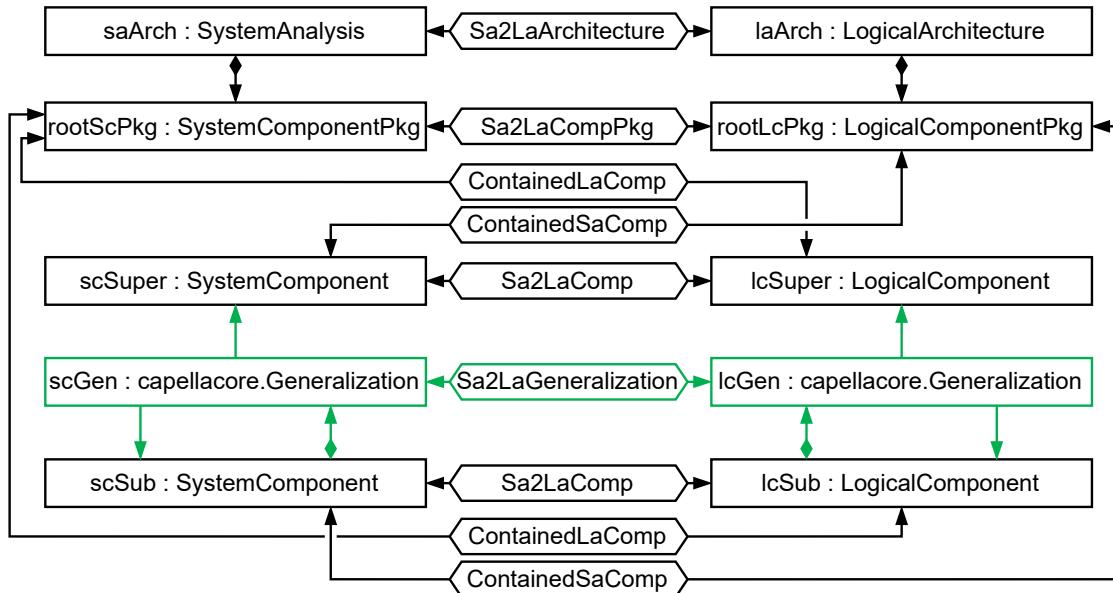


Abbildung A.24.: TGG-Regel zum Erzeugen von Generalisierungsbeziehungen zwischen Komponenten.

