Statische Bestimmung von Einstiegspunkten in Aufrufgraphen für Unity-Programme

Bachelorarbeit

Matz Hannek Habermann

Matrikelnummer: 4141896

10. Januar 2023

Fachbereich 3 — Mathematik und Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Prof. Dr. Jan Peleska

ZUSAMMENFASSUNG

Die Unity Spiele-Engine verwendet einige Mechanismen, die von C# Analysetools nicht immer vollständig erkannt werden. Dies führt dazu, dass bei Analysen viele Methoden als toter Code erkannt werden, obwohl sie von Unity aufgerufen werden. Diese Arbeit befasst sich mit den Tools der AXIVION-Suite, mit dem Ziel, die erstellten Aufrufgraphen zu erweiteren, sodass auch Unitys Mechanismen dargestellt werden. Um dies zu erreichen, wurde das Tool Unity2RFG entwickelt, dessen Konzeptionierung und Umsetzung hier vorgestellt werden.

ERKLÄRUNG

Ich versichere, diese Arbeit — sofern dies nicht explizit anders gekennzeichnet wurde — ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Enger, den 10. Januar 2023

Matz Hannek Habermann

GENDER-HINWEIS

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

INHALTSVERZEICHNIS

1 EINFÜHRUNG

In dieser Bachelorarbeit geht es um die Erweiterung von Axivions Aufrufgrafen um bestimmte Mechanismen der Unity-Engine, sodass diese nicht fälschlicherweise als toter Code gemeldet werden. In diesem Kapitel wird dies entsprechend motiviert und die weitere Struktur dieser Arbeit vorgestellt.

1.1 MOTIVATION UND ZIEL

Durch die Art wie Unity implementiert ist, werden viele Methoden von Axivions Dead Code-Analyse als tot gemeldet, obwohl sie durch Unity aufgerufen werden. Dieses Problem tritt auf, da die Dead Code-Analyse einen festen Einstiegspunkt erwartet, der für Unity Projekte allerdings in der Form nicht existiert. Stattdessen werden viele Methoden über Unity spezifische Mechanismen aufgerufen, die von Axivions Dead Code-Analyse nicht erkannt werden, da sie im Aufrufgraphen nicht korrekt abgebildet sind.

Ziel dieser Arbeit soll es sein, den vorhandenen Aufrufgraphen eines Unity Projekts so zu erweitern, damit Axivions Dead Code-Analyse auf diesem modifizierten Aufrufgraphen korrekte Ergebnisse produziert.

1.2 STRUKTUR

Die Umsetzung wird in den nachfolgenden Kapiteln genauer beschrieben. Dafür werden in Kapitel Grundlagen 2 zunächst Unity, Roslyn und die Axivion-Suite vorgestellt, soweit es zum Nachvollziehen dieser Arbeit relevant ist. Im anschließenden Kapitel Konzept 3 wird genauer auf die betrachteten Unity Mechanismen eingegangen. Fokus ist dabei, wie die einzelnen Mechanismen funktionieren und was beachtet werden muss. Das Kapitel Umsetzung 4 stellt das als Teil der Arbeit implementierte Tool Unity2RFG vor. Dies umfasst die allgemeine Struktur des Tools und wie es die vorgestellten Unity Mechanismen behandelt und im Aufrufgraphen darstellt. Hierbei werden auch aufgetretene Probleme sowie alternative Umsetzungsideen beschrieben. Nachfolgend wird im Kapitel Evaluation 5 vorgestellt, wie Unity2RFGs Funktionalität untersucht wurde und welche Ergebnisse gefunden wurden. Abschließend werden die Ergebnisse der Arbeit im Kapitel Fazit 6 noch einmal zusammengefasst und Ideen zur Weiterentwicklung vorgestellt.

2 GRUNDLAGEN

In diesem Kapitel werden kurz die für diese Bachelorarbeit wesentlichen Elemente vorgestellt. Dazu zählen unter anderem einige Funktionen der Unity Spiele-Engine, der C# Compiler Roslyn und Axivions Resource Flow Graph.

2.1 UNITY SZENEN

Szenen können als voneinander abgeschlossene Level betrachtet werden und definieren die Welt, in der ein Spieler sich bewegt. In ihnen ist die Struktur der GameObjects und Komponenten festgelegt. (Seifert und Wislaug, 2017, Kapitel 2.3.4)

2.2 UNITY GAMEOBJECTS

GameObjects sind die Grundbausteine für alle Objekte in Unity. Sie stellen Container für Komponenten dar, durch die ihre Eigenschaften bestimmt werden, haben aber von sich aus keine Funktion. Jedes GameObject erhält von Unity eine eindeutige Id über die es identifiziert werden kann, die sich jedoch Ändern kann, wenn neue GameObjects erstellt werden. Neue Gameobjects können im Unity-Editor oder im Quellcode zu Szenen hinzugefügt werden. Zusätzlich können GameObjects Kinder anderer GameObjects sein, wodurch sie verschiedenen Attribute teilen können, wie z.B. ihre Position in der Szene. Auf dieselben Arten können auch Komponenten zu GameObjects hinzugefügt werden. Eine detailliertere Beschreibung kann in (Seifert und Wislaug, 2017, Kapitel 2.3.5) gefunden werden.

2.3 UNITY KOMPONENTEN

Komponenten bieten die Funktionalität für alle Elemente in Unity. Sie werden meistens dadurch definiert, dass sie von der `MonoBehaviour` Klasse erben, weswegen sie manchmal auch als *Behaviours* bezeichnet werden. Zur Verwendung muss jede Komponente Teil eines Gameobjects sein. Eine Eigenschaft dieser Komponenten ist, dass Unity für alle automatisch bestimmte Methoden aufruft, die in Kapitel 3.2.1 näher beschrieben werden. Weitere Eigenschaften werden in (Seifert und Wis-

laug, 2017, Kapitel 4.3) vorgestellt, sind aber größtenteils nicht relevant für diese Bachelorarbeit.

2.4 UNITY PREFABS

Als Prefabs werden in Unity vordefinierte GameObjects bezeichnet, die im Unity-Editor oder im Quellcode instanziiert werden können. Sie bieten eine einfache Möglichkeit GameObjects wie z.B. einen Baum in einer Szene wiederzuverwenden, ohne es für jede Verwendung neu zusammenbauen zu müssen (Seifert und Wislaug, 2017, Kapitel 15).

2.5 RELEVANTE UNITY DATEIN

Zu jeder Szenen-, Prefab- oder Script-Datei die zum Unity Projekt gehört, existiert eine gleichnamige .meta Datei. Innerhalb dieser Datei ist für jedes Element eine GuID festgelegt, die Unity zu Identifikation verwendet.

Szenen und Prefabs sind als .unity Dateien im YAML Format gespeichert. In diesen Dateien werden alle GameObjects, Komponenten und Prefabs, die sie enthalten festgehalten. Für Komponenten und Prefabs kann über die gegebene GuId die Datei identifiziert werden, in dem sie definiert sind.

2.6 C#

C# ist eine objektorientierte Programmiersprache, entwickelt von Microsoft für .NET Framework Anwendungen. Unity selbst ist zum Teil in C# geschrieben und ist die Hauptsprache, in der Unity Projekte entwickelt werden. Anstatt .NET Framework verwendet Unity jedoch das Mono Projekt, um Plattformen neben Windows zu unterstützen (Seifert und Wislaug, 2017, Kapitel 3.1). Das für diese Bachelorarbeit entwickelte Tool Unity2RFG ist ebenfalls in C# geschrieben.

2.7 DER ROSLYN COMPILER

Als Roslyn wird die Open-Source Implementierung der .NET Compiler Plattform für C# und Visual Basic bezeichnet. Ein entscheidendes Feature von Roslyn ist die Bereitstellung einer umfangreichen API, mit der auf den beim Kompilieren erstellten Syntaxbaum, sowie dazugehörige semantischen Informationen zugegriffen werden kann (Roslyn, 2021). Teil dieser API ist die CSharpSyntaxWalker (2021) Klasse, die als Basis für einen eigenen Visitors verwendet werden kann, um Roslyns Syntaxbaum mit dem Visitor-Pattern zu durchlaufen.

2.8 RESOURCE FLOW GRAPH

Der Resource Flow Graph (im weiteren RFG) ist eine von der Axivion-Suite verwendete Form zur Repräsentation eines Programms. Er verwendet Konten und gerichtete Kanten, um die unterschiedlichen Programmelemente darzustellen. Knoten unterteile sich dafür in verschiedene Typen, wie z.B. Klasse und Methoden. Dasselbe gilt für Kanten. Des Weiteren haben die Typen eine hierarchische Struktur, d.h. die Typen *Explicit_Call* und *Implicit_Call* gehören beide zum darüber liegenden Type *Call*.

RFGs bieten außerdem die Möglichkeit, Informationen in getrennten Views darzustellen. Die Funktion der Views ist es, bestimmte Aspekte eines Programms getrennt voneinander abzubilden. Dabei können Knoten und Kanten in mehreren Views sein. Für diese Arbeit sind die *Assembly View*, die *Code Facts View* und die *Entries View* relevant:

- **Assembly View**: Repräsentiert die Struktur von C# Assemblies.
- **Code Facts View**: Stellt den analysierten Quellcode dar.
- **Entries View**: Enthält Routinen Knoten, die als Einstiegspunkte des Programms gelten.

