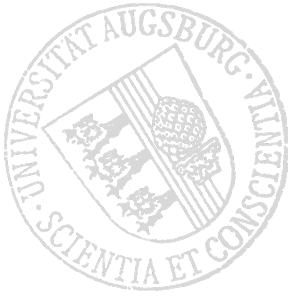
Universität Augsburg
Fakultät für Angewandte Informatik

Forschungsmodul
im Studiengang Informatik
(Bachelor of Science)

SCA2AMALTHEA

Verwendung, Konzept und Erweiterung

Dominic Beger



Software Methodologies
for Distributed Systems

Dominic Beger

SCA2AMALTHEA

Betreuer: **M. Sc. Christoph Etzel**,
Fakultät für Angewandte Informatik, Universität Augsburg

Matrikelnummer: 1530241

Abgabedatum: 24. Februar 2021

Zusammenfassung

Diese Arbeit soll einen Überblick über die verschiedenen Aspekte des clang-basierten Plugins SCA2AMALTHEA geben, das im APP4MC für die automatisierte Generierung eines AMALTHEA-Softwaremodells aus existierendem C-Quellcode verwendet werden kann. Zunächst wird hierfür Schritt für Schritt erläutert, wie die Installation/Einrichtung abläuft. Im Anschluss wird auf die C++-Implementierung sowie den zugehörigen Java-Teil im Plugin detailliert eingegangen. Da das Projekt stetig weiterentwickelt wird, ist hierbei das Ziel, dem Leser eine ausreichend verständliche Erklärung zu vermitteln, damit dieser zukünftige Änderungen ebenso nachvollziehen und Probleme lokalisieren kann. Die genaue Verwendung im APP4MC soll dann anhand eines Beispieles demonstriert werden. Für einen ausgewählten, fehlenden Teil des Plugins, der sich aus der Implementierungsanalyse ergibt, soll zusätzlich eine entsprechende Erweiterung gebaut werden. Somit soll der Leser auch in der Lage sein, eigene, notwendige Erweiterungen für SCA2AMALTHEA zu entwickeln. Da sich das AMALTHEA-Metamodell oder das APP4MC selbst ebenfalls mit jeder neuen Version verändern wird und Bosch nicht immer in Echtzeit Updates für das Plugin veröffentlicht, ist dies durchaus hilfreich, um nicht auf etwaige Migrationen warten zu müssen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung und Motivation

Das APP4MC (*Application Platform Project For MultiCore*) dient als Werkzeug für die Entwicklung und Optimierung von Software auf Mehrkernprozessoren, die inzwischen zum Großteil in der automobilen Domäne für Steuergeräte eingesetzt werden, um den dort stetig wachsenden Anforderungen und Funktionalitäten gerecht zu werden. Beispielhaft wäre hier das autonome Fahren zu nennen, da dort stattfindende Berechnungen sehr rechenintensiv sind und entsprechend leistungsstarke Systeme erfordern (vgl. [1, S. 1f]). Als Basis für alle Kalkulationen nutzt das APP4MC dabei ein Modell namens AMALTHEA, das alle Hard- und Softwareaspekete sowie zusätzliche Umgebungsbedingungen des Gesamtsystems repräsentiert und damit Metriken erzeugen kann, die der Verbesserung der Software dienen.

Diese Arbeit soll einen Überblick über die verschiedenen Aspekte des clang-basierten Plugins SCA2AMALTHEA geben, das im APP4MC für die automatisierte Generierung eines AMALTHEA-Softwaremodells aus existierendem C-Quellcode verwendet werden kann. Zunächst wird hierfür Schritt für Schritt erläutert, wie die Installation/Einrichtung abläuft. Im Anschluss wird auf die C++-Implementierung sowie den zugehörigen Java-Teil im Plugin detailliert eingegangen. Da das Projekt stetig weiterentwickelt wird, ist hierbei das Ziel, dem Leser eine ausreichend verständliche Erklärung zu vermitteln, damit dieser zukünftige Änderungen ebenso nachvollziehen und Probleme lokalisieren kann. Die genaue Verwendung im APP4MC soll dann anhand eines Beispieles demonstriert werden. Da sich das AMALTHEA-Metamodell bzw. das APP4MC selbst ebenfalls mit jeder neuen Version verändern werden und Bosch nicht immer in Echtzeit Updates für das Plugin veröffentlicht, ist dies durchaus hilfreich, um nicht auf etwaige Migrationen warten zu müssen. SCA2AMALTHEA stellt dabei in der aktuellen Form kein vollautomatisiertes Modellgenerierungs-Werkzeug dar. Es hilft lediglich bei der Transformation der grundlegenden Codebasis, die dann im Nachgang jedoch noch manuell erweitert und konfiguriert werden muss. Zwar wird dem Nutzer somit das eigenständige Anlegen jeder einzelnen Funktion und Variable im Modell erspart, eine komplette Erkennung der komplexeren Zusammenhänge in einer Anwendung ist hingegen dadurch nicht abgedeckt. Das AMALTHEA-Modell umfasst aber wesentlich mehr Informationen, die diese Beziehungen beschreiben und dann händisch eingepflegt werden müssen. Ebenso ist das Plugin noch nicht in der Lage, detaillierte Aktivitätsgraphen nach den verfügbaren Möglichkeiten der Spezifikation zu erstellen. Es wird nur ein Teil abgedeckt. Aus diesem Grund soll im Rahmen dieser Arbeit auch eine Erweiterung von SCA2AMALTHEA stattfinden, die zusätzliche Funktionen des Metamodells unterstützt.

2 Einrichtung des Plugins

Der Quellcode des Plugins kann im offiziellen git-Repository[1] des APP4MC eingesehen und heruntergeladen werden. Es handelt sich dabei um eine Erweiterung für die Entwicklungsumgebung Eclipse, welche auch als Grundlage für die AMALTHEA-Plattform dient. Diese Arbeit bezieht sich auf Version 0.9.8 des APP4MC, die am 30. April 2020 veröffentlicht wurde. Alle vorgestellten Schritte sind jedoch analog auf die Versionen 0.9.9 und 1.0.0 übertragbar.

2.1 Vorbereitung und Installation

Damit eine Installation des Plugins durchgeführt werden kann, wird eine aktuelle Updatesite benötigt. Unter Umständen ist die des Repositories bereits auf dem aktuellen Stand und kann direkt verwendet werden. Um jedoch sicherzugehen, sollten die Quelldateien des referenzierten Repositorys zunächst in eine Eclipse-Umgebung eingebunden und mit Maven neu kompiliert werden. Hierzu muss der Quellcode zunächst in Form einer Archivdatei heruntergeladen werden. Dafür steht im git der in Abbildung 2.1 markierte Button `[tgz]` zur Verfügung.

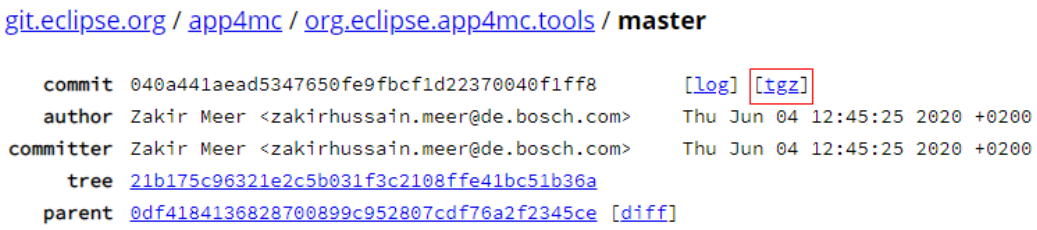


Abbildung 2.1: Download der SCA2AMALTHEA-Quelldateien

Es empfiehlt sich den aktuellsten Stand (Commit) des master-Branches auszuwählen. Es kann jedoch möglich sein, dass Bosch erst mit einiger Verspätung eine Aktualisierung von SCA2AMALTHEA für die neueste Version des APP4MC ins Repository einpflegt, sodass hier unter Umständen eine ältere Version verwendet werden muss. Die Migrations-Commits für das Plugin sind auch als solche in der Historie hinterlegt. Entsprechend schnell findet man somit mögliche Versionen, die man testen kann.

[1] https://git.eclipse.org/r/plugins/gitiles/app4mc/org.eclipse.app4mc.tools/

2.1.1 Import in Eclipse

Nachdem das Herunterladen abgeschlossen ist, muss die gewählte Archivdatei, die als `tar.gz` vorliegt, entpackt werden. Hierfür muss ein beliebiges Tool, wie beispielsweise 7zip oder WinRAR verwendet werden. Es hat sich jedoch gezeigt, dass beim Entpacken mit 7zip Probleme auftreten können. Da das Plugin sehr lange Ordner- und Dateinamen enthält, kam es beispielsweise vor, dass Namen von Java-Klassen oder Dateien abgeschnitten wurden oder Dateiendungen fehlten bzw. unvollständig waren. Dies führt natürlich dazu, dass das Projekt nicht richtig kompiliert werden kann. Beim Verwenden von WinRAR sind diese Probleme nicht aufgetreten. Der Ordner für das Plugin befindet sich im entpackten Verzeichnis unter `eclipse-tools/sca2amalthea`. Für die Kompilierung zu JAR-Dateien muss dieser in Eclipse importiert werden. Dies erfolgt in Eclipse links in der (noch leeren) Projektübersicht über den Link `Import Project`. Es öffnet sich ein Fenster, in dem man nun als Import Wizard `Existing Maven Projects` auswählen muss (siehe Abbildung 2.2).

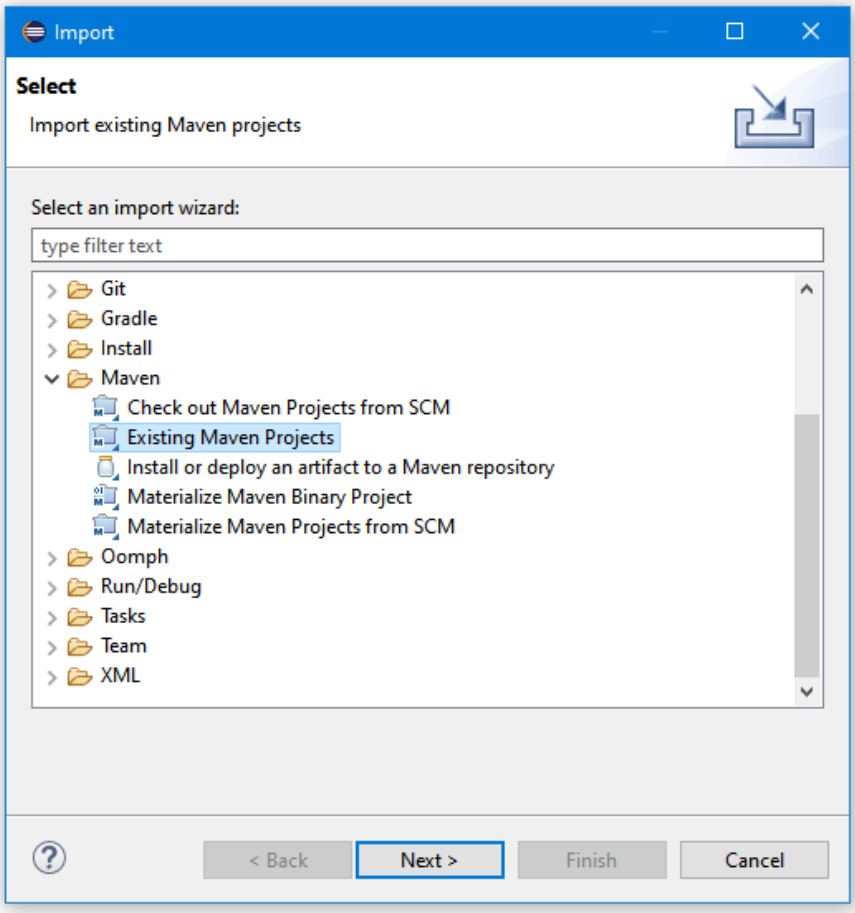


Abbildung 2.2: Import eines Maven-Projekts

Im nächsten Schritt kann der Plugin-Ordner ausgewählt werden (vgl. Abbildung 2.3).