## B. Weitere Messergebnisse des Laufzeitverhaltens einer Synchronisierung

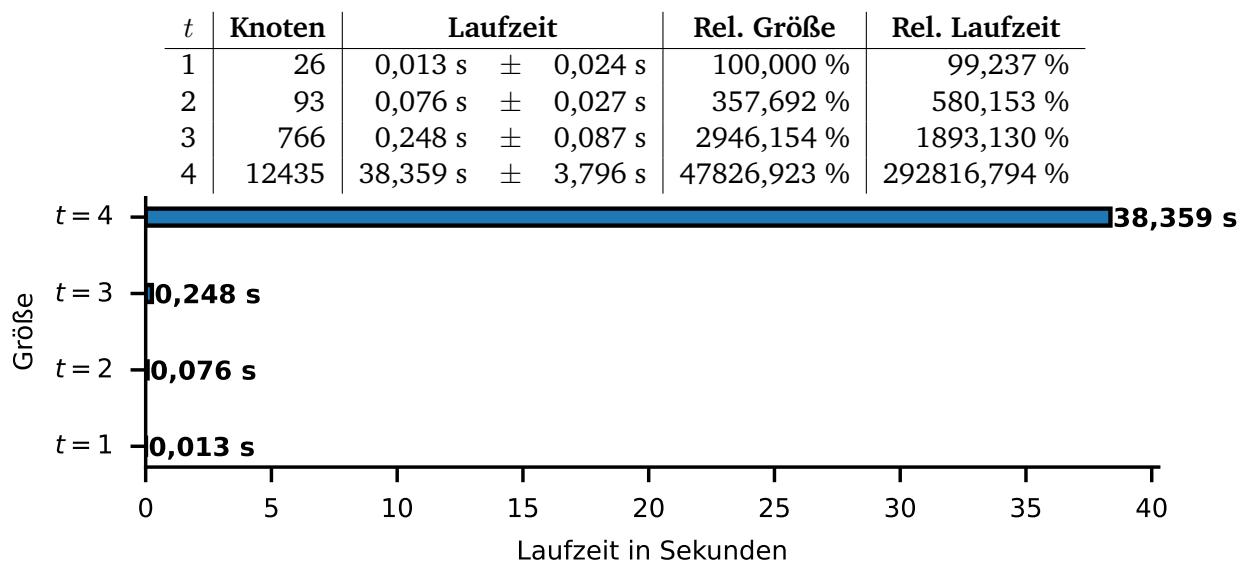


Abbildung B.1.: Laufzeiten einer Synchronisierung von Modellen der Größe  $t$  nach Hinzufügen einer Komponenten-Generalisierungsbeziehung. Als Tabelle und als Balkendiagramm dargestellt.

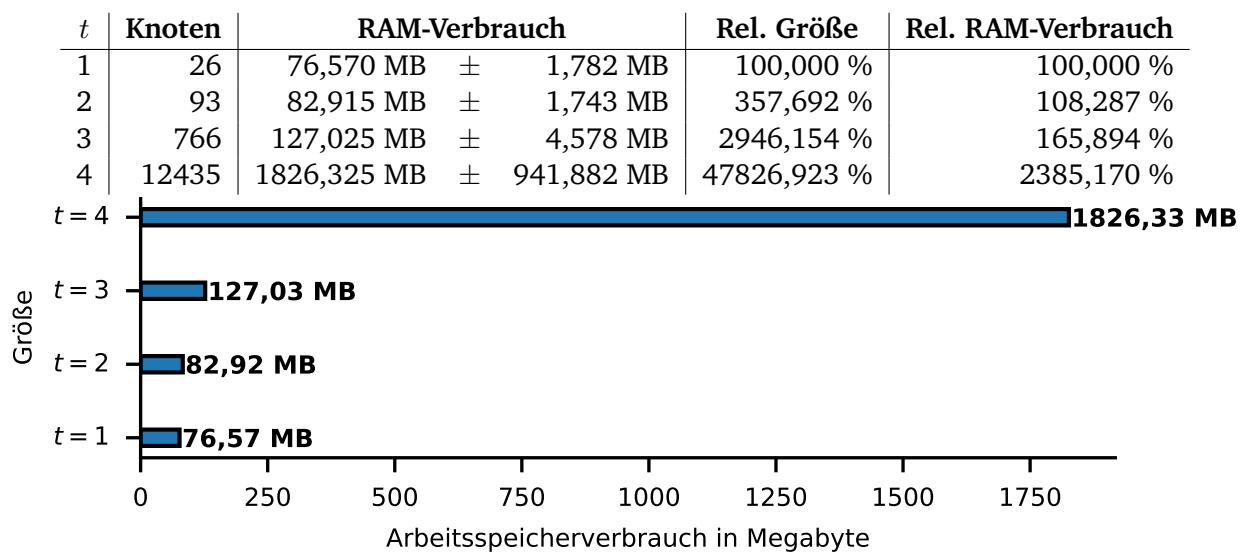


Abbildung B.2.: RAM-Verbrauch einer Synchronisierung von Modellen der Größe  $t$  nach Hinzufügen einer Komponenten-Generalisierungsbeziehung. Als Tabelle und als Balkendiagramm dargestellt.