Zudem enthalten Knoten und Kanten Attribute, die zusätzliche Informationen, wie z.B. den Namen oder die dargestellte Quellcodeposition beinhalten. (Axivion, 2022, Kapitel 3.8)

2.9 AXIVIONS C# ANALYSE

Die Axivions-Suite besitzt in Version 7.5 zwei Tools zur Analyse von C#-Code, CafeSharp und CSharp2RFG. CafeSharp ist das Legacy-Tool und soll in naher Zukunft durch CSharp2RFG ersetzt werden. CSharp2RFG befindet sich noch in aktiver Entwicklung und unterstützt nicht alle Funktionen, die von CafeSharp unterstützt werden. Im Abschnitt CafeSharp oder CSharp2RFG wird weiter auf die Unterschiede zwischen den Tools eingegangen.

3 KONZEPT

In diesem Kapitel sollen die Ziele dieser Arbeit sowie die damit verbundenen Anforderungen genauer erläutert werden. Dies umfasst, welches C# Analysetool der Axivion-Suite unterstützt werden soll, die Unity Mechanismen, die als Ziel dieser Bachelorarbeit behandelt wurden und die allgemeine Darstellung der Unity Mechanismen im RFG.

3.1 CAFESHARP ODER CSHARP2RFG

Die Tools CafeSharp und CSharp2RFG produzieren leicht unterschiedliche RFGs für dasselbe C# Projekt, in der Art wie sie die Elemente des analysierten Projekts darstellen. Ein Beispiel hierfür ist, dass Assembly Knoten mit CafeSharp nicht die Dateiendung „.dll“ mit im Name haben, wärend CSarp2RFG die Dateiendung beibehält. Da Unity2RFG einen existierenden RFG erweitern soll, musste vor Beginn der Entwicklung entschieden werden, ob CafeSharp oder CSharp2RFG unterstützt werden sollten.

CSharp2RFG befindet sich wie bereits beschrieben in aktiver Entwicklung und enthält noch nicht alle Funktionen, die CafeSharp bietet, soll es aber in naher Zukunft ablösen. Für diese Arbeit besonders relevant ist das fehlende *Linkage.Is_Definition* Attribut an Knoten, ohne das die DeadCode-Detection keine sinnvollen Ergebnisse liefert. Abschnitt 4.6.1 behandelt, wie mit dieser Einschränkung umgegangen wurde.

Da CSharp2RFG noch nicht alle Funktionen von CafeSharp unterstützt, wurde Unity2RFG so implementiert, dass es mit RFGs von beiden Tools umgehen kann.

3.2 ZIELE

In diesem Teil werden die Unity Mechanismen vorgestellt, die für diese Arbeit betrachtet wurden.

3.2.1 *Unity Messages*

Als Unity Message werden hier alle Methoden beschrieben, die Unity automatisch in allen Komponenten aufruft. Hierzu zählen z.B. `Start()`, was einmal zu Start der Anwendung ausgeführt wird und `Update()`, dass in jedem Frame erneut aufgerufen wird. Die Sichtbarkeit der

Methoden ist dabei nicht relevant. Entscheidend für den Aufruf ist, dass die Komponente Teil eines aktiven GameObjects in einer Szene ist.

Das Besondere an diesen Methoden ist, dass sie über Reflection von Unitys C++ Code aufgerufen werden, wodurch es nicht möglich ist diese Methoden in der C# `MonoBehaviour` Klasse zu finden. Das heißt, dass es nicht möglich ist anhand des Roslyn Syntaxbaums eines Unity Projekts festzustellen, welche Methoden Unity Messages sind.

Zusätzlich zu den Unity Message abhängig von `MonoBehaviour` gibt es auch Methoden die ausschließlich für Komponenten die von *Netcode*s `NetworkBehaviour` erben aufgerufen werden. Ein Beispiel hierfür ist `OnStartServer()`.

Netcode*: Eine high-level Networking und Multiplayer library für Unity.*

Eine Liste der unterstützten Messages kann in Unitys Dokumentation zur `MonoBehaviour` Klasse gefunden werden (MonoBehaviour, 2021). Für `NetworkBehaviour` existiert eine solche Liste in der Netcode Dokumentation (NetworkBehaviour, 2021).

Methodenaufrufe dieser Art sollen im RFG dargestellt werden, wenn eine Komponente in einer Szene vorkommt.

3.2.2 *Methodenaufrufende Methoden*

Unity bietet eine Reihe von Methoden an, über die andere Methoden aufgerufen werden können, die als Parameter übergeben werden. Die erste Gruppe hier sind `SendMessage()`, `SendMessageUpwards()` und `BroadcastMessage()`, über die Methoden im selben, Eltern oder Kinder GameObjects aufgerufen werden können. Eine weitere Gruppe sind `Invoke()` und `InvokeRepeated()`, die Methoden aufrufen, solange diese für den Aufrufer sichtbar sind. Die letzte betrachtete Methode hier ist `StartCoroutine()`, wodurch die übergebene Methode als *Coroutine* aufgerufen wird.

Coroutine*: Erlaubt es, das eine Methode über mehrere Frames ausgeführt wird.*

Die aufzurufenden Methoden können direkt als Parameter oder „string“ übergeben werden, wodurch alle Methoden mit diesem Namen aufgerufen werden.

Alle Methodenaufrufe über einer der hier aufgeführten Methoden sollen im RFG abgebildet werden.

3.2.3 *C# und Unity Events*

Neben nativen C#-Events besitzt Unity auch ein eigenes Eventsystem, genannt UnityEvents (UnityEvents, 2021), das funktional sehr ähnlich zu den nativen C# Events ist. Beide Arten von Events erlauben es Methoden zu registrieren, die an einem späteren Punkt über `Invoke()` aufgerufen werden. Alle Aufrufe, die über C# Events oder Unity Events erfolgen, sollten im RFG abgebildet werden.

3.2.4 *Game Object Analyse*

Das letzte Ziel ist es, dass die Analyse berücksichtigen soll, ob ein betrachteter Methodenaufruf aus einem GameObject erfolgt, dass tatsächlich Teil einer Szene ist, da Unity die bisher genannten Methoden nur in diesem Fall aufruft. Dies betrifft Szenen und GameObjects die im Unity-Editor erstellt wurden, aber auch GameObjects und Komponenten die zur Laufzeit erstellt oder verändert werden. Konkret ist hiermit die Instanziierung neuer GameObjects oder Prefabs durch Unitys dafür vorgesehenen Methoden und das Hinzufügen von Komponenten zu existierenden GameObjects mittels `AddComponent()` gemeint.

3.2.5 *Darstellung im RFG*

Als Teil der Analyse müssen neue Knoten und Kanten in den gegeben RFG eingefügt werden, um die Unity Mechanismen darzustellen. Dies bezieht sich zum einen auf womöglich fehlende Knoten für die in den Ziele beschriebenen Methoden. Allerdings ist nicht davon auszugehen, dass im RFG Knoten von Methoden fehlen, die tatsächlich im Quellcode existieren. Des Weiteren ist es kein Ziel von Unity2RFG solche Knoten nachträglich einzufügen. Gemeint ist hier, künstliche Knoten hinzuzufügen, um die abgebildeten Unity Mechanismen klarer zu repräsentieren.

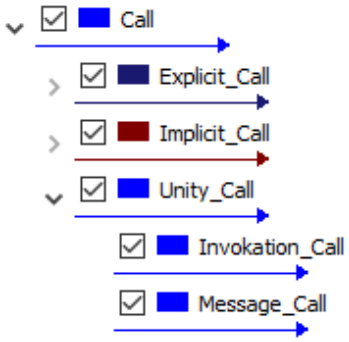

Abbildung 3.1: Aufrufkanten typen um RFG.

Hauptsächlich sollen Aufrufkanten für die betrachteten Unity Mechanismen hinzugefügt werden. Der RFG enthält hierfür den Kantentyp *Call* der in Abbildung 3.1 gezeigt wird. Der Kantentype *Unity_Call* wurde mit den Untertypen *Invokation_Call* und *Message_Call* hinzugefügt, um Aufrufe durch die Unity Mechanismen zu repräsentieren. Diese neuen Kantentypen wurden eingeführt, damit besser nachvollziehbar ist, wann es sich um einen Unity spezifischen Aufruf handelt und woher die Aufrufe durch Unity Mechanismen kommen. Hierbei wird der *Message_Call* Type für Unity Messages verwendet und *Invokation_Call* für Methodenaufrufende Methoden und C# und Unity Events. Die

neuen Kantentypen werden von AXIVIONS Dead Code-Analyse korrekt interpretiert, da sie Kinder vom *Call* Kantentyp sind.

4 UMSETZUNG

In diesem Kapitel werden die Implementierung des entwickelten Tools Unity2RFG, sowie die dabei getroffenen Designentscheidungen vorgestellt. Nachdem kurz die Entscheidung die Programmiersprache C# zu verwenden erläutert wurde, wird anschließend ein Überblick über den grundlegenden Aufbau von Unity2RFG gegeben, bevor die Szenen Analyse und die Analyse und RFG Erweiterung genauer erläutert werden.