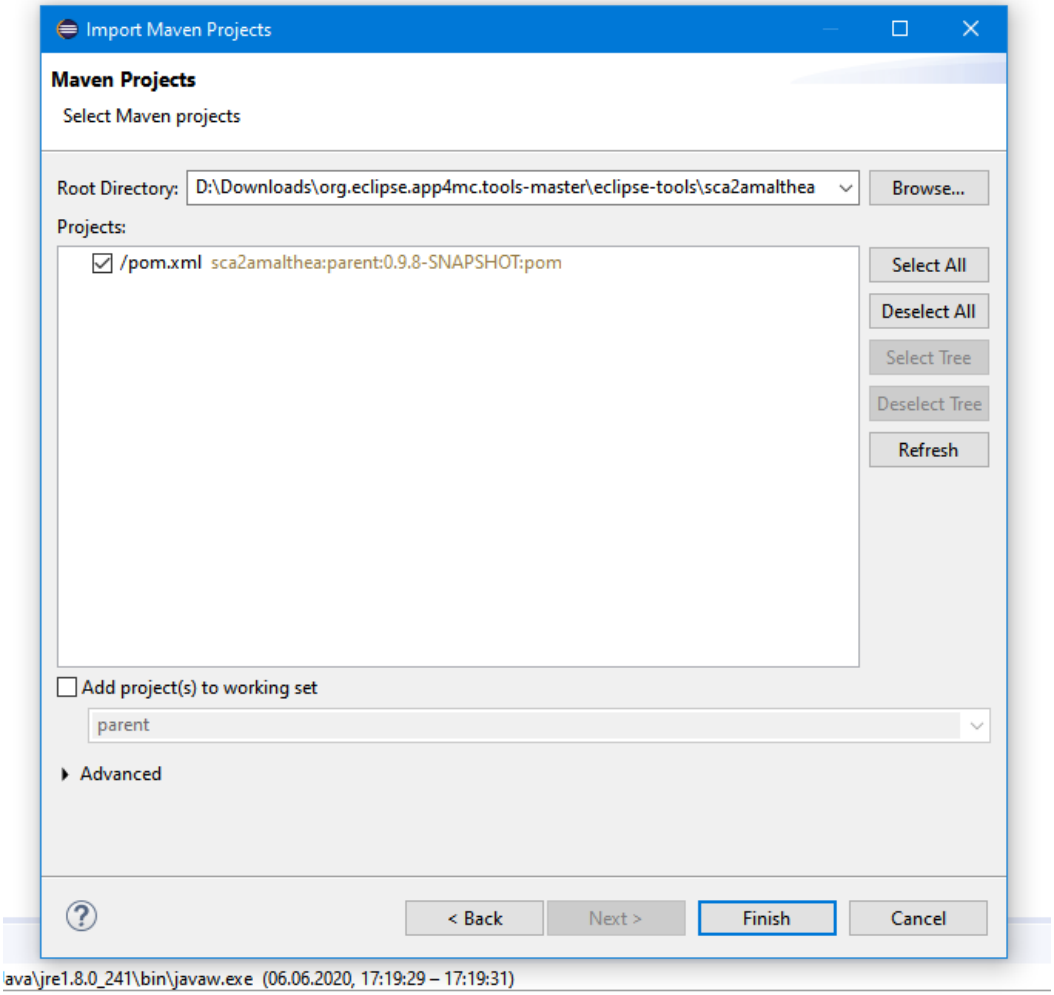


Abbildung 2.3: Auswahl des Plugin-Ordners für den Import

Es erscheinen die Namen der im Wurzelverzeichnis gefundenen Maven-Pakete mit der entsprechenden Version.

In diesem Fall findet Eclipse dabei die Haupt-`pom.xml`-Datei (`sca2amalthea 0.9.8-SNAPSHOT`), die auch als solche in der Liste `Projects` erkannt und angewählt werden sollte. Weitere `pom.xml`-Dateien für die einzelnen Projekte innerhalb des kompletten Pakets von SCA2AMALTHEA, die in den jeweiligen Unterordnern liegen, werden automatisch durch diese `pom.xml` referenziert. Der Import kann nun über den Knopf `Finish` abgeschlossen werden, sodass der Quellcode des Plugins nun in der Projektübersicht im Ordner `plugins` zur Verfügung steht. Die restlichen Verzeichnisse sind zunächst nicht von Relevanz.

2.1.2 Kompilieren der Updatesite

Um nun fertige Binaries zu erhalten, muss Maven ausgeführt werden. Dazu wird eine adäquate Run Configuration benötigt, die erst angelegt werden muss.

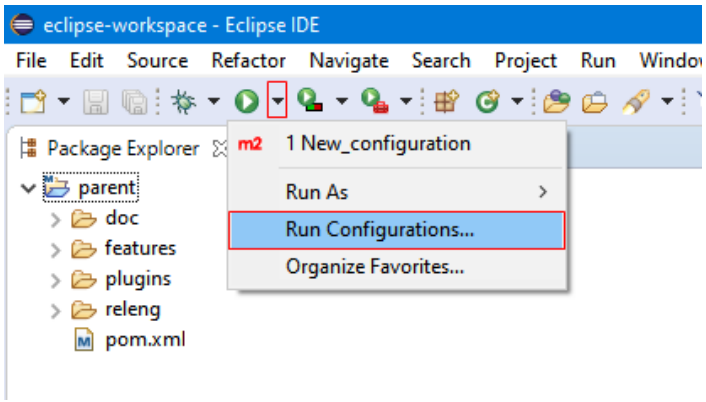


Abbildung 2.4: Einrichtung der Run Configuration

Mittels eines Klicks auf den in Abbildung 2.4 gezeigten Menüeintrag öffnet sich ein Dialog, in dem nun auf der linken Seite `Maven Build` ausgewählt werden kann (vgl. Abbildung 2.5).

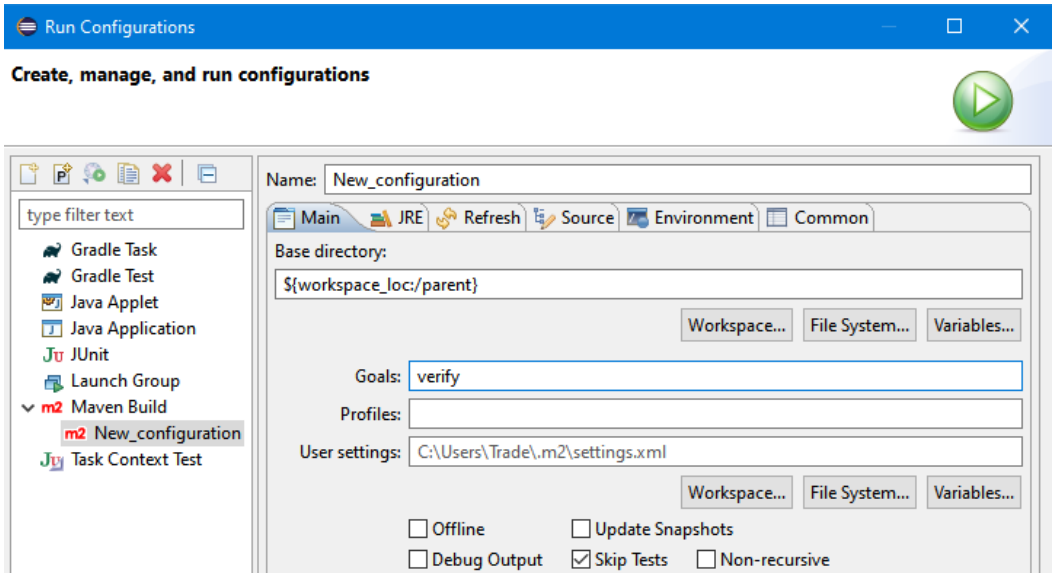


Abbildung 2.5: Maven Build Run Configuration

Als Basisverzeichnis dient der Ordner `sca2amalthea`, in dem auch die pom.xml liegt. Da dieser bereits in die Eclipse-Umgebung importiert wurde, kann er dynamisch mit `${workspace_loc:/parent}` angegeben werden. Das Ziel, das Maven ausführen soll, ist `verify`. Nun kann diese Konfiguration durch den Knopf `Run` angestoßen werden.

2.1.3 Installation im APP4MC

Die Installation von SCA2AMALTHEA erfolgt anschließend im APP4MC über den Menüreiter `Help > Install New Software`. Es öffnet sich ein Dialog, in dem dann Quellen für die Software/Plugins, die installiert werden soll(en), ausgewählt werden können. Wie in Abbildung 2.6 ersichtlich, klickt man nun auf `Add` und dann `Local`, um den Zielordner mit den kompilierten JAR-Dateien auszuwählen. Dieser befindet sich im Plugin-Ordner (`eclipse-tools/sca2amalthea`) unter `releng/org.eclipse.app4mc.sca2amalthea.updatesite/target/repository/`.
Wichtig ist, **nicht** die Archivdatei im `target`-Ordner über den `Archive`-Knopf zu wählen. Diese hat zwar augenscheinlich denselben Inhalt und die Funktionalität für das Installieren von gepackten Ordnern wird zwar angeboten, schlägt dann jedoch fehl, weil bestimmte Abhängigkeiten nicht aufgelöst werden können.

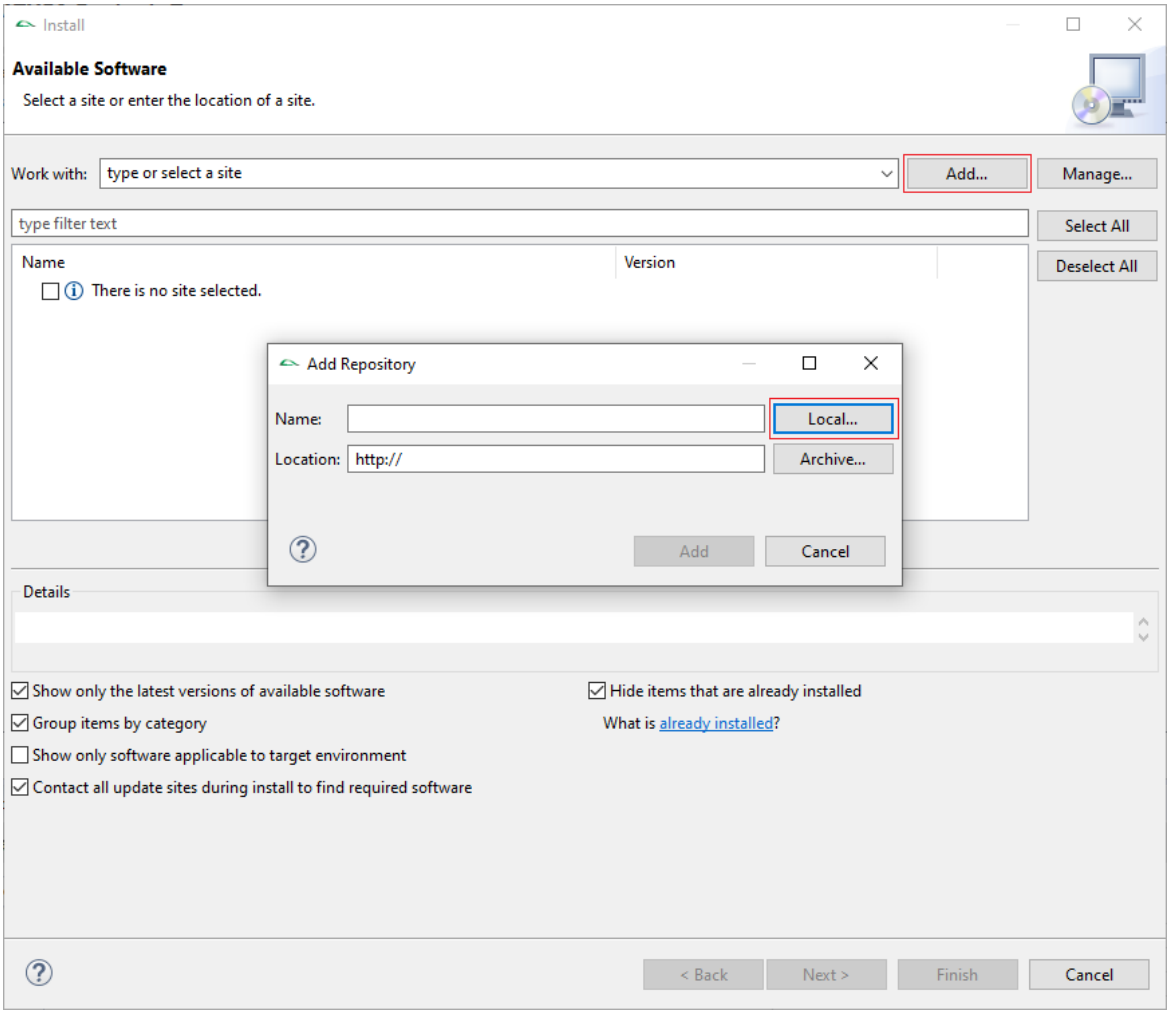


Abbildung 2.6: Installation eines Plugins aus lokalen Dateien

Abschließend müssen noch die Inhalte des Plugins installiert werden. Dazu muss in der Liste, in der der Eintrag `SCA2AMALTHEA` erscheint, die Runtime ausgewählt werden. Das SDK ist für die Verwendung nicht relevant. Vor der Installation müssen noch die Lizenzen akzeptiert werden. Sobald diese Schritte abgeschlossen sind, kann der Installationsfortschritt in der Statusleiste unten rechts eingesehen werden. Falls eine Warnung erscheint, dass das zu installierende Plugin aus unsicherer Quelle stammt, muss `Install anyway` angeklickt werden. Die Anwendung wird automatisch einen Neustart vorschlagen, wenn die Installation abgeschlossen ist.