<b>t</b>	<b>Knoten</b>	<b>Laufzeit</b>		<b>Rel. Größe</b>	<b>Rel. Laufzeit</b>
1	26	0,009 s	± 0,004 s	100,000 %	98,901 %
2	93	0,061 s	± 0,025 s	357,692 %	670,330 %
3	766	0,244 s	± 0,085 s	2946,154 %	2681,319 %
4	12435	37,388 s	± 3,014 s	47826,923 %	410857,143 %

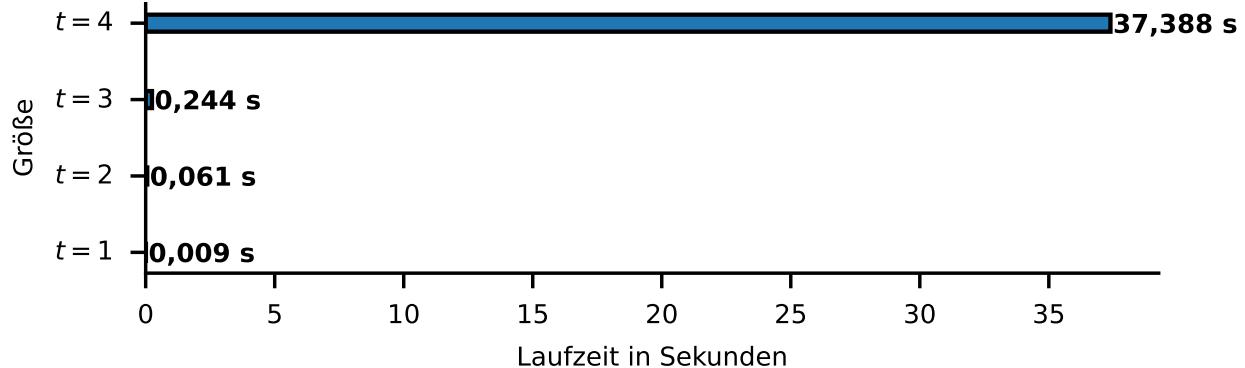


Abbildung B.3.: Laufzeiten einer Synchronisierung von Modellen der Größe  $t$  nach Entfernen einer Komponenten-Generalisierungsbeziehung. Als Tabelle und als Balkendiagramm dargestellt.

<b>t</b>	<b>Knoten</b>	<b>RAM-Verbrauch</b>		<b>Rel. Größe</b>	<b>Rel. RAM-Verbrauch</b>
1	26	83,225 MB	± 2,106 MB	100,000 %	100,000 %
2	93	87,815 MB	± 1,833 MB	357,692 %	105,515 %
3	766	133,415 MB	± 1,372 MB	2946,154 %	160,306 %
4	12435	1820,815 MB	± 914,631 MB	47826,923 %	2187,822 %

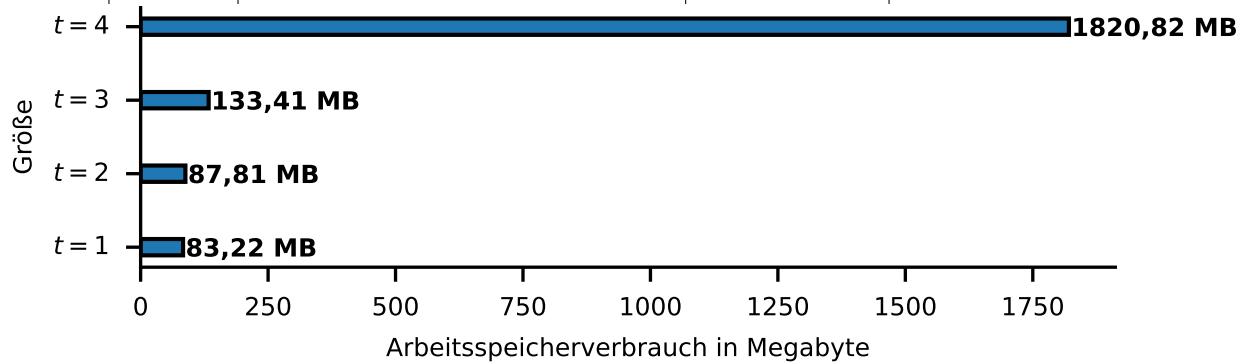


Abbildung B.4.: RAM-Verbrauch einer Synchronisierung von Modellen der Größe  $t$  nach Entfernen einer Komponenten-Generalisierungsbeziehung. Als Tabelle und als Balkendiagramm dargestellt.