4.1 UNITY2RFG IN C#

Unity2RFG wurde hauptsächlich aus zwei Gründen in C# implementiert. Der Erste ist das, wie in Abschnitt 2.7 bereits erwähnt, Roslyn einfachen Zugriff zu einem Syntaxbaum für das analysierte Unity-Projekt bietet, der über die vorhandene C# API analysiert werden kann. Und zweitens, besitzt die Axivion-Suite eine C# API, um auf den RFG zuzugreifen und diesen zu manipulieren.

Alternative wäre auch Python Programmiersprache möglich gewesen, da die Axivion-Suite eine umfangreichere Python API besitzt, im Vergleich zur C# API. Aber wegen der einfachen Benutzbarkeit des Roslyn Syntaxbaums mit C# und keinen besseren Möglichkeiten die GameObject Daten eines Unity-Projekts mit Python zu sammeln wurde die Option ohne größere Nachforschungen zugunsten von C# verworfen.

4.2 AUFBAU

Die wesentlichen Klassen, aus denen Unity2RFG zusammengesetzt ist, sollen hier kurz vorgestellt werden.

- **Programm**: Die zentrale Klasse von Unity2RFG, enthält die Main-Methode und Command Line Optionen. Diese Klasse lädt den RFG und die Konfiguration, kompiliert das angegebe Projekt und startet die Analyse.
- **SceneParser**: Enthält die Statische Szenen-Analyse sowie Funktionen zum Importieren und Exportieren der Szenen-Daten. Außerdem stellt sie der Analyse die gesammelten Szene-Daten zur Verfügung.

- **PreparationWalker**: Visitor-Implemetation für den Roslyn Syntaxbaum. Sammelt die für die Analyse benötigten Daten wie Methoden, die an Events registriert wurden oder im Quellcode erstellte GameObjects.

- **SyntaxWalker**: Der zweite Visitor für den Roslyn Syntaxbaum. Hier wird die tatsächliche Analyse und Erweiterung des RFGs durchgeführt.
- **Element**: Basisklasse für die internen Repräsentationen von GameObjects und Komponenten.
- **BaseGameObject**: Die Basisklasse zur Repräsentation von GameObjects.
- **GameObject**: Repräsentiert GameObjects in der Analyse.
- **SerializedGameObject**: Alternative Repräsentation von GameObjects für den Import und Export der Szenen-Daten.
- **Element**: Repräsentiert Komponenten in der Analyse.
- **RFGUtil**: Stellt RFG bezogene Hilfsmethoden zur Verfügung.
- **MessageSendingMethodCall**: Diese Klasse stellt Aufrufe durch Methodenaufrufende Methoden dar.
- **Signature**: Repräsentation von Methoden Signatur, zur Identifikation von Methodenaufrufenden Methoden.
- **Util**: Enthält Hilfsmethoden für die Analyse.

Zusätzlich wurde auch eine vom Rest unabhängige Unity-Komponente implementiert, die in Anschnitt 4.5.3 behandelt wird.

4.3 KONFIGURATION

Um zu gewährleisten, dass Unity2RFG einfach an neuen Unity Versionen angepasst werden kann, sind ein Großteil der betrachteten Methoden in einer beigefügten Json Konfigurations-Datei definiert. Dies umfasst Unity und Netcode Messages, Methoden zum Aufrufen anderer Methoden, C#- und Unity Events, sowie Methode die GameObjects oder Komponenten zur Laufzeit erzeugen oder manipulieren.

Der Hauptgrund für diese Konfiguration ist, dass im Abschnitt Unity Messages bereits erwähnten Problem, dass die Unity und Netcode Messages nicht aus Unitys C# Quellcode inferiert werden können. Also müssen die existierenden Messages Unity2RFG auf einem anderen Weg bekannt gemacht werden eine Konfigurations-Datei erlaubt hier ein

einfacheres Hinzufügen neuer Messages, die mit neuen Unity Versionen eingeführt werden, als sie direkt im Quellcode festzulegen.

Dieses Schema wurde für die anderen relevanten Methoden beibehalten, da die Konfigurations-Datei bereits existierte und dadurch dieselbe Anpassbarkeit auch für die anderen Ziele gegeben ist.

4.4 GENERIEREN DER ROSLYN SYNTAXBÄUME

Als allerersten Schritt kompiliert Unity2RFG das Unity-Projekt anhand der angegebenen Projektdatei. Sowohl „.csproj“ als auch „.sln“ werden unterstützt. Es verwendet dazu den auf dem System installierten Roslyn Compiler. Wenn mehrere Instanzen gefunden werden, muss interaktiv bestimmt werden, welcher Compiler verwendet werden soll. Über Unity2RFGs Command Line Optionen können Parameter an den Compiler weitergegeben werden.

4.5 SZENEN ANALYSE

Wie im Kapitel Game Object Analyse beschrieben, soll Unity2RFG berücksichtigen, ob eine Komponente in einem GameObject verwendet wird und dieses GameObject Teil einer Szene ist. Um dies umzusetzen werden einige Informationen benötigt, die vor der Analyse gesammelt werden müssen.

1. Für alle Szene die Teil eines Unity Projekts sind müssen alle GameObjects gesammelt werden, die in diesen Szenen vorkommen. Dazu zählen auch GameObjects aus Prefabs oder solche, die zur Laufzeit erst erstellt werden.

2. Für jedes dieser GameObjects ist es nötig zu wissen, welche Kind GameObjects es besitzt und dazu auch implizit das Eltern GameObject.

3. Und es werden alle Komponenten benötigt, die zu einem GameObject gehören. Auch hier werden die Komponenten hinzugezählt, die erst zur Laufzeit hinzugefügt werden.

Die Szenen-Analyse befasst sich mit allen GameObjects und Komponenten, die nicht erst zur Laufzeit erstellt werden. Wie zur Laufzeit instanziierte GameObjects behandelt werden, wird im Abschnitt Game Objekte und Komponenten genauer beschrieben. Leider bietet Unity keine Möglichkeit für ein Programm von außerhalb des Projekts an die oben beschriebenen Daten zu kommen, weswegen zwei unterschiedliche Funktionen implementiert wurden, um dieses Problem zu lösen. Beiden haben jedoch ihre Nachteile und erfordern einen gewissen Mehraufwand vom Benutzer, damit Unity2RFG korrekte Ergebnisse liefern kann.

4.5.1 *GameObject und Komponenten Darstellung in Unity2RFG*

Innerhalb von Unity2RFG werden GameObjects und Komponenten als gleichnamige Klassen repräsentiert, die alle für die Analyse notwendigen Informationen enthalten.

Für GameObjecs beinhaltet diese eine eindeutige ID, die Szene oder das Prefab aus dem dieses GameObject kommt, ob das GameObject Teil einer Szene und/oder eines Prefabs ist, Referenzen zu allen Eltern und Kind GameObject Objekten, sowie Referenzen zu den enthaltenen Component Objekten. Die ID ergibt sich aus dem im Unity-Editor festgelegtem Namen und der von Unity vergebenen eindeutigen ID. Die genaue ID ist für die Analyse nicht relevant, weswegen der Name hinzugefügt wurde, damit ein Benutzer einfacher nachvollziehen kann, um welches GameObject es sich handelt. Der Name alleine würde allerdings nicht als eindeutiges Identifikationsmerkmal ausreichen, da dieser optional ist und nicht eindeutig sein muss.

Die Component Klasse besitzt ebenfalls eine eindeutige ID, sowie Referenzen zu allen GameObject Objekten, die die Komponente enthalten. Was als ID verwendet wird, ist abhängig, wie die Komponenten Informationen gesammelt wurden. Bei der Statische Szenen-Analyse wird die Datei verwendet, da diese Information leicht gefunden werden kann, während die *GameObjectDataCollector* Komponente den qualifizierten Klassennamen verwendet, da dieser hier wiederum leicht zugänglich ist. Dieser Unterschied ist der Grund, wieso die beiden Methoden nicht gemeinsam verwendet werden können.

4.5.2 *Statische Szenen-Analyse*

Die erste Methode ist die statische Szenen-Analyse als Teil von Unity2RFG, die die im Abschnitt Relevante Unity Datein beschriebenen Szenen und Prefab Dateien liest und die benötigten Informationen daraus sammelt. Da diese Dateien das YAML Format besitzen, wird hierfür die YamlDotNet C# Bibliothek verwendet. Zusätzlich werden die „.meta“ Dateiwn gelesen, um referenzierte Komponenten und Prefabs zu identifizieren. Über Unity2RFGs Kommandozeilenoptionen kann hierfür festgelegt werden, welche Ordner nach Szenen-, Prefab- und Script-Dateien durchsucht werden sollen.

Die Unity Szenen- und Prefab-Dateien entsprechen nicht 100% dem YAML Standard und können deswegen nicht einfach so von einem YAML-Parser gelesen werden. Unity2RFG entfernt diese Elemente vor dem tatsächlichen Parsen der Dateien.