2.2 Konfiguration

Um Code am Ende in ein Modell übersetzen zu können, ist eine statische Codeanalyse notwendig. Diese wird nicht im APP4MC selbst durch das Plugin durchgeführt, sondern basiert auf einem im SCA2AMALTHEA-Repository beigefügten C++-Projekt, das sich selbst wiederum der LLVM-Compilerinfrastruktur mit clang als Frontend bedient. Die konkrete Architektur und Implementierung wird in Kapitel 3 ausführlich beschrieben. Dieses Projekt muss zu einer entsprechenden, ausführbaren Datei kompiliert werden (`sca.exe`), die dann in den Plugineinstellungen innerhalb des APP4MC referenziert und beim Übersetzungsvorgang parametrisiert ausgeführt wird.

2.2.1 Kompilieren der SCA-Executable

Um das SCA-Projekt kompilieren zu können, müssen zunächst die Quelldateien für **llvm** und **clang** vom offiziellen Remote-Repository geladen werden, da diese als Basis für die Verarbeitung des C-Quellcodes und Generierung der IR (für das AMALTHEA-Metamodell) dienen. Im SCA2AMALTHEA-Ordner befindet sich unter `plugins/org.eclipse.app4mc.sca2amalthea.docu/help_documents` eine Präsentation, die die folgenden Schritte erläutert. Diese ist jedoch bereits älter und referenziert daher auch weit zurückliegende Versionen der verwendeten Software. Sie dient als Anhaltspunkt für diese Arbeit. Ein paar Schritte erfordern jedoch diverse Anpassungen.

Vorbereitung

Um die Quelldateien kompilieren zu können, ist ein entsprechender Compiler bzw. eine IDE notwendig. Als Empfehlung wäre Microsoft Visual Studio[2] zu nennen, da es viele verschiedene Compiler unterstützt. Diese Arbeit stützt sich auf den MSVC (Microsoft Visual C++-Compiler), der standardmäßig verwendet wird. Alternativ kann beispielsweise auch JetBrains' CLion verwendet werden, das bereits auf CMake basiert. Visual Studio unterstützt inzwischen auch eine direkte CMake-Integration, allerdings wird diese Arbeit den klassischen Weg beschreiben, den auch Bosch so offiziell in der Dokumentation angibt.

[2] https://visualstudio.microsoft.com/de/vs/features/cplusplus/

Dieser macht zusätzlich noch eine Verwendung der CMake-GUI[3] für die Generierung der Builddateien notwendig. Installiert werden sollte laut offizieller Anleitung unter anderem auch ein SCM-Checkout-Tool. Hierfür könnte entweder git über die Kommandozeile ausgeführt oder ein Werkzeug wie TortoiseGit, das auch eine Benutzeroberfläche mit Integration für den Windows-Explorer anbietet, benutzt werden. Alternativ steht jedoch der Quellcode jeder Version bereits als Archiv im Repository zur Verfügung.
Das sca-Projekt wurde für LLVM 4.x kompiliert. Zum Zeitpunkt dieser Arbeit existiert bereits Version 10.0.1, die durch Änderungen, die inbesondere an der clang-API vorgenommen wurden, nicht mehr direkt kompatibel ist. Daher bleibt entweder eine Anpassung des Quellcodes auf die neueste Version oder das Verwenden eines älteren Releases. Letztere Variante funktioniert in jedem Fall ohne größere Anpassungen und ist rein theoretisch für den Verwendungszweck ausreichend. Da sich die Fehler, die aufgrund der geänderten API entstanden sind, jedoch auf die Nutzung einer in Version 7.x ersetzten (bzw. umbenannten) Methode[4] und ein Problem mit der Kommandozeilen-API beschränken, ist die erste Option (vor allem aus Gründen der Aktualität) vorzuziehen und soll daher auch in dieser Arbeit umgesetzt werden. Unabhängig von clang sind ohnehin noch kleinere Anpassungen nötig, da der Namespace für das rekursive Iterieren in Ordnerstrukturen nicht gefunden wird und somit einen Compiler-Fehler erzeugt. Die Schritte für die Behebung werden im Abschnitt Behebung von Fehlern erläutert.

Der Inhalt des Repositories kann zunächst entweder in der git bash mit `git clone -b master --single-branch https://github.com/llvm/llvm-project.git` geklont werden (siehe Abbildung 2.7) oder direkt als Release[5] in komprimierter Form heruntergeladen werden (empfohlen).

Generierung der Build-Dateien

Im Folgenden bezeichnet `llvm-path` das lokale Verzeichnis, in das der Inhalt des Repositories geklont wurde. Sobald alle notwendigen Werkzeuge installiert und einsatzbereit sind, ist der erste Schritt, den Ordner `llvm-path/clang` in den Ordner `llvm-path/llvm/tools` zu kopieren. Danach kommt erneut der Plugin-Ordner (`eclipse-tools/sca2amalthea`) ins Spiel.
Unter `/plugins/org.eclipse.app4mc.sca2amalthea.c_sources` befindet sich ein Ordner `sca`, der nach `llvm-path/llvm/tools/clang/tools` kopiert werden muss. In diesem befindet sich das C++-Projekt von Bosch, das die statische Codeanalyse verwaltet. Damit das SCA-Projekt nun auch eingebunden wird und verwendet werden kann, muss die entsprechende `CMakeLists.txt`-Datei unter `llvm-path/llvm/tools/clang/tools` angepasst werden, indem man ganz oben folgende Zeile hinzufügt: `add_subdirectory(sca)`
Dies stellt sicher, dass das SCA-Projekt auch als Unterordner berücksichtigt wird.

[3] https://cmake.org/runningcmake/
[4] https://reviews.llvm.org/D50346
[5] https://github.com/llvm/llvm-project/releases/tag/llvmorg-10.0.1

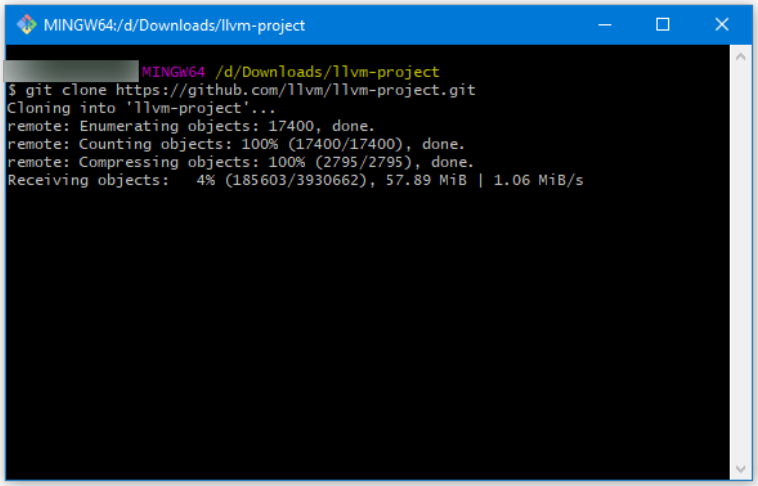


Abbildung 2.7: Klonen des LLVM-Repositories

Projektdateien für Visual Studio Falls Microsoft Visual Studio verwendet wird, sind Build-Dateien notwendig, damit die Entwicklungsumgebung eine passende Projektstruktur vorfindet, um alle Projekte importieren und erstellen zu können. Sollte eine andere IDE benutzt werden, die direkt mit CMake arbeiten kann, ist dieser Schritt überspringbar. Es muss nun die CMake-GUI ausgeführt werden. Für den Quellcode-Pfad ist `llvm-path/llvm` einzusetzen. Der Pfad für die Zieldateien kann beispielsweise einfach ein Ordner `llvm-path/llvm/build` sein. Als Generator wird der für die entsprechende Version von Microsoft Visual Studio verwendet. In dieser Arbeit ist dies `Visual Studio 16 2019` (siehe Abbildung 2.8).

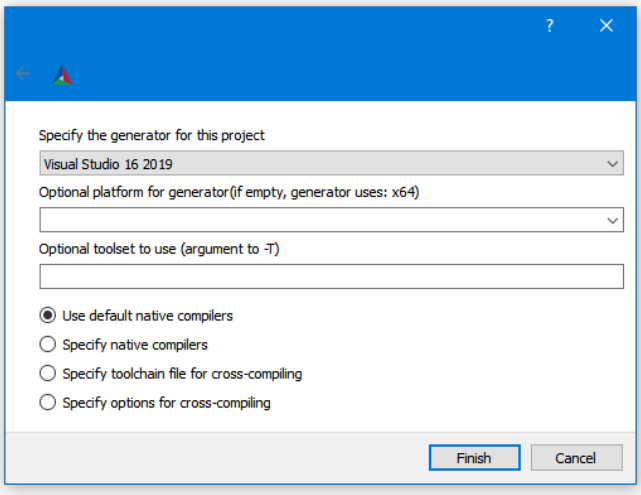


Abbildung 2.8: Generator für Visual Studio 2019

Durch einen Klick auf `Configure` wird alles vorbereitet. Sollte in der Ausgabe ein Fehler `The keyword signature for target_link_libraries has already been used with the target sca` erscheinen, muss die `CMakeLists.txt` im `sca`-Ordner angepasst werden und ganz unten die Zeile `target_link_libraries(sca` durch `target_link_libraries(sca PRIVATE` ersetzt werden. Dann kann mit einem Klick auf `Generate` die Generierung gestartet werden (siehe Abbildung 2.9).
Sobald diese abgeschlossen ist, kann die komplette Solution (`LLVM.sln`) in Visual Studio geöffnet werden.

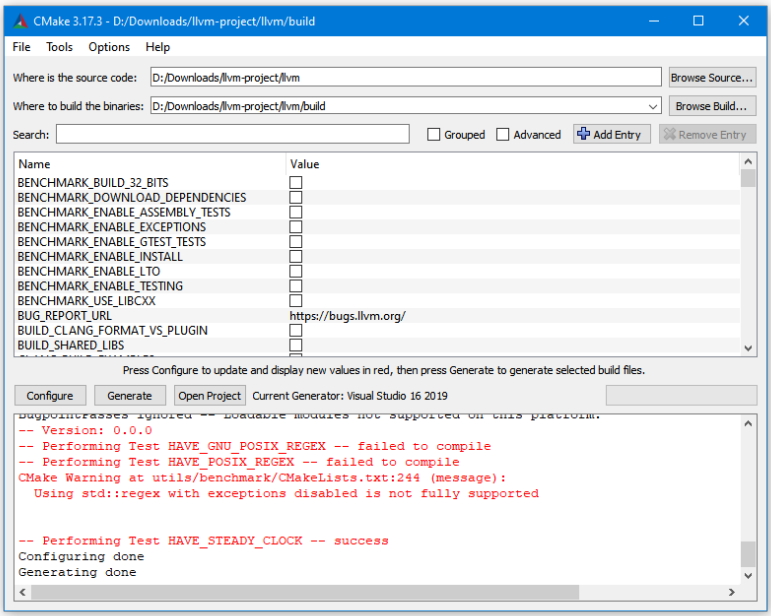


Abbildung 2.9: CMake-GUI für die Generierung der Projektdateien

Behebung von Fehlern

Da der Stand des sca-Projektes nicht mehr aktuell ist und auch unklar bleibt, ob und wann Bosch ein Migrationsupdate für dieses veröffentlicht, müssen folgende Fehler nun manuell behoben werden. Alternativ könnte man auch auf veraltete Stände der Standardbibliothek zurückgreifen, um den Code auszuführen, was jedoch wenig zielführend ist. Der verwendete Namespace `std::tr2::sys` wurde bereits in C++14 als veraltet markiert und schließlich in C++17 entfernt. Aus diesem Grund wird dieser beim Öffnen des Projektes in Visual Studio bzw. beim späteren Kompilieren nicht gefunden. Er wurde in C++17 durch einen neueren Namespace `std::filesystem` ersetzt. Der gewählte Standard im sca-Projekt ist jedoch C++14, sodass dieser in dieser Form noch nicht verwendet werden kann. Stattdessen befindet sich der äquivalente Inhalt in einem Experimental-Namespace (`std::experimental::filesystem`). Die entsprechende Einbindung für `std::tr2::sys` in Zeile 23 der Datei `Helpers.cpp` muss dann durch

`using namespace std::experimental::filesystem;` ersetzt werden. Die Inkludierung von `filesystem` In Zeile 16 muss außerdem zu `experimental/filesystem` geändert werden. Innerhalb der Datei `TraverseASTMainHelpers.cpp` befindet sich in Zeile 25 ebenso noch ein redundanter Import des Namespaces, der gelöscht werden kann. Standardmäßig ist in diesem Header eine Fehlermeldung hinterlegt, die besagt, dass der Namespace veraltet ist und in zukünftigen C++-Versionen entfernt wird. Dieser Fehler tritt dann auch beim Kompilieren auf. Um ihn zu deaktivieren und die Klasse dennoch nutzen zu können, muss eine entsprechende Präprozessordirektive definiert werden. Hierfür muss im Projektmappenexplorer unter `clang executables` ein Rechtsklick auf das Projekt `sca` erfolgen und dann `Eigenschaften` ausgewählt werden. Es öffnet sich das in Abbildung 2.10 gezeigte Fenster. Unter `Konfigurationseigenschaften > C/C++ > Praeprozessor` kann dann in der Liste `Praeprozessordefinitionen` der Eintrag `<Bearbeiten...>` gewählt werden.