<b><i>t</i></b>	<b>Knoten</b>	<b>Laufzeit</b>		<b>Rel. Größe</b>	<b>Rel. Laufzeit</b>
1	26	0,008 s	± 0,004 s	100,000 %	101,266 %
2	93	0,046 s	± 0,018 s	357,692 %	582,278 %
3	766	0,223 s	± 0,048 s	2946,154 %	2822,785 %
4	12435	35,556 s	± 3,244 s	47826,923 %	450075,949 %

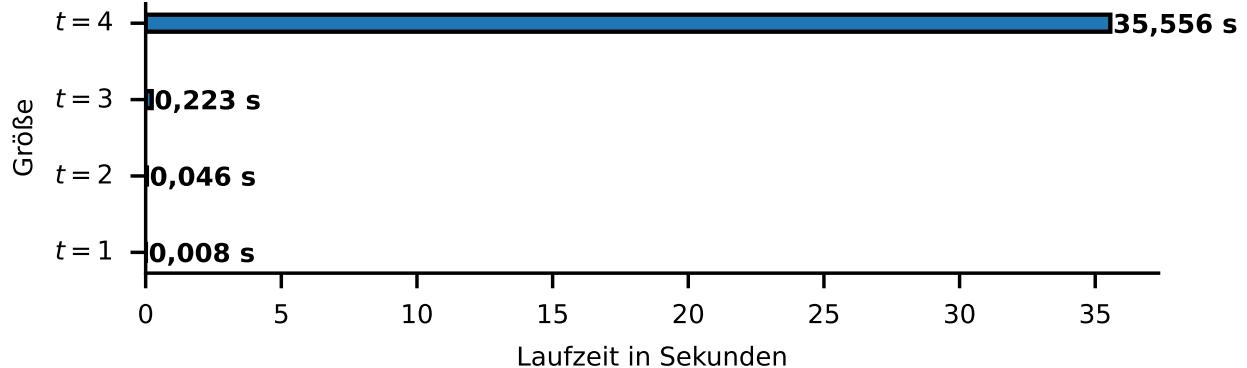


Abbildung B.5.: Laufzeiten einer Synchronisierung von Modellen der Größe  $t$  nach Entfernen einer Komponente. Als Tabelle und als Balkendiagramm dargestellt.

<b><i>t</i></b>	<b>Knoten</b>	<b>RAM-Verbrauch</b>		<b>Rel. Größe</b>	<b>Rel. RAM-Verbrauch</b>
1	26	90,180 MB	± 1,783 MB	100,000 %	100,000 %
2	93	94,315 MB	± 2,041 MB	357,692 %	104,585 %
3	766	138,690 MB	± 1,046 MB	2946,154 %	153,792 %
4	12435	1657,610 MB	± 900,513 MB	47826,923 %	1838,113 %