Der größte Nachteil an dieser Methode ist, dass Unity keinerlei Informationen über die Beziehungen zwischen den einzelnen GameObjects innerhalb der Szenen- und Prefab-Dateien speichert und es ist von außerhalb des Unity-Projekts nicht möglich diese Information aus einer anderen Quelle zu bekommen, da nicht öffentlich bekannt ist, wie

Unity dies speichert. Diese Informationen sind notwendig, um die im Methodenaufrufende Methoden Abschnitt beschriebene Methoden korrekt zu behandeln. In diesem Fall bleiben einem Benutzer nur die Möglichkeiten fehlende Informationen und daraus resultierende falsche Ergebnissen bei der Dead-Code-Analyse zu akzeptieren, oder die gesammelten Daten als Json zu exportieren, manuell für jedes GameObject die Kinder zuzuordnen und die vervollständigte Json-Datei wieder in Unity2RFG zu importieren. Hinzu kommt, dass dieser manuelle Prozess regelmäßig komplett wiederholt werden muss, da wie in GameObject und Komponenten Darstellung in Unity2RFG beschrieben die von Unity vergebenen IDs für GameObjects nicht stabil sind.

4.5.3 *Unity Komponente*

Als Alternative zur Statische Szenen-Analyse wurde die Unity Komponente *GameObjectDataCollector* implementiert, die in Kombination mit Unity2RFG verwendet werden kann. Die Idee ist, dass die Komponente die benötigten GameObject Daten sammelt, während ein Benutzter sein Projekt im Unity-Editor testet. Da die Komponente selbst Teil des Unity-Projekts ist, kann sie alle relevanten Informationen auslesen und benötigt kein manuelles Nachbearbeiten wie die statische Szenen-Analyse.

Die *GameObjectDataCollector* Komponente ist so implementiert, dass sie beim Start der Anwendung `InvokeRepeating()` aufruft, wo durch die aktuell geladenen Szenen in regelmäßigen Abständen analysiert werden und die Daten von allen in den Szenen vorhandenen GameObjects gesammelt werden. Dabei kann über Felder in der Komponente eingestellt werden, wie häufig die Analyse durchgeführt wird und wie lange nach dem Start der Anwendung gewartet wird für den ersten Durchlauf. Beim Beenden der Anwendung schreibt die Komponente dann die gesammelten Daten in Json Format in die als *OutputPath* angegebene Datei. Durch eine Präprozessor Anweisung wird sichergestellt, dass die Komponente nur Daten sammelt, wenn die Anwendung im Unity-Editor ausgeführt wird.

Zusätzlich enthält die Datei die Klasse *GameObjectData* als Repräsentation der Daten die von der Komponente gesammelt werden. Sie ist identisch zu der *SerializedGameObject* Klasse aus Unity2RFG.

Ein Nachteil ist jedoch, dass das Tool nur aktive Szenen abdeckt, was bedeutet das Informationen fehlen, wenn beim Testen nicht alle Szenen geladen wurden. Dasselbe Problem existiert in kleinem Ausmaß, wenn beim Testen Funktionen ausgelassen werden, die neue GameObjects oder Komponenten erzeugen, entweder einzeln oder in Form von Prefabs. Im zweiten Fall versucht Unity2RFG jedoch noch diese Informationen bei der Analyse des Quellcodes zu erkennen.

Des Weiteren muss ein Benutzer diese Komponente in sein eigenes Unity-Projekt in irgendeiner Form integrieren und seine Anwendung

ausführlich im Unity-Editor durchgehen, um alle Szenen und GameObjects innerhalb seines Projekts abzudecken. Dazu kommt, dass dieser Vorgang regelmäßig komplett wiederholt werden muss, da wie in Abschnitt GameObject und Komponenten Darstellung in Unity2RFG beschrieben, die Unitys IDs nicht stabil sind und sich durch neue oder geänderte Szenen oder GameObjects ändern können. Deshalb ist es nicht möglich, dass spätere Analysen die Daten aus vorherigen Durchläufen wiederverwenden und darauf aufbauen.

4.6 ANALYSE UND RFG ERWEITERUNG

Mit den gesammelten Szenen-Daten analysiert Unity2RFG die generierten Roslyn Syntaxbäume nach den vorgestellten Zielen und erweitert den RFG entsprechend.

4.6.1 *Fehlendes Knoten Attribut mit CSharp2RFG*

In RFGs, die mit CSharp2RFG erstellt wurden, fehlt in Release 7.5 das Attribut *Linkage.Is_Definition* an allen Knoten vom Type Routine. Dieses Attribut wird von AXIVIONS DeadCode-Detection benötigt, denn es werden nur Methoden als tot gemeldet, wenn es am entsprechenden Knoten enthalten ist. Um mit diesem Problem umzugehen gibt Unity2RFG die Option, als Teil der Analyse das Attribut an alle Methoden Knoten anzufügen. Allerdings ist dies eine sehr aggressive Operation, da keine Analyse durchgeführt wird, die entscheidet, an welche Knoten das Attribut angefügt wird, wie es bei CafeSharp der Fall ist.

4.6.2 *Unity Messages*

Die Abdeckung von Unity Messages, wie sie in den Zielen 3.2.1 beschrieben wurden, ist relativ gradlinig umgesetzt. Dabei unterscheiden sich die Messages aus `MonoBehaviour` und `NetworkBehaviour` nur in den betrachteten Methodennamen und der relevanten Basisklasse der Komponenten.

Unity2RFG sucht im Roslyn Syntaxbaum nach Methodendeklaration, die den gleichen Namen haben wie eine der in der Konfiguration unter *UnityMessages* oder *NetcodeMessages* definierten Methoden. Für jede so gefundenen Methodendeklaration wird anschließend geprüft, ob die einschließende Klasse von der `MonoBehaviour` oder `NetworkBehaviour` Klasse erbt und sie Teil eines GameObjects ist, das in einer Szene vorkommt. Sind diese Bedingungen erfüllt, wird eine Kante vom Type *Message_Call* in den RFG eingefügt. Die Abbildung 4.1 zeigt die eingefügte Kante vom Methodenknoten in der `MonoBehaviour` Klasse zum Methodenknoten in der Komponente.

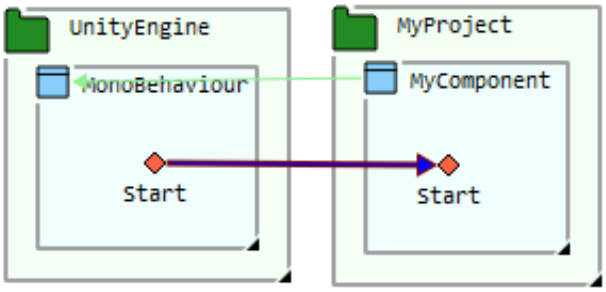

Abbildung 4.1: Darstellung von Unity Messages im RFG.

Hierbei ist anzumerken, dass die Knoten für die Unity Messages in den Klassen `MonoBehaviour` und `NetworkBehaviour` künstlich sind und von Unity2RFG eingefügt werden. Der RFG enthält diese Knoten nicht, da die Unity Messages im C# Quellcode von `MonoBehaviour` und `NetworkBehaviour` nicht existieren. Diese Knoten werden hinzugefügt, um einen einheitlichen Einstiegspunkt für die Unity Messages zu haben. Alternativ wäre es auch möglich jeden Methodenknoten der Unity Messages in Komponenten direkt als Einstiegspunkt zu deklarieren, wenn dieser aufgerufen wird. In diesem Fall müssten keinen neuen Kanten hinzugefügt werden, allerdings ist es so schwerer im RFG nachzuvollziehen, welche Methoden aufgerufen werden und welche nicht, da die Eigenschaft, ob ein Knoten ein Einstiegspunkt ist, im Gegensatz zu verbundenen Kanten nicht direkt sichtbar ist.

4.6.3 *Methodenaufrufende Methoden*

Die Analyse für Methodenaufrufende Methoden besteht aus zwei Schritten. Dabei unterscheidet sich die Behandlung der in Ziel 3.2.2 beschriebenen nur darin, wonach entschieden wird, welche Methoden jeweils aufgerufen werden. In der Konfiguration wird festgelegt, wie welche Methoden behandelt werden, dabei gibt es die folgende Unterteilung:

Im ersten Schritt, also beim ersten Durchlauf über den Roslyn Syntaxbaum, werden Methodenaufrufe der in oben beschriebenen Konfigurationselementen definierten Methoden untersucht. Für jeden Aufruf merkt sich Unity2RFG, welche Methode aufgerufen wird, den Namen der als Parameter übergebenen Methode, die einschließende Methode, in der dieser Aufruf ist und die Komponente aus der Aufruf kommt. Anhand dieser Informationen wird im zweiten Schritt festgestellt, welche Methoden aufgerufen werden.

Beim zweite Durchlauf über den Syntaxbaum, wird nach den Methodendeklarationen für alle Methoden, die im ersten Schritt als aufzurufen identifiziert wurden gesucht. Für jede dieser Methoden wird anschließend geprüft, ob sie tatsächlich aufgerufen wird, nach dem Kriterium, dass über die Konfiguration, wie in Abbildung 4.2 beschrie-

- `MessageSendingMethodsAny`: Die einzige Einschränkung ist, dass die aufgerufene Methode sichtbar ist für die aufrufende Methode.
- `MessageSendingMethodsCurrent`: Die aufgerufene Methode ist aus einer Komponente, die Teil desselben GameObjects ist.
- `MessageSendingMethodsUpwards`: Die aufgerufene Methode ist aus einer Komponente, in demselben GameObject oder einem Eltern GameObject.
- `MessageSendingMethodsDownwards`: Die aufgerufene Methode ist aus einer Komponente, in demselben GameObject oder einem Kind GameObject.