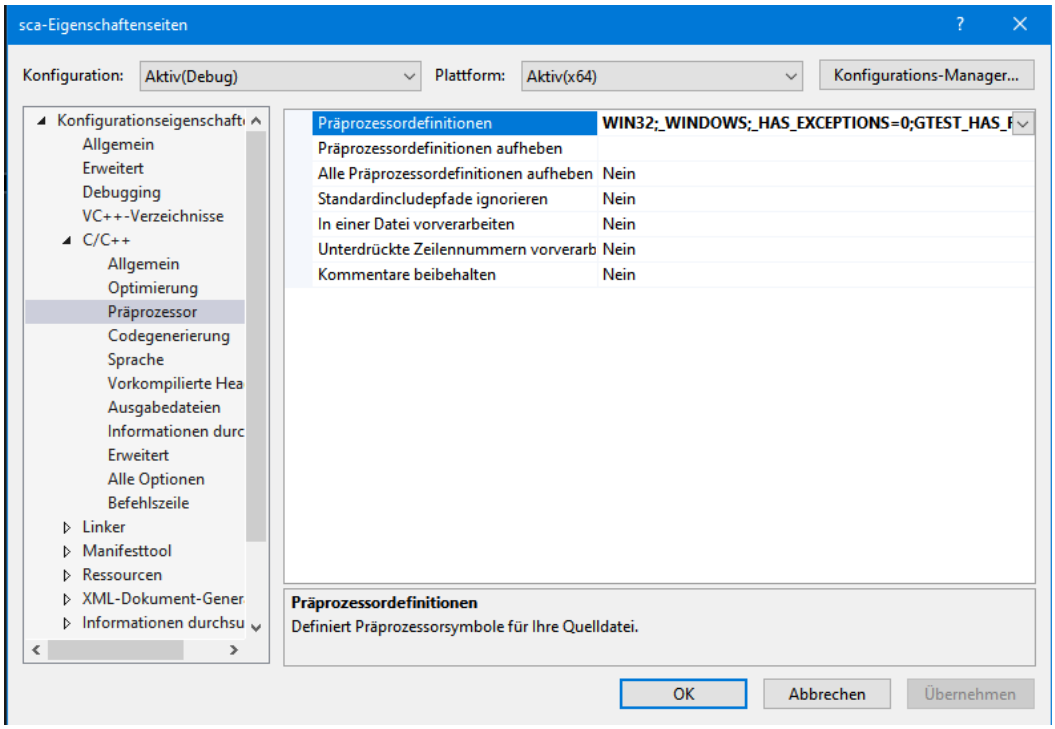


Abbildung 2.10: sca-Eigenschaftsfenster

Im folgenden Dialog (vgl. Abbildung 2.11) muss dann im obersten Textfeld ganz unten die Zeile `_SILENCE_EXPERIMENTAL_FILESYSTEM_DEPRECATION_WARNING` angefügt werden. Sobald diese Direktive vom Präprozessor gefunden wird, ignoriert er die definierte Fehlermeldung und der Namespace kann ohne Probleme verwendet werden. Abschließend muss an das Ende der Zeilen 80 und 144 in `Helpers.cpp` noch jeweils ein `string()`-Aufruf angehängt werden, sodass sich deren Inhalt zu `std::string fileName = i->path().filename().string();` ändert. Da das sca-Projekt weiterhin auf C++14 basieren wird und ein eventuelles Update von Bosch sowieso die notwendigen Änderungen direkt mit umsetzen würde, kann dies so gelöst werden.

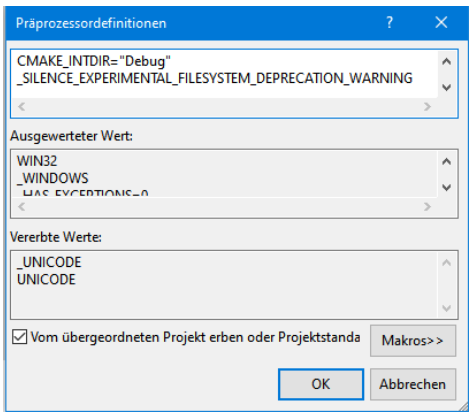


Abbildung 2.11: sca-Präprozessordefinitionen

In der Klasse `TraverseASTClassAction.cpp` muss außerdem noch jedes Vorkommen von `getLocStart` durch `getBeginLoc` ersetzt werden. Hierfür kann die Schnellersetzung (Strg+H) verwendet werden.

Die wichtigste Änderung ist, in der Klasse `TraverseASTMainHelpers.cpp` vor Zeile 414 noch den Code `cl::ResetCommandLineParser();` zu setzen, sodass sich folgendes ergibt:

```
407 void createAndStartJobs() {
408
409   char **args = getStringArrayFromStringList(headerList);
410
411   //all the header directories + program name + directory
       to parse + -- symbol for the comming includes
412   int val = headerList->size();
413
414   cl::ResetCommandLineParser();
415   CommonOptionsParser OptionsParser(val, (const char **)
       args, MyToolCategory);
416
417   vector<thread> jobs;
418
419   for (unsigned i = 0; i < allTraversingData.size(); i++) {
420
421     TraverseASTWorker t(&OptionsParser, filesForThreads.at(
       i), directoryToParse,
422                         i);
423
424     jobs.push_back(thread(&TraverseASTWorker::executeWorker
       , t));
425   }
426
```

```
427   for (unsigned i = 0; i < jobs.size(); i++) {
428
429     jobs.at(i).join();
430   }
431 }
```

Der Grund hierfür ist, dass sich die Implementierung des im sca-Projekt verwendeten `CommonOptionParser`s in LLVM 10.0.1 stark von der in Version 4.x unterscheidet. In beiden Versionen wird intern die `ParseCommandLineOptions`-Methode des `cl` -Namespaces (definiert in `llvm/lib/support/CommandLine.cpp`) aufgerufen, die intern die übergebenen Kommandozeilenparameter analysiert. Da diese Methode in der `main`-Methode bereits für die sca-Optionen (pdir, cdir, etc.) genutzt wird, baut diese eine Vielzahl unterschiedlicher Caches für die verfügbaren Optionen auf, in der Informationen zu diesen gehalten werden. Im `CommonOptionParser`-Objekt soll nun die Parameterliste hinterlegt werden, die unter anderem das Quellverzeichnis sowie die notwendigen Include-Befehle enthält und auf Basis dieser dann die Kompilierungsdaten vorbereitet. Zwar wird beim Initialisieren diese Objektes `cl::ResetAllOptionOccurrences ()` aufgerufen, es scheint jedoch noch ein zusätzliches, nicht genauer identifiziertes Cache-Objekt zu geben, das nicht zurückgesetzt wird und daher Probleme bereitet. Indem nämlich nun die eben genannte Methode `ParseCommandLineOptions` nochmals aufgerufen wird, greift diese intern auch erneut auf die registrierten sca-Optionen zurück. Hierbei findet sie unter anderem die `pdir`-Option, welche obligatorisch ist, in der neuen Parameterliste jedoch nicht gefunden wird (da diese mit den sca-Parametern gar nicht zusammenhängt und somit andere Daten beinhaltet). In Version 4.x war hierfür keine entsprechende Fehlerbehandlung integriert und das Problem wurde dahingehend einfach von der `CommonOptionParser`-Klasse ignoriert, sodass die Ausführung weiterlief. In LLVM 10.0.1 hingegen wird die Rückgabe des Aufrufs auf fehlerfreie Ausführung überprüft und im Falle eines Problems auch eine Meldung in den Error-Stream geschrieben. Im gleichen Zug erfolgt dann durch die neue Implementierung eine Beendigung der Programmausführung in der `ParseCommandLineOptions`-Methode. Folglich stürzt das sca-Projekt ab und in der Datei `stdErr.log` findet sich die Information `for the --pdir option: must be specified at least once! LLVM ERROR: CommonOptionsParser: failed to parse command-line arguments`.
Durch das erneute Leeren dieses Caches wird diese Problematik umgangen. Ob dies tatsächlich einen Bug oder gewünschte Funktionalität in LLVM darstellt, ist unklar. Im Gegenzug stellt sich dadurch ebenso die Frage, ob an dieser Stelle dann theoretisch ein Implementierungsfehler im sca-Projekt vorliegt, der durch die unterdrückte Meldung und fehlende Beendigung der Ausführung unbemerkt geblieben ist und sich auch nicht weiter ausgewirkt hat. (vgl. [2], [3], [4]) Im Rahmen dieser Arbeit soll jedoch auf diese Frage keine Antwort gesucht werden, da LLVM 10.x eigentlich nicht Bestandteil des Plugins ist und sich mit der genutzten Version 4.x keine Probleme ergaben.

Kompilierung

Um ausführbare Dateien zu erhalten, muss das Projekt noch mit dem aktuellen Stand kompiliert/gebaut werden. Hierzu muss in Visual Studio der Reiter `Erstellen > Projektmappe erstellen` ausgewählt werden (siehe Abbildung 2.12). Es reicht beim ersten Mal **nicht** aus, nur das sca-projekt zu bauen. Lediglich für den Fall, dass nur das sca-Projekt nicht kompiliert werden konnte und alle anderen Abhängigkeiten bereits erstellt wurden, ist es ausreichend und sinnvoll, speziell für dieses das Erstellen zu starten. Der Fortschritt kann in der Ausgabe unten eingesehen werden. Der Erstellungsprozess der gesamten Projektmappe kann aufgrund der Größe bis zu zwei Stunden dauern und viel Rechenleistung beanspruchen. Sobald er abgeschlossen ist und für alle Projekte als erfolgreich angegeben wird, liegt die `sca.exe` in `build > Debug > bin` bzw. `build > Release > bin` (je nach Build Configuration), wobei `build` der Pfad ist, der in CMake für die Build-Dateien angegeben wurde.

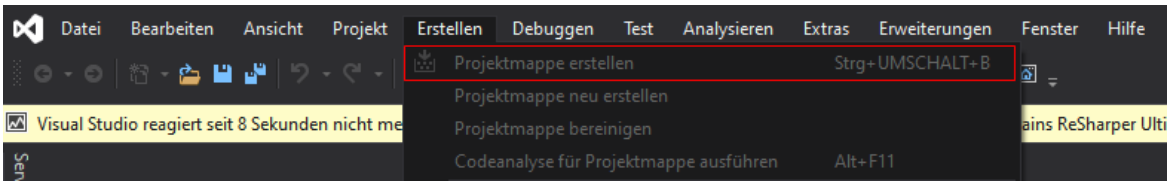


Abbildung 2.12: Erstellen der kompletten LLVM-Solution

Falls noch andere Fehler vor dem Kompilieren erscheinen, sollte trotzdem versucht werden, einen Erstellprozess zu starten. Oft kann es sein, dass die Entwicklungsumgebung Abhängigkeiten nicht direkt erkennt, bis diese kompiliert wurden, oder fälschlicherweise andere Stellen im Code bemängelt. Im Kontext dieser Arbeit wurde zusätzlich das Plugin Resharper[6] für Visual Studio verwendet, welches beispielsweise eine Methodensignatur in einer der sca-Klassen nicht richtig auflösen konnte und einen Fehler anzeigte. Es hat sich jedoch herausgestellt, dass der Compiler dies korrekt auflösen konnte und es sich nur um ein Problem des Parsers gehandelt hat. Für den Fall, dass sich Meldungen (keine Warnungen) ergeben, die auch nach dem Erstellen auftreten und hier nicht behandelt werden, können oft Lösungen aus dem Netz getestet werden, um diese manuell zu beheben. Wenn dies nicht möglich ist, könnte das Erstellen eines Beitrags im offiziellen Forum[7] hilfreich sein. Wenn jedoch alle diese Schritte mit genau gleichen Versionen und Konfigurationen befolgt werden, sollten sich keine Schwierigkeiten ergeben.

[6] https://www.jetbrains.com/resharper/
[7] http://www.amalthea-project.org/index.php/forum/fsupport

2.2.2 Einrichtung im APP4MC

Der letzte Schritt ist es nun, den Pfad zur `sca.exe` in den Plugin-Einstellungen zu hinterlegen, damit diese aufgerufen werden kann. Dafür steht im Plugin inzwischen ein Wizard bereit, der beim ersten Ausführen aufgerufen wird. Früher gab es für die Konfiguration eine entsprechende Seite in den Einstellungen, die mit der Migration auf Version 0.9.7 dann ersetzt wurde. Um diesen Wizard ausführen zu können, muss das erste Mal eine Modell-Generierung angestoßen werden. Hierfür wird zunächst ein Modellierungs-Projekt als Grundlage benötigt. Dafür muss im APP4MC links im Model Explorer mit einem Rechtsklick `New > Modeling Project` ausgewählt werden (vgl. Abbildung. 2.13). Dann muss im in Abbildung 2.14 gezeigten Dialog ein Name eingegeben werden. Nach einem Klick auf `Finish` ist dieser Schritt abgeschlossen.