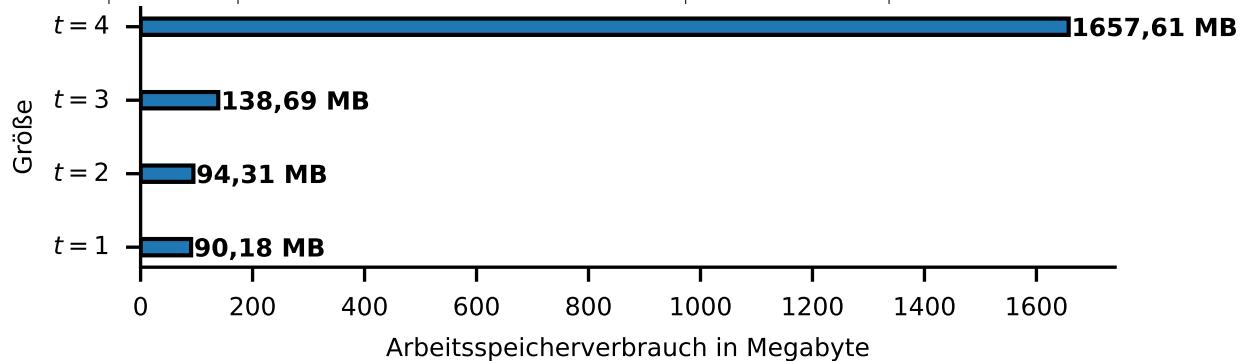


Abbildung B.6.: RAM-Verbrauch einer Synchronisierung von Modellen der Größe  $t$  nach Entfernen einer Komponente. Als Tabelle und als Balkendiagramm dargestellt.

---

## Akronyme

---

**TGG** Tripel-Graph-Grammatik

**SE** Systems Engineering

**MBSE** Model-Based Systems Engineering

**MBE** Model-based Engineering

**MDE** Model-driven Engineering

**MDD** Model-driven Development

**MDA** Model-driven Architecture

**INCOSE** International Council on Systems Engineering

**ISO** International Organization for Standardization

**OMG** Object Management Group

**UML** Universal Modelling Language

**MOF** Meta Object Facility

**ML** Modeling Language

**DSL** Domain-specific Language

**LHS** left-hand side

**RHS** right-hand side

**NAC** Negative Application Condition

**ARCADIA** Architecture Analysis and Design Integrated Approach

**CMOF** Complete Modelling Object Facility

**EMOF** Essential Modelling Object Facility

**EMF** Eclipse Modeling Framework

**Ecore** EMF core

**OA** Operational Analysis

**CTX** Context Architecture

**SA** System Need Analysis

**LA** Logical Architecture

**PA** Physical Architecture

**EPBS** End Product Breakdown Structure and Integration Contracts

**OEBD** Operational Entity Breakdown

**OCB** Operational Capability Blank

**OES** Operational Entity Scenario

**SAB** System Architecture Blank

**LAB** Logical Architecture Blank

**LFBD** Logical Function Breakdown

**LDFB** Logical Dataflow Blank

**LAB** Logical Architecture Blank

**RAM** Random Access Memory (engl. für Arbeitsspeicher)

**SysML** Systems Modeling Language

**S2ML** System Structure Modeling Language

**AUTOSAR** AUTomotive Open System ARchitecture

# Literaturverzeichnis

---

- [1] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 151–163, Berlin, Heidelberg, 1994. Springer.
- [2] Lars Fritzsche, Jens Kosiol, Adrian Möller, Andy Schürr, and Gabriele Taentzer. *A Precedence-Driven Approach for Concurrent Model Synchronization Scenarios Using Triple Graph Grammars*, pages 39–55. Association for Computing Machinery, New York, NY, USA, 2020.
- [3] Charles N. Calvano and Philip John. Systems Engineering in an Age of Complexity. *Systems Engineering*, 7(1):25–34, December 2003.
- [4] Bugra Alkan, Daniel Vera, Mussawar Ahmad, Bilal Ahmad, and Robert Harrison. Complexity in manufacturing systems and its measures: A literature review. *European J. of Industrial Engineering*, 12(1):116 – 150, February 2018.
- [5] International Organization for Standardization. ISO/IEC/IEEE International Standard - Systems and software engineering – System life cycle processes. Technical report, International Organization for Standardization, May 2015.
- [6] INCOSE. SE101: Why do systems engineering? Online, January 2014. Accessed: 2022-01-08.
- [7] John V. Farr. Life Cycle Cost Considerations for Complex Systems. In Boris Cogan, editor, *Systems Engineering*, chapter 5, page 131. IntechOpen, Rijeka, 2012.
- [8] Jeff A. Estefan et al. Survey of Model-Based Systems Engineering (MBSE) Methodologies. *Incose MBSE Focus Group*, 25(8):2–3, 2007.
- [9] James N Martin. *Systems Engineering Guidebook: A Process for Developing Systems and Products*. CRC press, 2020.
- [10] Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle. *INCOSE International Symposium*, 1(1):57–65, 1991.
- [11] Technical Operations INCOSE. Systems Engineering Vision 2020. *INCOSE, San Diego, CA, accessed Jan*, pages 1–32, September 2007.
- [12] Herbert Stachowiak. *Allgemeine Modelltheorie*, pages 128–133. Springer, 1973.
- [13] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*, volume 3. Morgan & Claypool Publishers, March 2017.
- [14] Matthias Bajzek, Johannes Fritz, Hannes Hick, Michael Maletz, Clemens Faustmann, and Gerald Stieglbauer. Model Based Systems Engineering Concepts. In *Systems Engineering for Automotive Powertrain Development*, Powertrain, pages 195–234. Springer International Publishing, Cham, 2021.