Abbildung 4.2: Konfigurierbare Wirkungsbereiche der Methodenaufrufenden Methoden.

ben, für die aufrufende Methode festgelegt ist. Abbildung 4.3 zeigt die Standard-Konfiguration für alle Methoden, die unter Methodenaufrufende Methoden fallen.

```
"MessageSendingMethodsAny": [
  "Invoke",
  "InvokeRepeating",
  "StartCoroutine"
],
"MessageSendingMethodsCurrent": [
  "SendMessage"
],
"MessageSendingMethodsUpwards": [
  "SendMessageUpwards"
],
"MessageSendingMethodsDownwards": [
  "BroadcastMessage"
],
```

Abbildung 4.3: Standardkonfiguration für die Methodenaufrufenden Methoden.

Im Falle des Aufrufs, fügt Unity2RFG eine *Invoking_Call* Kante von der aufrufenden zur aufgerufenen Methode ein, die in Abbildung 4.4 rot markiert ist. Die aufrufende Methode in der Abbildung ist `SendMessage()` und die aufgerufene `CalledBySendMessage()`. Der Aufruf `SendMessage("CalledBySendMessage")` kommt von der Methode `Start()`. Anzumerken ist hier, dass der Knoten für `SendMessage()` in der Klasse *MyComponent* künstlich ist. CafeSharp und CSharp2RFG fügen automatisch künstliche Knoten ein für aufgerufene Methoden, die in einer Oberklasse definiert wurden. Die Kanten werden ausgehend von diesem künstlichen Knoten eingefügt und nicht von der tatsächlichen Definition der Methode, da so nachvollziehbar ist, wo der Aufruf herkommt.

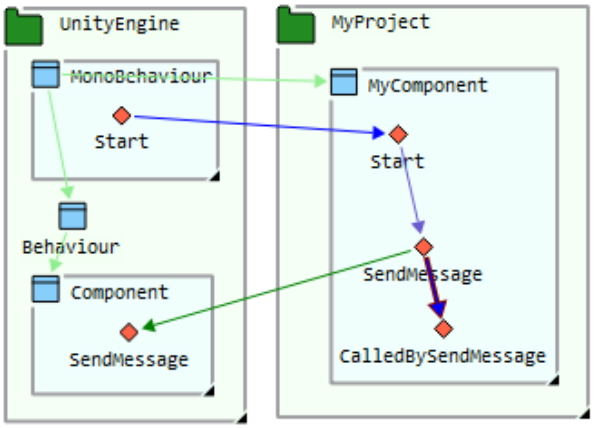

Abbildung 4.4: Darstellung von Methoden aufrufenden Methoden im RFG.

4.6.4 *C# und Unity Events*

Für C# und Unity Events muss zum einen festgestellt werden, welche Methoden alle bei einem Event registriert werden und, ob das Event irgendwann ausgelöst wird. Unity2RFG tut dies mithilfe von zwei aufeinander folgenden Durchläufen über den Roslyn Syntaxbaum.

Im ersten Durchlauf werden die Registrierungen getrennt für jedes Event Objekt gesammelt. Für Unity Events werden Methoden über `AddListener()` registriert, C# verwendet den $+=$ Operator. Der zweite Durchlauf sucht nach `Invoke()` aufrufen auf den gefundenen Event Objekten und fügt *Invokation_Call* Kanten zu allen Methoden ein, die Registriert sind.

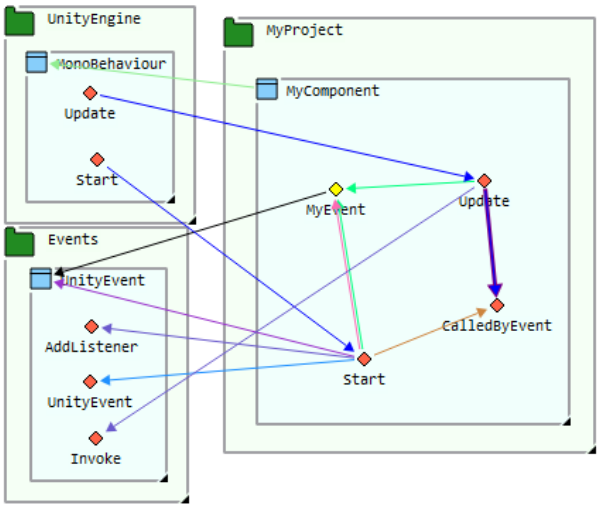

Abbildung 4.5: Darstellung von C#- und Unity Events im RFG.

Die Abbildung 4.5 zeigt an einem Beispiel, wie die Event-Aufrufe im RFG dargestellt werden. Die von Unity2RFG eingefügte Kante geht von `Update()` zu `CalledByEvent()` und ist rot markiert. In diesem

Beispiel wird in der Methode `Start()` an dem Unity Event `MyEvent` mit `AddListener(CalledByEvent)` die Methode `CalledByEvent()` registriert. Anschließend wird `MyEvent.Invoke()` in `Update()` aufgerufen, wodurch `CalledByEvent()` aufgerufen wird.

Für alle Event-Aufrufe wird als Ursprung die einschließende Methode genommen, in der `Invoke()` für das Event aufgerufen wird, anstatt der `Invoke()` Methode selbst. Die Aufrufe werden auf diese Weise dargestellt, da so noch relativ gut nachvollziehbar ist, über welches Event Methoden aufgerufen wurden. Wenn die Kanten von `Invoke()` ausgehen würde, wäre es praktisch unmöglich zu erkennen, welche Kante durch welches Event zustande kommt. Mit der gewählten Darstellung tritt dieses Problem zwar weiterhin auf, wenn mehrere Events in derselben Methode aufgerufen werden, aber die Methode zeigt zusätzlich noch auf, welche Event Objekte zugegriffen wurde.

Auch für Events kann in der Konfiguration festgelegt werden, über welche Methoden zum Registrieren und welche zum Aufrufen verwendet werden.

4.6.5 *Game Objekte und Komponenten*

GameObjects können im Quellcode erstellt oder verändert werden, weswegen Unity2RFG als Teil der Analyse nach GameObject Instanziierungen und neu hinzugefügten Komponenten sucht. Welchen Instanziierungs-Methoden abgedeckt werden, kann über die Konfiguration unter *GameObjectInstantiation* festgelegt werden. Per Default beinhaltet dies `Instantiate()`, `InstantiatePrefab()` und `Load()`. Zuerst versucht Untiy2RFG anhand der Parameter bei der Instanziierung festzustellen, ob das neue GameObject bereits bekannt ist. Wenn dies der Fall ist, wird für das GameObject gesetzt, dass es Teil einer Szene ist. Im Falle eines bisher unbekannten GameObjects, wird ein neues Objekt hierfür erstellt und bei der restlichen Analyse mit berücksichtigt. Für Prefabs ist es wichtig, dass das instanziierte Prefab bei der Szenen-Analyse bereits geparst wurde, da in diesem Schritt nur bereits bekannte Prefabs berücksichtigt werden.

Komponenten können nur über `AddComponent()` zu einem GameObject hinzugefügt werden, weswegen hier nicht konfigurierbar ist, welche Methoden betrachtet werden. Dies ist umgesetzt, indem die als Parameter angegebene Komponente zu allen GameObjects hinzugefügt wird, die die Komponente, die `AddComponent()` Aufruft beinhalten.

Dieser Teil der Analyse hat den größten Einfluss auf die statische Szenen-Analyse, wenn hingegen die *GameObjectDataCollector* Komponente verwendet wird, ist sie nur relevant, wenn der entsprechende Codepfad beim Daten sammeln nicht aufgerufen wurde.

4.7 ALTERNATIVE ANSÄTZE

Unity2RFG wurde als eigenständiges Tool separat von Unity und der Axivion-Suite entwickelt. In diesem Abschnitt sollen kurz andere mögliche Ansätze betrachtet werden.

4.7.1 *Unity2RFG als Komponente*

Unity2RFG wurde so entwickelt, dass es separate zu Unity verwendet werden kann. Es besteht aber auch die Möglichkeit Unity2RFG komplett, als Unity Komponente zu realisieren. Dafür spricht zum einen, dass es wesentlich einfacher wäre, die notwendigen Szenen und GameObejct Daten zu sammeln, da Unity2RFG selbst Teil des zu analysierenden Projekts ist. Des Weiteren gäbe es vielleicht Möglichkeiten andere relevante Informationen direkt aus den GameObjects und Komponenten auszulesen, anstatt dass sie wie im Unity2RFG Tool anhand des Roslyn Syntaxbaumes erkannt werden müssen.

Eine solche Komponente besitzt aber auch einige Nachteile, die Gegen sie sprechen. Der entscheidendste Punkt ist wohl, dass ein Benutzter gezwungen wäre, eine Komponente in sein Projekt einzubauen, die nichts mit seinem eigentlichen Ziel zu tun hat. Dazu kommt eine feste Abhängigkeit auf die C# API der Axivion-Suite. Des Weiteren ist die Integration mit der Axivion-Suite schwierig, da für die Komponente bereits ein RFG existieren muss, aber die Dead-Code-Analyse erst ausgeführt werden soll, nachdem die Komponente seine Analyse abgeschossen hat.