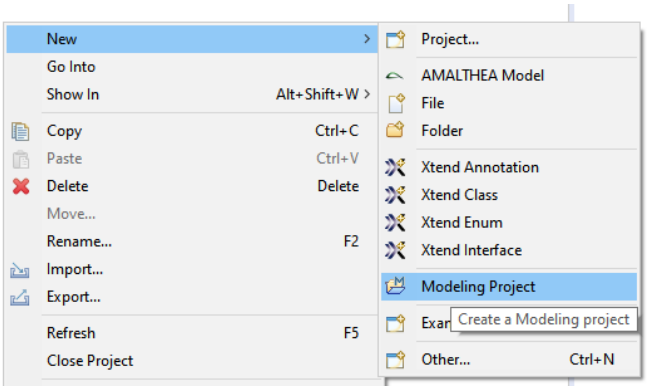


Abbildung 2.13: Hinzufügen eines Modellierungsprojektes

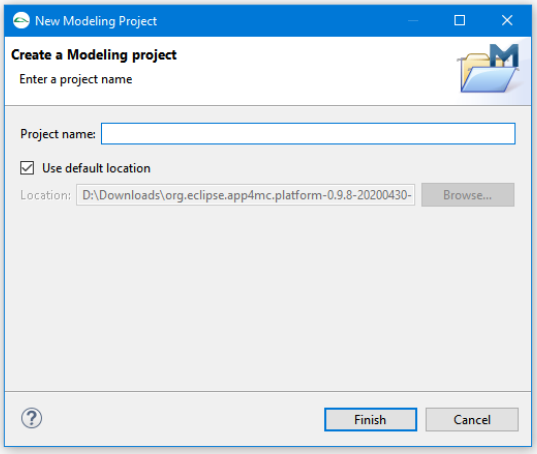


Abbildung 2.14: Eingabe eines Namens für das Modellierungsprojekt

Nun kann auf das Projekt mit einem Rechtsklick unter `SCA Tools > Generate AMALTHEA (from Source code)` der Wizard geöffnet werden (vgl. Abbildung 2.15).

Abbildung 2.15: Generierung von AMALTHEA aus Quellcode

Innerhalb des Wizards, der aus einer Seite besteht, kann unter `LLVM Executable directory`, der fälschlicherweise als `optional` gekennzeichnet ist, nun der Pfad zur vorher erstellten `sca.exe` angegeben werden. Die Einrichtung ist damit an sich abgeschlossen. Für konkrete Testzwecke werden in dieser Arbeit noch die anderen Konfigurationsfelder verwendet und erklärt werden. Speziell für die Einbindung der Standardbibliothek(en) müssen die Header-Verzeichnisse noch angegeben werden.

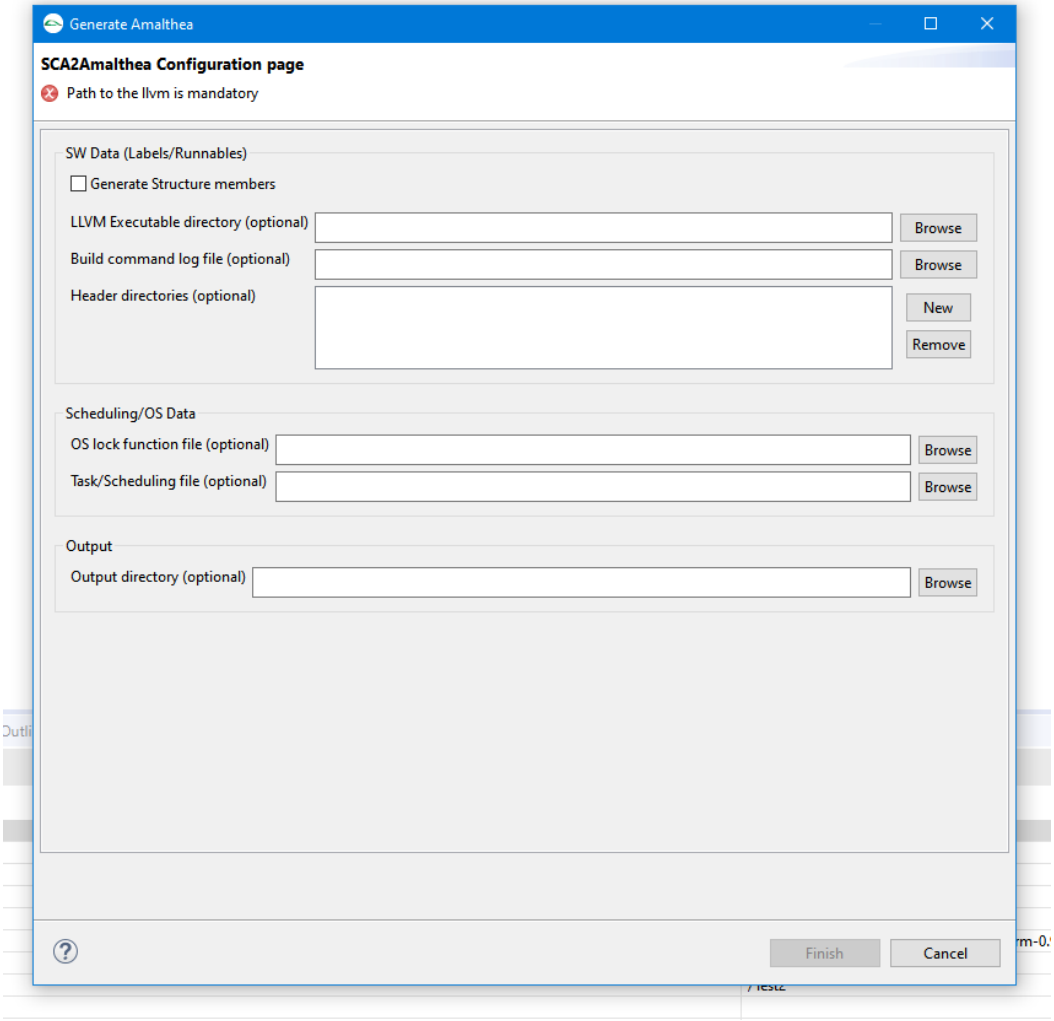


Abbildung 2.16: AMALTHEA-Einrichtungswizard

3 Konzept und Funktionsweise

Um zu verstehen, wie die Generierung des AMALTHEA-Softwaremodells funktioniert, wird im Folgenden das Zusammenspiel aller beteiligten Komponenten sowie die Aufgabe und Funktionsweise der verwendeten Compilerinfrastruktur erläutert. SCA2AMALTHEA verwendet für die Generierung der Modelldaten das OpenSource-Projekt LLVM mit clang und setzt basierend auf der dort eingebauten, statischen Quellcodeanalyse eine entsprechende Implementierung um. Die Daten stehen dann in Form eines proprietären XML-Formates zur Verfügung, das im APP4MC selbst importiert und für die Modell-Generierung genutzt wird.

3.1 LLVM-Compilerinfrastruktur und Clang

Das LLVM-Projekt (*Low Level Virtual Machine*) stellt eine modulare und erweiterbare Oberbau-Unterbau-Architektur für das Kompilieren und Ausführen von Quellcode bereit. Ein Unterbau (Frontend) bezeichnet in diesem Fall den Teil der Infrastruktur, der für die Eingabeverarbeitung zuständig ist und zu einer Zwischendarstellung des Quellcodes führt. Diese Darstellung nennt sich LLVM Intermediate Representation (kurz LLVM IM). Dabei handelt es sich um Bytecode, der einerseits für die Ausführung in der integrierten, virtuellen Maschine dient. Diese ist in der Lage, die IR zur Laufzeit für die jeweilige Prozessorarchitektur zu übersetzen und Optimierungen vorzunehmen. Dieser Ansatz wird beispielsweise auch bei der Programmiersprache Java mit der Java Virtual Machine verfolgt. Die Verwendung der virtuellen Maschine ist jedoch nicht obligatorisch, denn es besteht auch die Möglichkeit die Intermediate Representation direkt in optimierten Maschinencode zu übersetzen. Hierfür ist das LLVM-Backend zuständig. So können Compiler-Tools im Frontend mit moderatem Aufwand Maschinensprache erzeugen, indem sie den Quellcode in die LLVM IR übersetzen und diese an das Backend übergeben. Es kann mit der Infrastruktur eine Vielzahl an Programmiersprachen kompiliert werden. Zu den wichtigsten zählen unter anderem C, C++, Java, Objective-C, Swift, Ruby und Python. Da bei der Entwicklung von Steuergerätsoftware C/ C++ zum Einsatz kommt und SCA2AMALTHEA auch nur die Generierung eines Softwaremodells aus C-Quellcode unterstützt, beschränkt sich das Frontend in diesem Fall auch auf diese Sprache. `clang` ist ein nativer LLVM-Compiler für C, C++ und Objective-C und wird entsprechend auch im Plugin verwendet. Dieser übernimmt jedoch nicht nur die Generierung der Intermediate Representation, sondern bietet unter anderem auch eine umfangreiche API für die Codeanalyse und -traversierung, von der das sca-Projekt Gebrauch macht. (vgl. [5], [6])

3.2 Implementierung im SCA-Projekt

Es sei erwähnt, dass das sca-Projekt zum Stand dieser Arbeit nicht vollständig den gängigen, objektorientierten Prinzipien folgt. So wurde während Analyse des vorhandenen Quellcodes festgestellt, dass unter anderem das Single Responsibility-Prinzip nicht sauber eingehalten wurde, da einige Methoden mehrere, unterschiedliche Aufgaben übernehmen. Diese Anmerkung ist ebenso auf die Klassenebene ausdehnbar. Zusätzlich ist die Nomenklatur teilweise nicht optimal gewählt und manche Methoden haben abweichende Implementierungen als man zunächst durch deren Namen implizieren würde. Des Weiteren finden sich einige Passagen, in denen Code mehrfach vorkommt (Verstoß gegen das DRY-Prinzip) und auch kompakter formuliert werden könnte. Allokierter Speicherplatz wird außerdem nicht mittels `delete` freigegeben, was jedoch aufgrund der kurzen Ausführungsdauer der Anwendung keine merklichen Probleme bereiten dürfte. Auffällig ist auch, dass Funktionalitäten in gröbere Methoden verpackt und ausgelagert wurden, aber dennoch zum Teil ungenutzt bleiben. Insgesamt ist somit festzuhalten, dass der Quellcode nicht optimiert und ideal lesbar ist. Hinsichtlich der Größe des Projektes ist eine verständliche Analyse dennoch mit moderatem Aufwand möglich. Diese Arbeit soll dahingehend die Implementierung chronologisch traversieren, einen Überblick über die Struktur geben und die Aufgaben der einzelnen Teile verständlich erläutern.

3.2.1 Einstiegspunkt der Anwendung

Die `main`-Methode, die die Argumente des SCA2AMALTHEA-Plugins empfängt, befindet sich in der Klasse `TraverseASTMain`. Von hier aus werden alle weiteren Teilaufgaben an die zuständigen Helper-Klassen `TraverseASTMainHelpers` und `Helpers` delegiert. Das Ziel ist es, eine Liste von Argumenten für die clang-Kommandozeile zu erzeugen, die alle (für die Kompilierung sowie Generierung der Ausgaben) notwendigen Daten beinhalten. Darunter befinden sich entsprechend auch die zu analysierenden Dateien bzw. deren Pfade. sca unterstützt verschiedene Möglichkeiten, wie Dateipfade spezifiziert werden können. Beispielsweise kann ein einzelnes Verzeichnis mit Header-Dateien direkt angegeben werden. Alternativ ist auch eine Spezifizierung über eine Textdatei erlaubt, die eine Liste von unterschiedlichen Pfaden enthält. Je nach Wahl würden dann unterschiedliche Parameter übergeben. So entstehen verschiedene, mögliche Kombinationen. Im Beispiel ergibt sich je nach Variante für die Übergabe dann die Argumentliste `-pdir= -hdir=` oder `-pdir= -hlist=`.

Clang bietet in seiner Bibliothek Namespaces und Klassen für die Verwaltung von Kommandozeilenparametern an. Das Klassen-Template `llvm::cl::opt<DataType, ...>` ermöglicht es dabei in Form von Optionen, in einer Clang executable (eine Anwendung, die auf clang aufsetzt) unterstützte Kommandozeilenargumente mit Metadaten zu spezifizieren. Entsprechend nutzt sca diese Funktionalität auch. So findet sich im Quellcode beispielsweise folgende Zeile:

```
54 llvm::cl::opt<std::string> projectDirectory("pdir", llvm::
     cl::desc("Mandatory option ...Specify absolute path of
     project directory"), llvm::cl::value_desc("Project
     directory"), llvm::cl::cat(MyToolCategory), llvm::cl::
     Required);
```

Listing 3.1: TraverseASTMain.cpp

Sie gibt an, dass das Projektverzeichnis mit den C-Dateien über die Option `pdir` **verpflichtend** angegeben werden muss. Eine Assertion schlägt andernfalls fehl und die Anwendung wird mit einer Fehlermeldung beendet. Der Projektpfad, der über dieses Argument übergeben wurde, wird im Programm in der Variable `projectDirectory` gespeichert.