- 
- [15] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*, chapter Model-based Systems Engineering, pages 15–29. Morgan Kaufmann, November 2012.
- [16] Object Management Group. OMG Meta Object Facility (MOF) Core Specification. Technical Report 2.5.1, Object Management Group, October 2016.
- [17] Barbara König, Dennis Nolte, Julia Padberg, and Arend Rensink. *A Tutorial on Graph Transformation*, pages 83–104. Springer International Publishing, Cham, 2018.
- [18] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.
- [19] Esther Guerra and Juan de Lara. Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. *Universidad Carlos III de Madrid*, 2006.
- [20] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.
- [21] Marius Lauder, Anthony Anjorin, Gergely Varró, Andy Schürr. Bidirectional Model Transformation with Precedence Triple Graph Grammars. Technical report, Technical University Darmstadt, Darmstadt, 2012.
- [22] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, pages 401–415, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [23] Marius Lauder. *Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars*. PhD thesis, Technische Universität, Darmstadt, November 2012.
- [24] Erhan Leblebici, Anthony Anjorin, Lars Fritzsche, Gergely Varró, and Andy Schürr. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In *Graph Transformation*, pages 179–195, Cham, 2017. Springer International Publishing.
- [25] AFNOR. Technologies de l'information - ARCADIA - Méthode pour l'ingénierie des systèmes soutenue par son langage de modélisation conceptuel - Description Générale - Spécification de la méthode de définition de l'ingénierie et du langage de modélisation XP Z67-140. Technical report, AFNOR, 2018.
- [26] Stephane Bonnet, Jean-Luc Voirin, and Juan Navas. Augmenting requirements with models to improve the articulation between system engineering levels and optimize V&V practices. *INCOSE International Symposium*, 29(1):1018–1033, 2019.
- [27] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
- [28] Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors. *MDE Basics with a DSL Focus*, pages 21–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [29] Nesrine Badache and Pascal Roques. Capella to SysML Bridge: A Tool-up Methodology for MBSE Interoperability. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, TOULOUSE, France, January 2018.
- [30] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Model Synchronization: A Formal Framework for the Management of Heterogeneous Models. *HAL open science*, pages 157–172, 2019.

- 
- [31] Michel Batteux Batteux, Tatiana Prosvirnova, and Antoine Rauzy. System Structure Modeling Language (S2ML). *HAL open science*, December 2015.
  - [32] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. *Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent*, pages 555–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
  - [33] Thomas Johnson, Aleksandr Kerzhner, Christiaan J. J. Paredis, and Roger Burkhardt. Integrating Models and Simulations of Continuous Dynamics Into SysML. *Journal of Computing and Information Science in Engineering*, 12(1), 12 2011. 011002.
  - [34] Nils Weidmann, Shubhangi Salunkhe, Anthony Anjorin, Enes Yigitbas, and Gregor Engels. Automating model transformations for railway systems engineering. *The Journal of Object Technology*, 20(3):10–1, 2021.