4.7.2 *Integration in CSharp2RFG*

Da sowohl CSharp2RFG und Unity2RFG Roslyn zur Analyse verwenden, ist der Gedanke naheliegend, die Funktionen von Unity2RFG direkt in CSharp2RFG zu integrieren, damit sie hinzugeschaltet werden können, wenn ein Unity-Projekt untersucht wird. Dieser Ansatz löst das Problem, dass Unity2RFG während der Analyse die entsprechenden RFG Knoten zu den betrachteten Knoten aus dem Roslyn Syntaxbaum finden muss. Allerdings ist eine separate Szenen-Analyse weiterhin nötig und dieselben damit verbundenen Problemen, wie in Abschnitt 4.5 beschrieben treten hier auf.

5 EVALUATION

Im Folgenden wird die Funktionalität von Unity2RFG evaluiert. Dazu werden die Ergebnisse der DeadCode-Detection der Axivion-Suite auf einem durch Unity2RFG erweiterten RFG mit den von Unitys eigne Funktionen entfernten Methoden verglichen. Zuerst wird hierzu auf die untersuchten Unity-Projekte eingegangen und erläutert, wieso diese ausgewählt wurden. Anschließend werden kurz Axivions DeadCode-Detection sowie Unitys Managed Code Stripping vorgestellt, die für den Vergleich verwendet wurden. Nach einer kurzen Erläuterung, wie die Evaluation durchgeführt wurde, werden die gefundenen Ergebnisse vorgestellt und diskutier.

5.1 ZIEL

Ziel dieser Evaluation ist es, die Korrektheit und Abdeckung von Axivions DeadCode-Detection auf von Unity2RFG erweiterten RFGs im Vergleich mit Unitys Managed Code Stripping zu überprüfen. Dabei werden die verschiedenen Arten der Szenen-Analyse die Unity2RFG unterstützt sowie mögliche Unterschiede zwischen CafeSharp und CSharp2RFG betrachtet.

5.2 UNTERSUCHTE PROJEKTE

Hier werden die verschieden Unity Projekte vorgestellt, die für die Evaluation untersucht wurden. Alle untersuchten Projekte verwenden mindestens Unity *2020.3*, da dies die älteste Version ist, die zum Zeitpunkt dieser Bachelorarbeit von Unity aktive unterstützt wird. Für alle Projekte beschränkt sich die Evaluation auf den Quellcode des Projekts, der Quellcode der Unity-Engine und weitere Thirdparty-Pakete werden nicht mit berücksichtigt.

5.2.1 *Unity2RFG Test Projekt*

Dieses Projekt wurde als Test für Unity2RFG angelegt und enthält Komponenten für alle Unity Mechanismen, außer für Netcode, die durch Unity2RFG abgedeckt werden. Es bietet dadurch einen Überblick für die abgedeckten Mechanismen, repräsentiert aber in keinster Weise den Aufbau eines realen Unity-Projekts.

5.2.2 *SEE*

SEE ist ein an der Universität Bremen entwickeltes Unity-Projekt zur interaktiven Visualisierung von Software. Es bedient sich dabei der *Code-City*-Metapher und unterstützt mehrere Benutzter auf verschiedenen Plattformen, wie z.B. Desktop und VR.

Code-City: *In der Code-City-Metapher werden Softwarekomponenten durch Gebäude in einer Stadt repräsentiert, wobei die Eigenschaften dieser Gebäude verschiedene Metriken der Software ausdrücken können — z. B. könnte die Höhe eines Gebäudes der Anzahl der Codezeilen entsprechen.*

Die Analyse des aktuellen Quellcodes von SEE mit CafeSharp war leider wegen eines Crashs im CafeSharp Parser nicht möglich. Deswegen wurde für die Evaluation ein RFG aus einer früheren Analyse verwendet. Ungünstigerweise ist für diesen RFG nicht bekannt, mit welcher Quellcode Version er erstellt wurde, weswegen der letzte Commit verwendet wurde, vor dem Erstellen des RFGs. Unterschiede im Quellcode können zu fehlenden Informationen bei der Analyse mit Unity2RFG führen, wodurch die Ergebnisse der Evaluation beeinflusst worden sein könnten.

5.2.3 *Unity Open Project #1: Chop Chop*

Chop Chop ist ein von Unity selbst erstelltes und unterstütztes open source Projekt mit dem Hintergrund, dass Mitglieder der Unity-Community anhand dieses Projekts die verschiedenen Funktionen der Unity Spiele-Engine lernen konnten. Ende 2021 wurde das Projekt eingestellt und wird seit dem nicht weiter entwickelt.

Gewählt wurde dieses Projekt in der Hoffnung, dass, da es ein Lehrprojekt ist, eine breite Menge von Unitys Mechanismen abdeckt werden (ChopChop, 2021).

5.2.4 *Boss Room*

Boss Room ist eine von Unity unterstütztes Beispiel Projekt, das zeigt wie Unitys online Komponente Netcode verwendet werden kann (BossRoom, 2021).

Dieses Projekt wurde hauptsächlich wegen der Verwendung von Netcode ausgewählt. Leider konnte dieses Projekt nicht mit CSharp2RFG analysiert werden, da der Quellcode Elemente enthält, der von CSharp2RFG zum Zeitpunkt des Schreibens dieser Arbeit nicht unterstützt werden.

5.3 UNITY: MANAGED CODE STRIPPING

Mit *Managed Code Stripping* bietet Unity die Möglichkeit beim Bauen des Projekts toten Quellcode zu entfernen, um die Endgröße des gebauten Projekts zu verringern. Dies erfolgt über eine statische Analyse beim Bau des Projekts, die nicht aufgerufenen Quellcode erkennt. Dabei kann über die Option *Managed Stripping Level* eingestellt werden, wie aggressiv

Unity sein soll. Beide Compiler, *Mono* und *IL2CPP*, unterstützten dieses Feature (Managed Code Stripping, 2021).

Bei der Durchführung der Evaluation ist aufgefallen, dass die durch Unitys Managed Code Stripping entfernten Methoden sich nicht groß zwischen den einzelnen Levels unterscheiden. Genauer sind die entfernten Methoden für die verwendeten Compiler und Level Kombinationen *Mono - Medium*, *Mono - High*, *IL2CPP - Medium* und *IL2CPP - High* gleich und in allen anderen betrachteten Kombinationen wurden keine Methoden entfernt. Deswegen wird im Weiteren, wenn von den Ergebnissen aus Unitys Managed Code Stripping gesprochen wird, nicht zwischen Compilern und Stripping Levels unterschieden.

Mono: *Unitys Just-In-Time Compiler basierend auf dem Mono Projekt.*

IL2CPP: *Unitys ahead-of-time Compiler, übersetzt C# zu C++ beim bauen.*

5.3.1 *Axivion-Suite: DeadCode-Detection*

DeadCode-Detection ist der Name für die RFG basierte toter Code Analyse der Axivion-Suite. Sie identifiziert allen Routinen als tot, die nicht über *Call*-Kanten von einem als Einstiegspunkt definierten Knoten erreichbar sind.

5.4 ABLAUF UND DURCHFÜHRUNG

Um die Funktionalität von Unity2RFG zu evaluieren, wurden die Methoden, die von der DeadCode-Detection als tot gemeldet werden, in einem durch Unity2RFG erweiterten RFG, mit denen die von Unitys Managed Code Stripping entfernt wurden, verglichen. Als Methode werden hierbei Klassenmethoden, lokale Funktionen und getter und setter von Properties verstanden.

Hierzu wurde zuerst jedes der untersuchen Projekte mit dem Mono und IL2CPP compiler gebaut, mit den stripping Levels Low, Medium und High, sodass wir für jedes Projekt sechs Versionen erhalten. Zusätzlich wurde zum Vergleich noch eine Version ohne Managed Code Stripping gebaut. Im Weiteren werden von jedem Projekt nur die Assemblies betrachtet, die zum Projekt selbst gehören. Die Assemblies von Unity oder anderen externen Plugins wurden für die Evaluation nicht mit betrachtet. Alle betrachteten Assemblies wurde anschließend mit *.Net Reflector* decompiliert und mithilfe von *DiffMerge* verglichen. Hierdurch wurde erkannt, welche Methoden in den jeweiligen Versionen durch Unitys Managed Code Stripping als tot identifiziert und entfernt wurden.

Zur Analyse mit Axivions DeadCode-Detection wurden zunächst soweit möglich für jedes Projekt zwei RFGs erstellt. Jeweils einer mit CafeSharp und CSharp2RFG. Außerdem wurde für jedes Projekt Szenen-Daten mit der *GameObjectDataCollector* Komponente gesammelt. Dafür wurden jeweils alle Szenen des Projekts im Unity-Editor geladen und die Szenen-Daten gesammelt. Hierbei wurde allerdings nicht jede Funktion eines Projekts ausgeführt.

.Net Reflector: *.Net Reflector ist ein von Red Gate entwickeltes Tool mit dem C# Assemblies durchsucht und dekompiliert werden können.*

DiffMerge: *DiffMerge its ein visuelles Datei und Ordner Diff-Tool von SourceGear.*

Unity2RFG wurde anschließend auf die beiden RFGs eines Projekts angewendet. Dabei einmal mit den Szenen-Daten, die mithilfe der Unity2RFG als Komponente gesammelt wurden und ein weiteres Mal mit der Statische Szenen-Analyse. Sodass für jedes Projekt sechs RFGs erstellt wurden, die mit AXIVIONS DeadCode-Detection untersucht wurden.