Da nun, wie beschrieben, mehrere Ansätze für die Argumentübergabe existieren und das sca-Projekt nicht sofort weiß, in welcher Option sich die gewünschten Daten befinden, müssen diese erst noch auf Korrektheit geprüft und entsprechend gefiltert werden. Hierfür gibt es die Methode `void handleCommandLineArguments(int argc, const char **argv)` in der Klasse `TraverseASTMainHelpers.cpp`. Diese wird von der `main` -Methode mit ihren Eingabeparametern aufgerufen. Sie zählt dann die Anzahl der Argumente und delegiert die Initialisierung von internen Variablen an weitere Untermethoden, die sich auf die Validierung ihrer jeweiliger Kombinationen von Pfadangaben spezifizieren. Auf diese Weise werden die Pfade gesammelt und dann einheitlich in der Anwendung zur Verfügung gestellt (unabhängig davon, aus welchen Optionen sie stammen, sodass dies nicht immer wieder abgefragt werden muss). Eine entsprechende Überladung des Zuweisungsoperators erlaubt es dabei, den Wert der Option direkt ohne notwendige Casts oder Methodenaufrufe in einer jeweiligen, neuen Variable zu speichern. Es wird also keine Kopie des `opt`-Objektes selbst erzeugt und eine Übergabe an die intern verwendete Variable `directoryToParse` ist unter anderem mit einem Ausdruck wie `directoryToParse = projectDirectory` möglich. (vgl. [7])

3.2.2 Verfügbare Kommandozeilenoptionen

Die folgende Tabelle zeigt eine grobe Übersicht über die in der sca-Anwendung vorhandenen Optionen. Diese können bei der Ausführung als Prozessargumente übergeben werden.

Tabelle 3.1: Verfügbare Optionen

Option	Kurzbeschreibung	Notwendig
pdir	Projektverzeichnis	ja
clist	Textdatei mit Liste von C-Dateien	nein
cdir	Verzeichnis für C-Dateien	nein
hlist	Textdatei mit Liste von Header-Dateien	nein
hdir	Verzeichnis für Headerdateien	nein
dlist	Textdatei mit Liste von defines	nein
n	Anzahl an Abarbeitungs-Threads	nein

Deutlich erkennbar sind jeweils direkt die verschiedenen Möglichkeiten für die Angabe von C- und Headerdateien. Es stellt sich daher die Frage, welche Optionen verwendet werden, wenn beliebige Kombinationen an die sca-Anwendung übergeben werden und wie sie sich unterscheiden.

Es ergibt sich für die Optionen, die C-Quelldateien angeben, folgende Prioritätenliste:

1. cdir
2. clist
3. pdir

Analog für die Headerdateien:

1. hdir
2. hlist
3. pdir

Sollte eine Option nicht angegeben sein, fällt die Auswahl auf die nächstmögliche darunter, für die Argumente übergeben wurden. In beiden Fällen kann dies maximal bis zur Selektion von `pdir` führen, da dieser Parameter zwingend angegeben werden muss und daher nicht leer sein kann.

pdir

Das bereits genannte Projektverzeichnis wird automatisch von SCA2AMALTHEA ermittelt und übergeben. Es entspricht dem Pfad des aktuellen Projekts im APP4MC-Workspace, das auch für die Modell-Generierung verwendet wird und ist obligatorisch. Dieses Verzeichnis dient als initiale Quelle für die zu übersetzenden C- und Headerdateien. Wenn keine weiteren Angaben erfolgen, werden die Dateien entsprechend der Prioritätenliste aus diesem Ordner bezogen.

cdir

Diese Option stellt die Möglichkeit bereit, einen einzelnen Ordner mit den zu übersetzenden Quellcodedateien direkt anzugeben. Die Angabe des Projektverzeichnisses mittels `cdir` wäre aufgrund der obigen Feststellungen nicht sinnvoll/notwendig, da es äquivalent ist, dieses Argument dann wegzulassen und sca auf `pdir` zurückgreifen zu lassen. In Version 0.9.8 des APP4MC, auf der diese Arbeit basiert, ist es nicht möglich, diese Option direkt über die graphische Oberfläche zu nutzen.

clist

Alternativ ist es möglich, die C-Dateien über eine Textdatei zu referenzieren, die eine Liste von zugehörigen Dateipfaden enthält. Die Pfade müssen durch einen Zeilenumbruch getrennt sein, da die `clist`-Datei bei Verwendung dieses Parameters zeilenweise ausgelesen/iteriert wird. Sollte ein Dateipfad direkt angegeben sein, wird dieser unmittelbar zum internen Vektor, der die Dateipfade sammelt, hinzugefügt. Das sca-Projekt würde zwar alternativ auch die Angabe von Ordnerpfaden unterstützen, der Java-Teil der Implementierung erlaubt es jedoch aus unbekanntem Grund lediglich, Dateipfade zu spezifizieren. Diese Option kann entsprechend über die sogenannte `Build Command Logfile` angegeben werden. Es sei angemerkt, dass der Projektordner, auch wenn er verpflichtend angegeben werden muss, bei der Verwendung von `clist` **nicht** automatisch bei der Dateisuche berücksichtigt wird, wenn seine Inhalte nicht explizit in die Liste aufgenommen wurden.

```
1 D:\Downloads\MyProject\a_source.c
2 D:\Downloads\AdditionalSources\my_source.c
```

Listing 3.2: Beispieldatei: clist.txt

hdir

Dieser Kommandozeilenparameter verhält sich wie `cdir`, spezifiziert jedoch Headerstatt C-Dateien innerhalb eines einzigen Ordners. Auch er kann im Plugin nicht direkt über die graphische Benutzeroberfläche verwendet werden.

hlist

Auch `hlist` bildet ein Analogon zum Argument `clist` und spezifiziert Headerdateien. Hier besteht die Möglichkeit, diesen auch im APP4MC-Plugin verwenden zu können. Dies ist auch durchaus sinnvoll, da sich vorallem Header der Standardbibliothek fast immer in anderen Verzeichnissen befinden und diese für die Kompilierung inkludiert/-gelinkt werden müssen. Auch hier gilt wieder, dass die Inhalte des Projektordners nicht automatisch berücksichtigt werden. Zwar ist dieses Verhalten in der internen Methode grundsätzlich implementiert, aber der Aufruf mit beiden Parametern (Textdatei und Projektordner) erfolgt in der Methode `createHeaderDirectoryList` nie, wodurch es unbenutzt bleibt.

```
1 C:\Program Files (x86)\Microsoft Visual Studio\2017\
    Community\VC\Tools\MSVC\14.15.26726\include
2 C:\Program Files (x86)\Windows Kits\10\Include
    \10.0.17134.0\ucrt
3 D:\Downloads\AdditionalHeaders\my_header.h
4 D:\Downloads\AdditionalHeaders\different_header.h
```

Listing 3.3: Beispieldatei: hlist.txt

dlist

Mit dieser Option können gemäß Dokumentation `define`s spezifiziert werden. Diese werden entsprechend vom Präprozessor abgearbeitet und damit zugehörige Codeteile in der resultierenden Translation Unit definiert oder ignoriert. Ein genauer Verwendungszweck konnte im Rahmen der in dieser Arbeit getätigten Codeanalyse nicht ermittelt werden. Da in Embedded Code Konfigurationen für unterschiedliche Hardwarevarianten oder Funktionalitäten oftmals über `define`s spezifiziert/gesteuert werden, liegt es jedoch nahe, dass sich hinter dieser Option auch dieser Einsatzzweck verbirgt. Auf diese Weise kann dann die Unterstützung diverser Softwareteile auf dem Steuergerät eingeschränkt werden. Denkbar wäre dies bei unterschiedlichen Ausstattungsvarianten oder Features, die schnellere Hardware erfordern. Damit in einem solchen Fall das generierte Softwaremodell dann auch mit der spezifizierten Hardwarekonfiguration des AMALTHEA-Modells übereinstimmt, muss dieses darüber konfiguriert werden.

n

Um die Dateianalyse zu optimieren, wird diese mittels Multithreading parallelisiert. Dabei kann über diesen Parameter die Anzahl der zu verwendenden Threads angegeben werden. Sollte dieser Wert größer oder gleich der Anzahl der Dateien sein, wird für jede Datei genau ein eigener Thread ausgeführt. Für den Fall, dass weniger Threads als Dateien zur Verfügung stehen, werden diese gruppiert. Mehr hierzu dann in der Sektion zu Vorbereitung der Analyse. Die Standardanzahl an Threads beträgt 8.

3.2.3 Vorbereitung der Analyse

Nachdem nun eine Übersicht der Optionen zur Verfügung steht, können die weiteren Hauptschritte des sca-Projektes erläutert werden. Sobald die Eingabeargumente gefiltert wurden, wird die Methode `void createListOfFilesToAnalyse()` in der Klasse `TraverseASTMainHelpers.cpp` aufgerufen. Diese ist nun für das Bauen der Liste mit den zu analysierenden C-Dateien verantwortlich. Wie bereits beschrieben, wird die Liste je nach verwendetem Kommandozeilenparameter auf unterschiedliche Weise befüllt. Die Details zu den einzelnen Abläufen wurden in den Erklärungen zu pdir, cdir und clist bereits genannt. Es findet insofern nur eine Fallunterscheidung nach diesen drei Optionen und eine Ausführung der jeweiligen Implementierung statt. Zusätzlich wird die Zahl der auszuführenden Threads im Falle `#Dateien < #Threads` noch auf `#Dateien` beschränkt (siehe Erklärung zu n). Die gefundenen Pfade werden in einer globalen Variable `std::vector<std::string>* fileList` gespeichert.

Anschließend erfolgt die Zuordnung/Gruppierung von Dateien zu/nach Threads. Hierfür existiert in der Klasse `Helpers.cpp` eine Methode `createFilesGroupToAnalyze (vector<std::string>* listOfFiles)`, die die Verteilung vornimmt. Gespeichert werden die Gruppen von Dateien dann in einer Variable `std::vector<std::vector <std::string>*> filesForThreads`. Jeder Eintrag des äußeren Vektors entspricht dabei einem Job/Thread, repräsentiert durch dessen abzuarbeitende Liste an Dateien im inneren Vektor. Um die Informationen, die später beim Durchlaufen des Syntaxbaumes gesammelt werden, verwalten zu können, existiert eine Klasse `TraversingData` im Ordner `data_structure`. Für jeden Thread wird für die Aggregation aus diesem Grund ein Objekt dieser Klasse erstellt und in einem Array gespeichert. Zusätzlich wird jedem Dateipfad das in seinem Thread zugehörige `TraversingData`-Objekt zugeordnet. Hierfür existiert eine globale Map `traversingDatas` [sic!].

Bevor nun die Analyse selbst angestoßen wird, muss noch die Liste für die Headerdateien gebaut werden. Dies übernimmt die Methode `createHeaderDirectoryList` in der Klasse `TraverseASTMainHelpers.cpp`. Die Analogie zu den C-Dateien bei der Verwaltung von Headerdateien wurde in dieser Arbeit bereits festgestellt. Entsprechend liegt es auch nahe, dass die Implementierung an diesem Punkt sehr ähnlich ist. Zusätzlich werden in dieser Methode direkt noch die clang-Kommandozeilenparameter `-ferror-limit=0` für die Aufhebung des Fehlerzahllimits und `-w` zur Unterdrückung aller Warnungen an die Header-Liste angehängt. (vgl. [8]) Ebenso werden die durch `dlist` definierten `defines` angefügt und das Projektverzeichnis sowie der Pfad zur sca-Executable an den Anfang der Liste eingegliedert. Die Headerliste dient damit nicht nur als Sammlung von Dateipfaden, sondern wird direkt als Speicherliste der fertigen clang-Kommandozeilenargumente verwendet, die dann auch übergeben werden. Aus diesem Grund werden alle Headerpfade auch direkt mit der Option `-I` versehen, welche besagt, dass diese inkludiert werden sollen.