Am Ende wurde dann verglichen welche Methoden von AXIVIONund welche von Unity als tot identifiziert wurden.

Für die Laufzeit- und Speicheranalyse wurde Unity2RFG für jedes Projekt jeweils mit der Statische Szenen-Analyse, den Szenen-Daten von der Unity2RFG als Komponente und ohne Szenen-Analyse laufen lassen. Die Laufzeit wurde mit dem Measure-Command (2023) Befehlt gemessen. Für den Speicherverbrauch wurde der Performance Monitor verwendet. Beide Tool sind Teil vom Windows Betriebssystem. Die *GameObjectDataCollector* Komponente selbst wurde nicht untersucht, da Laufzeit und Speicherverbrauch für sie stark davon abhängig sind, wie das Unity-Projekt manuell getestet wird.

5.5 ERGEBNISSE UND DISKUSSION

Im Folgenden werden die bei der Evaluation gefundenen Ergebnisse zu Laufzeit, Speicherverbrauch und gefundenen toten Methoden vorgestellt.

Die Tabellen in diesem Abschnitt unterteilen die Ergebnisse häufig in die Spalten *Komponente*, *Parser* und *Alle*. Dies bezieht sich auf die Art der Szenen-Analyse, die durchgeführt wurde. *Komponente* heißt, dass Szenen-Daten mit der *GameObjectDataCollector* Komponente verwendet wurden. Mit *Parser* ist gemeint, dass die Statische Szenen-Analyse zum Einsatz kam. Und *Alle* bedeutet, dass keinen Szenen-Analyse durchgeführt wurde, sondern stattdessen für alle Komponenten angenommen wird, dass sie Teil einer Szene sind.

Im beigefügten USB-Stick können im Verzeichnis *Evaluation* die Quelldateien sowie die für die Evaluation erzeugten Rohdaten für alle untersuchten Projekte, mit Ausnahme von SEE, gefunden werden.

5.5.1 *Laufzeit und Speicherverbrauch*

Die Tabellen 5.1 und 5.2 zeigen die Laufzeiten von Unity2RFG für die verschiedenen Projekte für CafeSharp und CSharp2RFG. Sie zeigen, dass es *Komponente* und *Alle* in ihrer Laufzeit sehr nah beieinander liegen und *Parser* wesentlich länger läuft. Da *Parser* die einzige Variante ist, bei der die Szenen-Analyse mit ausgeführt wird, ist dies nachvollziehbar.

	Laufzeit in Sekunden		
Projekt	Komponente	Parser	Alle
Boss Room	18,61	24,86	16,84
Chop Chop	29,8	49,97	31,85
SEE	27,59	166,57	26,94
TestProjekt	9,27	9,33	9,21

Tabelle 5.1: Laufzeit in Sekunden Unity2RFG mit CafeSharp RFGs.

	Laufzeit in Sekunden		
Projekt	Komponente	Parser	Alle
Chop Chop	21,91	41,53	10,12
SEE	23,58	163,68	24,97
TestProjekt	4,89	4,82	4,75

Tabelle 5.2: Laufzeit in Sekunden von Unity2RFG mit CSharp2RFG RFGs.

Der Speicherverbrauch zeigt ein ähnliches Bild. Sowohl der durchschnittliche Speicherverbrauch in den Tabellen 5.3 und 5.4 als auch der maximale Speicherverbrauch zusehen in 5.5 und 5.6 sind für *Parser* wesentlich höher als für *Komponente* und *Alle*. Die anderen haben auch hier sehr ähnliche Werte.

	Durchschn. Speicherverbrauch in MB		
Projekt	Komponente	Parser	Alle
Boss Room	967,25	1.054,94	975,47
Chop Chop	798,25	970,1	787,94
SEE	1.215,39	1.834,43	1.217,53
TestProjekt	655,98	668,13	670,97

Tabelle 5.3: Durchschnittlicher Speicherverbrauch in MB von Unity2RFG mit CafeSharp RFGs.

	Durchschn. Speicherverbrauch in MB		
Projekt	Komponente	Parser	Alle
Chop Chop	535,72	726,16	521,46
SEE	823,77	1.228,83	832,07
TestProjekt	295,62	291,11	304,66

Tabelle 5.4: Durchschnittlicher Speicherverbrauch in MB von Unity2RFG mit CSharp2RFG RFGs.

Unabhängig von Unity2RFG fällt auf, dass die Analysen auf den CafeSharp RFGs bei allen Projekten mehr Speicher benötigt haben als auf den RFGs von CSharp2RFG. Diese ist sehr wahrscheinlich darauf zurückzuführen, dass die CafeSharp RFGs größer sind als die von CSharp2RFG.

	Max. Speicherverbrauch in MB		
Projekt	Komponente	Parser	Alle
Boss Room	1.240,7	1.271,71	1.239,28
Chop Chop	982,55	1.518,65	971,02
SEE	1.520,03	1.890,63	1.514,49
TestProjekt	825,04	830,54	828,17

Tabelle 5.5: Maximaler Speicherverbrauch in MB von Unity2RFG mit CafeSharp RFGs.

	Max. Speicherverbrauch in MB		
Projekt	Komponente	Parser	Alle
ChopChop	656,12	1.125,48	633,65
SEE	1.025,38	1.320,85	1.045,8
TestProjekt	384,34	385,51	380,65

Tabelle 5.6: Maximaler Speicherverbrauch in MB von Unity2RFG mit CSharp2RFG RFGs.

5.5.2 *DeadCode-Analyse*

Als Nächstes werden die Ergebnisse von Axivions DeadCode-Detection und Unitys Managed Code Stripping vorgestellt und verglichen. Dafür werden zuerst die verschiedenen Arten von Unity2RFGs Szenen-Analyse betrachtet.

Projekt	Komponente	Parser
Boss Room	2323	37
Chop Chop	3120	521
SEE	3.695	32.604
TestProjekt	53	58

Tabelle 5.7: Anzahl der Gefundenen GameObjecte von der Statischen Szenen Analyse und der *GameObjectDataCollector* Komponente.

Die Tabelle 5.7 zeigt, wie viele GameObjects durch die statische Szene-Analyse und die *GameObjectDataCollector* Komponente in jedem Projekt gefunden wurden. Es ist zu sehen, dass die Methoden sehr unterschiedliche Mengen an GabeObjects gefunden haben. Dies ist darauf zurückzuführen, welche Ordner für die statische Szenen Analyse durchsucht wurden und wie ausführlich das Projekt mit der *GameObjectDataCollector* Komponente getestet wurde. Die extrem hohe Anzahl, die vom *Parser* für SEE gefunden wurde, ergibt sich daraus, dass SEE viele Prefabs aus Plugins enthält, die mit analysiert wurden.

Die Tabellen 5.8 und 5.9 zeigen, wie viele Kanten Unity2RFG jeweils in die CafeSharp und CSharp2RFG RFGs eingefügt hat. In Verbindung mit Tabelle 5.7 ist zusehen, dass mehr bekannte GameObjects

Projekt	Komponente	Parser	Alle
Boss Room	202	48	216
Chop Chop	194	136	236
SEE	145	201	301
TestProjekt	169	168	184

Tabelle 5.8: Anzahl der zu den CafeSharp RFGs von Unity2RFG hinzugefügten Kanten.

Projekt	Komponente	Parser	Alle
Chop Chop	194	136	236
SEE	150	206	307
TestProjekt	170	169	185

Tabelle 5.9: Anzahl der zu den CSharp2RFG RFGs von Unity2RFG hinzugefügten Kanten.

zu mehr hinzugefügten Kanten führen. Da Unity2RFG nur Kanten hinzufügt, wenn es nachprüfen kann, dass eine Komponente in einer Szene vorkommt, ist die leicht nachvollziehbar.

	CafeSharp			CSharp2RFG			Unity
Projekt	Komp.	Parser	Alle	Komp.	Parser	Alle	MCS
Boss Room	463	790	451	-	-	-	8
Chop Chop	103	227	46	184	284	128	276
SEE	277	224	57	487	444	221	317
TestProjekt	33	34	17	31	32	15	0

Tabelle 5.10: Anzahl gefundener toter Methoden mit verschiedenen Szenen Analysen und von Unity.

Tabelle 5.10 zeigt, wie viele tote Methoden von Axivionund Unity gefunden wurden. Es ist zusehen, dass mehr bekannte GameObjects und dadurch mehr hinzugefügte Kanten zu weniger toten Methoden führen, da fehlende Aufrufe ergänzt wurden.

Außerdem fällt auf, dass die DeadCode-Detection mit CSharp2RFG wesentlich mehr tote Methoden gefunden hat als mit CafeSharp. Dies wird wahrscheinlich durch das, wie in Abschnitt 4.6.1 bereits beschrieben, zum CSharp2RFG RFG hinzugefügte *Linkage.Is_Definition* Attribut verursacht. Unity2RFG fügt das Attribut zu allen Methoden hinzu, wodurch auch viele Knoten dieses Attribut bekommen, die es mit CafeSharp nicht haben. Die DeadCode-Detection ignoriert Knoten ohne das *Linkage.Is_Definition* Attribute, weswegen sie mehr Methoden meldet, wenn alle Knote das Attribut haben.

Die Tabelle 5.10 zeigt auch, dass Axivion und Unity2RFG unterschiedlich viele tote Methoden gefunden haben. Eine erste Vermutung, woher dieser Unterschied kommt ist, dass aufgrund von fehlend Szenen-Daten nicht für alle existierenden Aufrufe Kanten hinzugefügt wurden. Eine

	Komponente		Parser		Alle	
Projekt	Gesamt	Stripped	Gesamt	Stripped	Gesamt	Stripped
Boss Room	463	6	790	7	451	6
Chop Chop	103	19	227	50	46	5
SEE	277	8	224	8	57	8
TestProjekt	33	0	34	0	17	0

Tabelle 5.11: Anzahl der überlappenden toten Methoden von Axivion und Unity für CafeSharp.

	Komponente		Parser		Alle	
Projekt	Gesamt	Stripped	Gesamt	Stripped	Gesamt	Stripped
Chop Chop	184	18	284	52	128	3
SEE	487	11	444	11	221	11
TestProjekt	31	0	32	0	15	0

Tabelle 5.12: Anzahl der überlappenden toten Methoden von Axivion und Unity für CSharp2RFG.

genauere Betrachtung der Ergebnisse zeigt allerdings, dass es sehr wenig Überlappungen zwischen den Ergebnissen von Axivion und Unity gibt, wie die Tabellen 5.11 und 5.12 zeigen.

In diesen Tabellen wird dargestellt, wie viele tote Methoden von Axivion gefunden wurden und wie viele sich mit den von Unity gefundenen überschneiden. Viele der Methoden, die von Unity als tot gemeldet werden, sind in den RFGs als Entries definiert, weswegen Axivion sie nicht als tot betrachtet, obwohl sie nie aufgerufen werden.

In der anderen Richtung könne fehlende Szenen-Daten ein Grund sein. Aber auch die Analyse unter der Annahme, dass alle Komponenten in einer Szene vorkommen, hat sehr unterschiedliche Ergebnisse produziert. Die Kriterien, mit denen Unity entscheidet, ob eine Methode tot ist, sind leider nicht öffentlich dokumentiert, weswegen die von Unity gefundenen toten Methoden schwer nachvollziehbar sind. Manuelles nachprüfen der Methoden, die nicht von Unity gemeldet wurden, zeigt jedoch, dass ein Großteil der nur von Axivion gefundenen Methoden tatsächlich tot sind.

5.6 ZUSAMMENFASSUNG

Durch die großen Unterschiede zwischen den von Axivion und Unity gefundenen toten Methoden lässt sich keine klare Aussage über die Korrektheit von Unity2RFG treffen. Beide Analysen melden Methoden als tot, die von der jeweils anderen nicht abgedeckt werden, aber es ist nicht vollständig nachvollziehbar, woher diese Unterschiede kommen.

Des Weiteren ist die Qualität der verwendeten Szenen-Daten entscheidend, damit Unity2RFG entsprechende Kanten in den RFG einfügt.

6 FAZIT

Das für die Bachelorarbeit erstellte Tool Unity2RFG ist in der Lage einen gegebenen RFG von CafeSharp oder CSharp2RFG um Aufrufe durch die betrachteten Unity Mechanismen zu erweitern. Dabei berücksichtigt es, ob Komponenten in Szenen vorkommen, solange die dafür benötigten Informationen vorhanden sind.

Die Evaluation zeigt, wie stark die Qualität der Szenen-Daten die Ergebnisse beeinflusst. Durch die Unterschiede zwischen den von Axivionund Unity gefundenen toten Methoden ist die Korrektheit von Unity2RFG aus ihr leider nicht ersichtlich.

6.1 AUSBLICK

Im Abschnitt Alternative Ansätze wurden bereits Möglichkeiten vorgestellt, wie ein Tool wie Unity2RFG als Unity Komponente oder Teil von CSharp2RFG umgesetzt werden könnte. Hier werden jetzt noch Ideen vorgestellt, wie Unity2RFG von hier weiter entwickelt werden könnte.

6.1.1 *Verbesserte Event Analyse*

Für Events betrachtet Unity2RFG nur das Registrieren von Methoden und den Aufruf des Events mit `Invoke()`. Sowohl C# als auch Unity Events erlauben es registrierte Methoden wieder zu entfernen. Dadurch besteht die Möglichkeit, dass eine registrierte Methode vor dem Event Aufruf wieder entfernt wurde und dadurch nie aufgerufen wird. Unity2RFG führt keine Analyse durch, um einen solchen Fall zu erkennen, was zu falsch hinzugefügten Aufrufkanten im RFG führt.

6.1.2 *GameObjects und Komponenten Zustand Berücksichtigen*

GameObjects und Komponenten können deaktiviert werden, wodurch Unity die dazugehörigen Unity Messages nicht aufruft. Eine Komponente, die in einer Szene vorkommt, aber nie aktiv ist, kann deswegen als tot angesehen werden. Der Zustand von GameObjects und Komponenten wird von Unity2RFG bisher nicht berücksichtigt. Eine entsprechende Analyse sollte aber implementierbar sein, sodass auch diese Fälle abgedeckt werden.

6.1.3 *Performance Verbesserung*

Unity2RFG arbeitet in drei aufeinander aufbauenden Schritten: 1. Szene Analyse, 2. Analyse Vorbereitung, 3. RFG Erweiterung. Alle diese Schritte sollten sich relative simple parallelisieren lassen. Den ersten Schritt betrifft dies hauptsächlich die statische Szenen-Analyse, da die gelesenen Szenen-Dateien nicht voneinander abhängig sind, können sie parallel gelesen werden. Die anderen beiden Schritte verarbeiten die Syntaxbäume der kompilierten Assemblies momentan nach einander, aber es spricht nichts dagegen mehrere Syntaxbäume in parallel zu durchlaufen. Aber da die Schritte jeweils auf Informationen aus dem vorhergehenden Schritt aufbauen, sollte bei parallelisieren darauf geachtet werden, dass die Schritte jeweils komplett abgeschlossen sind, bevor der nächste gestartet wird.

A

GLOSSAR

.Net Reflector .Net Reflector ist ein von Red Gate entwickeltes Tool mit dem C# Assemblies durchsucht und dekompiliert werden können. 25

Code-City In der Code-City-Metapher werden Softwarekomponenten durch Gebäude in einer Stadt repräsentiert, wobei die Eigenschaften dieser Gebäude verschiedene Metriken der Software ausdrücken können — z. B. könnte die Höhe eines Gebäudes der Anzahl der Codezeilen entsprechen. 24

Coroutine Erlaubt es, das eine Methode über mehrere Frames ausgeführt wird. 8

DiffMerge DiffMerge its ein visuelles Datei und Ordner Diff-Tool von SourceGear. 25

IL2CPP Unitys ahead-of-time Compiler, übersetzt C# zu C++ beim bauen. 25

Mono Unitys Just-In-Time Compiler basierend auf dem Mono Projekt. 25

Netcode Eine high-level Networking und Multiplayer library für Unity. 8, 12, 23, 24

B

ABBILDUNGSVERZEICHNIS

C

TABELLENVERZEICHNIS

D

LITERATURVERZEICHNIS

[Axivion 2022] GmbH, Axivion: *Axivion Suite Documentation*. 2022

[BossRoom 2021] Unity, Unity-Community: *Boss Room*. 2021. – URL https://github.com/Unity-Technologies/com.unity.multiplayer.samples.coop. – Zugriffsdatum: 2022-12-26

[ChopChop 2021] Unity, Unity-Community: *Unity Open Project #1: Chop Chop*. 2021. – URL https://github.com/UnityTechnologies/open-project-1. – Zugriffsdatum: 2022-12-26

[CSharpSyntaxWalker 2021] Microsoft: *CSharpSyntaxWalker Class*. 2021. – URL https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.csharpsyntaxwalker?view=roslyn-dotnet-4.3.0. – Zugriffsdatum: 2022-12-29

[Managed Code Stripping 2021] Unity: *Managed code stipping*. 2021. – URL https://docs.unity3d.com/Manual/ManagedCodeStripping.html. – Zugriffsdatum: 2022-12-26

[Measure-Command 2023] Microsoft: *Measure-Command*. 2023. – URL https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command

[MonoBehaviour 2021] Unity: *MonoBehaviour*. 2021. – URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. – Zugriffsdatum: 2022-12-26

[NetworkBehaviour 2021] Unity: *MonoBehaviour*. 2021. – URL https://docs.unity3d.com/Manual/class-NetworkBehaviour.html. – Zugriffsdatum: 2022-12-26

[Roslyn 2021] Microsoft: *.NET Compiler Platform-SDK*. 2021. – URL https://learn.microsoft.com/de-de/dotnet/csharp/roslyn-sdk/. – Zugriffsdatum: 2022-12-27

[Seifert und Wislaug 2017] Seifert, Carsten ; Wislaug, Jan: *Spiele Entwickeln Mit Unity 5*. Carl Hanser Verlag München, 2017. – ISBN 978-3-446-45197-1

[UnityEvents 2021] Unity: *UnityEvent*. 2021. – URL https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html. – Zugriffsdatum: 2022-12-26