3.2.4 Ausführung der Codeanalyse

Die Abarbeitung beginnt dann durch den Aufruf von `createAndStartJobs`. Hierbei wird für jeden Eintrag in `filesForThreads` (definiert durch die abzuarbeitende Liste von Dateien), ein entsprechender Thread (Job) für die parallelisierte Abarbeitung erstellt. Als Funktionszeiger für die auszuführende Funktion erhält er die Methode `executeWorker` eines zuvor instanziierten `TraverseASTWorker`-Objektes. Dieses wiederum erhält als Parameter ein `CommonOptionsParser`-Objekt, das sich intern um die Auswertung der übergebenen Optionen für die clang-Kommandozeile kümmert und mit den Inhalten der Header-Liste (Argumentliste) befüllt wird. Zusätzlich wird dem Worker die Liste seiner zuständigen C-Dateien mitgegeben. Die Methode `executeWorker()` startet dann die Ausführung des ClangTools, welches unter anderem die statische Codeanalyse und Kompilierung übernimmt. An dieser Stelle kommen dann die eigenen Klassen für die Analyse des abstrakten Syntaxbaumes (AST) zum Einsatz. Ein AST bezeichnet dabei eine Datenstruktur, die den in programmiersprachenunabhängige Einzelteile (Tokens) zerlegten Quellcode in einem Baum repräsentiert, der mit den gängigen Traversierungsalgorithmen (z.B. Tiefen- oder Breitensuche) durchlaufen werden kann. clang bietet hier einen sehr hohen Abstraktionsgrad, der es beim Ausführen des ClangTools ermöglicht, über `newFrontendActionFactory<T>()` `.get()` einen eigenen Besucher (i.S.d. Vistor Patterns) zu registrieren, der die Traversierungsdaten dann abarbeiten kann. Der Template-Parameter `T` entspricht dabei der eigenen Besucher-Klasse, die von `clang::ASTFrontendAction` erben muss. In der Datei `/algorithms/TraverseASTClassAction.h` des sca-Projektes findet sich hierfür folgende Definition:

```
213 class FindNamedClassAction : public clang::
      ASTFrontendAction {
214 protected:
215
216 public:
217
218   virtual std::unique_ptr<clang::ASTConsumer>
      CreateASTConsumer(clang::CompilerInstance &Compiler,
      llvm::StringRef InFile)
219   {
220     try{
221       lock_allTraversingDataMap.lock();
222       TraversingData* t = traversingDatas[InFile.str()];
223       lock_allTraversingDataMap.unlock();
224       t->currentlyAnalyzedFile = InFile.str();
225       printCurrentAnalyzedFile(InFile.str(), t);
226       t->analyzedFiles[InFile.str()] = 1;
227       return  std::unique_ptr<clang::ASTConsumer>(new
      FindNamedClassConsumer(&Compiler.getASTContext(), t));
228     }
```

```
229      catch (...){
230        cout << "Sudhindra error occured in
       findnamedclassaction" << endl;
231      }
232    }
233 };
```

Listing 3.4: TraverseASTClassAction.cpp

In Zeile 213 ist die genannte Veerbungsbeziehung zu sehen. Die in clang vorhandene Klasse `clang::ASTFrontendAction` definiert die Methode `std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(...)`, die in dieser Unterklasse überschrieben wird. Sie erlaubt es, im Kontext der aktuellen Analyse einen `clang::ASTConsumer` zu instanziieren, der den zugehörigen abstrakten Syntaxbaum traversieren lässt. Hierzu erhält dieser die aktuell bearbeitete Datei (Parameter `InFile`) und den zuständigen clang-Compiler in Form einer Referenz. Unter Vorbereitung der Analyse wurde beschrieben, dass die Map `traversingDatas` zu jeder Datei das im Thread-Kontext zugeordnete `TraversingData`-Objekt speichert, das für die Sammlung der Codeinformationen innerhalb dieses Jobs dient. Entsprechend speichert Zeile 222 dessen Zeiger in einer lokalen Variable und setzt dann in Zeile 224 die aktuell analysierte Datei auf die übergebene. Die Map `analyzedFiles` speichert zusätzlich für die aktuelle Datei einen numerischen Wert (`int`), wobei `1` bedeutet, dass diese analysiert wurde. `-1` steht für eine Analyse mit aufgetretenen Fehlern und `0` dafür, dass die Datei noch nicht verarbeitet wurde (entspricht dem Standardwert bei Initialisierung). Schließlich wird ein intelligentes Zeigerobjekt mit einer neuen Instanz der Klasse `FindNamedClassConsumer`, die selbst wiederum vom beschriebenen `ASTConsumer` erbt, zurückgegeben. Dieser erhält den aktuellen Syntaxbaum und das extrahierte `TraversingData`-Objekt.
Der `FindNamedClassConsumer` dient dabei nur als Zwischenklasse und initiiert die Traversierung, indem er diese an ein intern verwaltetes Besucher-Objekt delegiert.

```
184 class FindNamedClassConsumer : public clang::ASTConsumer {
185 public:
186   explicit FindNamedClassConsumer(clang::ASTContext *
      Context, TraversingData* traversingData)
187     : Visitor(Context, traversingData) {}
188
189   virtual void HandleTranslationUnit(clang::ASTContext &
      Context) {
190     // Traversing the translation unit decl via a
      RecursiveASTVisitor will visit all nodes in the AST.
191     try{
192       Visitor.TraverseDecl(Context.getTranslationUnitDecl()
      );
193     }
194     catch (...){
```

```
195         cerr << "An error happened in HandleTranslationUnit"
      << endl;
196       }
197     }
198 private:
199     // A RecursiveASTVisitor implementation.
200     FindInformationClassVisitor Visitor;
201
202 };
```

Listing 3.5: FindNamedClassConsumer.cpp (verkürzt)

Die Deklaration dieses Besucher-Objektes (`FindInformationClassVisitor`) ist in Zeile 200 zu sehen. Im Konstruktor von `FindNamedClassConsumer` erfolgt direkt eine Weiterleitung der übergebenen Parameter an den `Visitor`. Die überschriebene Methode `HandleTranslationUnit` dient der Verarbeitung der aktuellen Datei mit ihrem zugehörigen `ASTContext` (Syntaxbaum). Der Begriff `Translation Unit` bezeichnet dabei eine zu übersetzende Quelldatei (also bspw. C-Datei), nachdem sie durch den Präprozessor bearbeitet wurde. Sie entspricht also der unmittelbaren, fertig vorbereiteten Eingabe für den Compiler, der aus ihr dann eine Objektdatei erzeugt. (vgl. [9], S. 9) Eine solche Translation Unit dient im Kontext von clang und des Syntaxbaumes als Wurzelelement, da sie den umrahmenden Behälter für die zu analysierenden Codeinhalte darstellt. Aus diesem Grund bietet der übergebene Kontext eine Methode `getTranslationUnitDecl`, der den entsprechenden Wurzelknoten im Baum in Form eines Deklarationsobjektes kennzeichnet/zurückgibt. Somit kann dieser dann besucht und ausgehend von diesem der restliche Baum rekursiv abgearbeitet werden. Dies ist auch durch den Kommentar in Zeile 190 beschrieben. Als letzten Schritt benötigt man daher nur noch eine solche, rekursive Besucherklasse, die in clang durch die Oberklasse `RecursiveASTVisitor` beschrieben wird. Zeile 192 zeigt den Einstiegspunkt in die Traversierung mit der Methode `TraverseDecl` der im folgenden definierten `FindInformationClassVisitor`-Klasse.

```
38 class FindInformationClassVisitor
39   : public clang::RecursiveASTVisitor <
     FindInformationClassVisitor > {
40 public:
41   explicit FindInformationClassVisitor(clang::ASTContext *
     Context, TraversingData* traversingData) : Context(
     Context) {
42     this->traversingData = traversingData;
43   }
44
45     // ...
46
47   bool VisitFunctionDecl(clang::FunctionDecl* decl);
```

```
48
49  bool VisitVarDecl(clang::VarDecl* varDecl);
50
51  bool VisitDeclStmt(clang::DeclStmt* declStmt);
52
53  bool VisitImplicitCastExpr(clang::ImplicitCastExpr*
     implicitCastExpr);
54
55  //bool VisitTypeDefDecl(clang::TypedefDecl* typedefDecl);
56  bool VisitTypeDecl(clang::TypeDecl* typeDecl);
57  bool VisitRecordDecl(clang::RecordDecl* recordDecl);
58  bool parseFieldDecl(clang::FieldDecl* fieldDecl);
59  //bool VisitEnumDecl(clang::EnumDecl* enumDecl);
60  bool VisitEnumConstantDecl(clang::EnumConstantDecl*
     enumConstantDecl);
61
62  // ...
```

Listing 3.6: FindInformationClassVisitor.cpp (verkürzt)

Auf jede Methode und jedes Attribut detailliert einzugehen und deren Funktionsweise zu erklären, würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund soll lediglich eine grobe Übersicht geschaffen werden, wie die Traversierung abläuft, Daten extrahiert und gespeichert werden und welche Rolle das Besucher-Softwaremuster dabei spielt.

Zu sehen ist, dass clang in der Basisklasse `RecursiveASTVisitor` einige Methoden bereitstellt, die die Analyse unterschiedlicher Arten von Codeteilen erlaubt. Beim Aufruf von `TraverseDecl` wird, wie bereits analysiert, eine Traversierung des kompletten Syntaxbaumes angestoßen. Intern ruft der `RecursiveASTVisitor` dann je nach angetroffenem Knoten eine entsprechende, zugehörige Besuchermethode auf. Wenn beispielsweise beim Ablaufen eine Funktionsdeklaration vorliegt, erfolgt ein Aufruf von `VisitFunctionDecl(clang::FunctionDecl* decl)`. Die eigene Implementierung in der Besucher-Unterklasse kann dann mit diesem Parameter Informationen sammeln. Im `FindInformationClassVisitor`-Objekt werden diese dann gefiltert und an das zugeteilte `TraversingData`-Objekt, das ebenso in einer privaten Variable abgespeichert ist (siehe Zeile 42), in Form eigener Datenstrukturen übergeben. Für Funktionsdeklarationen steht etwa die Klasse `FunctionEntry` zur Verfügung, die unter anderem Daten zu Dateiname, Name, Rückgabetyp und Parametern enthält. Sie ist selbst wiederum in der Klasse `FunctionAndCallees` enthalten, welche eine Funktion komplett beschreibt und daher unter anderem noch Listen von aufgerufenen Funktionen und globalen Variablen enthält. Im Kapitel XML-Ausgabe der gesammelten Daten wird die Existenz und Aufgabe weiterer Strukturen im Kontext noch genauer beschrieben. Diese wurden definiert, da nur bestimmte Details für die Generierung des AMALTHEA-Softwaremodells notwendig sind (insofern also aus den LLVM-Datentypen extrahiert

werden) und diese für das Speichern und Einlesen bereits in einer von clang bzw. LLVM unabhängigen Form vorliegen sollen. Zum Zeitpunkt dieser Arbeit unterstützt das Plugin die Umsetzung von Deklarationen und Aufrufen von Variablen und Funktionen. Letztere werden außerdem in einer Struktur gespeichert, die die Implementierung in Form des sequentiellen Ablaufs wiedergibt. Diese Beschreibung beschränkt sich dabei auf die gerade erwähnten, unterstützen Inhalte. Ersichtlich sind diese dann im Modell selbst im Kapitel Praktischer Test. Verzweigungen (`if-else/switch-case`) oder Schleifen (`for/while`) werden aktuell nicht erkannt, obwohl das AMALTHEA-Metamodell theoretisch Möglichkeiten bietet, diese zu repräsentieren. Ziel dieser Arbeit soll es daher im Kapitel Erweiterung auch sein, eine solche Funktionalität zusätzlich in das SCA2AMALTHEA-Plugin zu integrieren.

3.2.5 XML-Ausgabe der gesammelten Daten

Nachdem die Daten in den jeweiligen `TraversingData`-Objekten gespeichert wurden, müssen diese nun aufgrund ihrer Verteilung auf die verschiedenen Threads zunächst wieder zusammengeführt und dann im gewählten XML-Format abgespeichert werden. Hierzu wird die Methode `mergeDataFromDifferentThreadsToFirst` der Klasse `TraverseASTMainHelpers` aufgerufen, die alle verwalteten `TraversingData`-Instanzen iteriert und deren Inhalte in das erste Objekt (`allTraversingData.at(0)`) speichert. Da alle Daten der `TraversingData`-Klasse in listenartigen Strukturen gespeichert werden, können diese ebenso mit einem Iterator abgerufen und im ersten Element wieder angehängt werden. Abschließend wird durch den Aufruf der Methode `printIRInformation` die interne Methode `printCallTreeToXMLAdvanced` der Klasse `Output` angstoßen, die die XML-Datei final erzeugt und befüllt. Der Aufbau enthält dabei zunächst eine Liste von Containern für jede Datei, wobei jeder einzelne wiederum eine Liste von Funktionen verfügbar macht. Eine Funktion wird gespeichert mit der zugehörigen Zeile und Spalte im Quellcode, deren Namen und einer, nach Zeilen geordneten Liste an Zugriffen auf globale Variablen und Funktionen, die sie durchführt. Für das Ausgeben dieser Zugriffsliste gibt es eine eigene Funktion `printFunctionContentInOrderAdvanced`. Als Basisstruktur dient die bereits angesprochene Klasse `FunctionAndCallees`. Die Aufrufsdaten bezieht diese jeweils aus der zugehörigen `list<CallEntry*>` und `list<GlobalVariableAccess*>`. `CallEntry` enthält dabei einen Pointer auf die `FunctionEntry`-Instanz für die aufgerufene Funktion sowie die Stelle im Quellcode mit Zeile und Spalte. Analog funktioniert die `GlobalVariableAccess`-Klasse, nur dass diese stattdessen auf die zugehörigen `GlobalVariable` zeigt.

`FunctionAndCallees` speichert im Modell ebenso eine Liste von lokal definierten Variablen und Kontrollflussgraph-Pfaden und -Zyklen (entspricht Verzweigungen und Schleifen), welche jedoch nicht ausgegeben werden. Auf diese wird in Kapitel 6 noch näher eingegangen. Es folgt dann noch eine Liste von globalen Variablen und den deklarierten Typedefs in der XML-Datei.

```
21 class GlobalVariableAccess
22 {
23 private:
24   GlobalVariable* globalVariable;
25   std::string accessType;
26   unsigned line;
27   unsigned col;
28   unsigned basicBlockID;
29 // ...
```

Listing 3.7: GlobalVariableAccess.h (verkürzt)

```
20 class CallEntry
21 {
22 private:
23   FunctionEntry* functionID;
24   unsigned callLine;
25   unsigned callColumn;
26   unsigned basicBlockID;
27 // ...
```

Listing 3.8: CallEntry.h (verkürzt)

Eine direkte Verwendung der LLVM IR erfolgt in diesem Fall also gar nicht. Das sca-Projekt setzt mit clang nur auf dem direkten Frontend der LLVM-Infrastruktur auf und nutzt dessen statische Codeanalyse zur Generierung eines proprietären XML-Formates, das die gesammelten Daten in einem für das SCA2AMALTHEA-Plugin einfach zu verarbeitenden Modell hält. Dennoch lässt sich im Quellcode des Plugins teilweise der Begriff “Intermediate Representation” finden, der etwas anders verwendet wird und im Kontext die eigene Datenstruktur bzw. das XML-Format bezeichnet (auch “SCA IR” oder “SCAIR” genannt). Die eigentliche Bytecode-IR im Sinne von LLVM ist hiermit nicht gemeint.

3.3 Implementierung im Java-Teil

Da nun ersichtlich ist, wie das Backend des Plugins funktioniert, bleibt noch zu klären, wie dieses aufgerufen wird und die Daten, die von diesem über die XML-Datei zurückgeliefert werden, vearbeitet und im APP4MC schlussendlich als Modell angezeigt werden. Dies übernimmt der Java-Teil des Plugins, der im APP4MC installiert wird. Dieser besteht aus einer Vielzahl von Paketen, die alle Aufgaben im Frontend übernehmen. Hierunter fallen die graphische Benutzeroberfläche, das Erzeugen und Laden der IR sowie die daraus resultierende Erzeugung des AMALTHEA-Modells.

3.3.1 Generierung der SCA-IR

Für das Starten des sca-Prozesses und die Übergabe der benötigten Argumente ist das Paket `org.eclipse.app4mc.sca2amalthea.llvm` verantwortlich. Der Einstiegspunkt mit der `main`-Methide befindet sich in der Klasse `SCA2AMALTHEACmdLauncher` im Unterpaket `headless`. Sie empfängt dabei als Argumente die angegebenen Optionen innerhalb des Wizards (d.h. die unterschiedlichen Pfade, die für die sca-Argumente gewählt wurden) und delegiert die Generierung einer `SCA2AMALTHEAStarterProperties` -Instanz, welche als Modell für das Abspeichern dieser dient, an die definierte Methode `parseCommandLineParameters` der Klasse `CmdLineArgumentParser`. Anschließend wird die Klasse `GenerateAmaltheaModelFromLLVM` instanziiert, die StarterProperties an diese übergeben und die Funktion `run` für die Ausführung der Aufgabe aufgerufen. Innerhalb dieser werden zunächst die übergebenen Optionen auf Validität geprüft. Anschließend erfolgt direkt der Start des sca-Projektes, wobei dieses Verhalten auch in eine andere Klasse `GenerateTraverseAstOutput` des Unterpaketes `starter` ausgelagert wurde. Hier für erfolgt ein Aufruf der entsprechend dort definierten Methode `runTraverseAst`:

```
52 public void runTraverseAst() throws IOException,
     InterruptedException {
53     String[] cmdLineArray = new String[this.commandLineList
     .size()];
54     cmdLineArray = this.commandLineList.toArray(
     cmdLineArray);
55     ProcessExecutor processExecutor = new ProcessExecutor(
     this.llvmProperties.getGenDirectory(),
56         this.llvmProperties.getLogDirecotry() + "/stdOut.
     log", this.llvmProperties.getLogDirecotry() + "/stdErr.
     log",
57         cmdLineArray);
58     // ...
59     processExecutor.runProgramm();
60     // ...
61   }
```

Listing 3.9: GenerateTraverseAstOutput.java (verkürzt)

In Zeile 278 wird ein `ProcessExecutor`-Objekt erzeugt, welches intern einen `java.lang.ProcessBuilder` benutzt, die Kommandozeilenargumente in Form eines Arrays an diesen übergibt, sca ausführt und dann auf die Beendigung des Prozesses wartet. Die Argumente stehen dabei in Form einer Liste zur Verfügung (siehe deren Verwendung in Zeile 53 von Codeausschnitt 3.9), welche durch eine interne Methode `generateCommandLineList` aus den `LLVMStarterProperties` erzeugt wurden. Der Aufruf dieser Methode findet dabei im Konstruktor von `GenerateTraverseAstOutput` statt.

3.3.2 Bau des AMALTHEA-Modells

Sobald die sca-Executable ohne Fehler ausgeführt wurde, steht die von diesem erzeugte XML-Datei (`XMLCalltree.xml`) mit der SCA-IR in einem Ordner `_gen` innerhalb des Projektverzeichnisses zur Verfügung. Die Elemente der SCA-IR haben auch im Java-Teil zugehörige Modell-Klassen. Das Paket `org.eclipse.app4mc.sca2amalthea.ir.scair` enthält diese. Da die SCA-IR die Daten bereits in angepasster Form (z.B. für den Funktionsablauf) enthält, sind diese nicht äquivalent zu den Klassen im sca-Projekt, sondern bereits auf das Format der XML-Datei angepasst.

Für den nächsten Schritt der Modelltransformation kommen nun weitere Pakete des Plugins zum Einsatz. Zunächst wird die SCA-IR geladen und in ein Objekt der Klasse `SCAResource` deserialisiert. Im nächsten Schritt wird überprüft, ob eine Datei mit Informationen zu Tasks angegeben wurde. Ein `Task` unterliegt dabei im Gegensatz zu einem `Runnable` direkt dem Betriebssystem und wird von diesem ausgeführt. (vgl. [10]) Dies ist relevant, da deren Ausführung zeitlich gesteuert über einen Scheduler funktioniert und im Betriebssystemmodell entsprechend hinterlegt sein muss. Falls diese spezifiziert wurde, wird das Laden an die Methode `getTasksInformation` der Klasse `SchedulingInformationLoader` (Paket `org.eclipse.app4mc.sca.scheduling.loader`) übergeben, die ihrerseits ein `OsConfModel` zurückgibt, welches lediglich die Aufgabe hat, Task-Daten zu halten. Um die Funktionen, die im Modell hinzugefügt werden sollen, einheitlich behandeln zu können, macht es keinen Sinn jedes Mal zusätzlich zu überprüfen, ob eine entsprechende Task-Definition vorhanden ist, damit man sie nicht als `Runnable` deklariert. Besser ist es, diesen einen Typ zuzuweisen, anhand dessen man die Art der Funktion identifizieren kann. Da jedoch die Task-Informationen erst hier in das Plugin geladen werden und damit nicht bereits in der SCA-IR zur Verfügung stehen können, muss diese zunächst noch überarbeitet werden. Das erledigt die Klasse `SCAIRModelEnrichmentUtils` mit der Methode `markTasksIsrsRunnablesInModel`, die das Modell sozusagen mit diesen Informationen "anreichert". Hierbei werden alle ausgelesenen Funktionen iteriert und für jede in der Task-/ISR-Liste, die im `OsConfModel`-Objekt gespeichert wurde, überprüft, ob sie dort separat angegeben wurde. Falls ja, wird ihr Typ entsprechend angepasst. Die Funktions-Klasse hält hierfür ein `EFunctionTypeEnum`-Objekt.

Analog wird die Angabe einer Datei mit Lock-Informationen geprüft, die angibt, welche Funktionen innerhalb von Sempaphoren (Locks) verwendet werden oder direkte Semaphorenaufrufe sind. Auch diese müssen entsprechend im Betriebssystemmodell hinterlegt werden und stellen somit keine regulären Runnables dar. Eine Klasse `LockDefinition` übernimmt dabei das Laden und Halten der Daten.

Die eigentliche Transformation der SCA-IR in ein tatsächliches AMALTHEA-Modell findet im nächsten Schritt im Paket `org.eclipse.app4mc.sca2amalthea.exporter` statt. Das AMALTHEA-Modell selbst wird durch eine Klasse `Amalthea` repräsentiert, welche in der Klasse `SCAToAmaltheaExporter` von der bereitgestellten Methode

`amaltheaTransformation(SCAResource, LockDefinition, OSConfModel)` zurückgegeben wird.

Die `Amalthea`-Klasse entstammt dabei nicht SCA2AMALTHEA, sondern ist in einem (im APP4MC integrierten) Plugin `org.eclipse.app4mc.amalthea.model` definiert. Die Transformation besteht dabei aus drei Teilen. Die für das Betriebssystemmodell, Komponentenmodell und schlussendlich das Softwaremodell. Letztere findet in einer ausgelagerten Klasse `SwModelTransformer` statt, die die Elemente der Ressource in unterschiedlichen Methoden filtert und sammelt. Das Softwaremodell selbst wird ebenso durch eine eigene Klasse `SWModel` repräsentiert, die im `Amalthea`-Objekt assoziiert ist. Die Klassen `OsModelTransformer` und `ComponentModelTransformer` funktionieren analog. Die `AmaltheaFactory` stellt dabei die Methoden bereit, um neue Elemente des Metamodells zu erstellen. In den jeweiligen Routinen der Transformer-Klassen wird die SCA-IR, die in Form der übergebenen `SCAResource` vorliegt, dann Schritt für Schritt durchgegangen und in ausgewählten Handler-Methoden werden dann die Modell-Elemente nach und nach hinzugefügt, bis das jeweilige Teil-Modell fertig ist. Beispielhaft wäre die Methode `transformLabels` zu nennen, die alle Labels aus den IR-Daten filtert und entsprechend AMALTHEA-Label-Objekte erzeugt. Dasselbe passiert dann für Funktionsaufrufe und Semaphorenzugriffe.

Sobald das Modell fertig initialisiert wurde, werden noch alle Tasks, die mit dem `Cycle ONCE` angegeben sind, mit einem Tag als `Ini`-Task markiert (vgl. Abbildung 3.1).

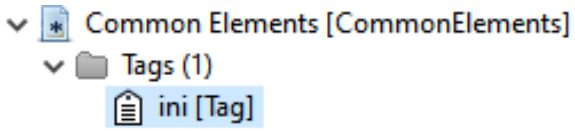


Abbildung 3.1: `ini`-Tag für Tasks

Abschließend wird das fertige Modell in die Datei `amaltheaModelFromLLVM.amxmi` serialisiert, die dann auch im APP4MC in der hierarchischen Listenform betrachtet werden kann.

4 Praktischer Test

Um nun das SCA2AMALTHEA-Plugin zu testen, wird nur das APP4MC und ein geeignetes C-Projekt benötigt. Da AMALTHEA hauptsächlich im automobilen Sektor eingesetzt wird, soll für den praktischen Test dieser Arbeit als Beispiel ein kleines, KI-basiertes Fahrzeug-Simulationsprojekt namens Need4Stek[1] verwendet werden. Es kann jedoch im Prinzip jedes beliebige C-Projekt für die Analyse herhalten. Etwa wäre es in der Realität für dieses Beispiel denkbar, dass ein Fahrzeug seine erfasste Umgebung visualisiert und in einem erzeugten 3D-Modell Kollisionserkennung für die Generierung von Fahrinstruktionen durchführt. Man treffe also nun die Annahme, dass dies in Echtzeit geschehen muss und der Quellcode daher mit der AMALTHEA-Plattform für einen Mehrkernprozessor optimiert werden soll.

Abbildung 4.1: Need4Stek Demo

[1] https://github.com/ronanboiteau/Need4Stek

4.1 Anlegen eines Modellierungsprojektes

Im ersten Schritt muss hierfür das APP4MC mit installiertem SCA2AMALTHEA gestartet werden. Dann muss im Model-Explorer ein neues Modellierungs-Projekt mit `Rechtsklick > New... > Modeling Project` angelegt werden (vgl. Abbildung 2.13). Dieses wird in dieser Arbeit `Simulation` genannt. Sobald das Projekt angelegt ist, müssen die C-Dateien spezifiziert werden. Dazu sollten die Inhalte in das Projektverzeichnis kopiert werden. In diesem Fall wird also der Inhalt des geklonten Repositories nach `APP4MC-Workspace > Simulation` kopiert, was auch per Drag&Drop möglich ist. Alternativ wäre es auch möglich, alle Pfade zu den C-Dateien einzeln in der BuildCommand-Logdatei anzugeben (siehe clist), was jedoch mit erheblichem Aufwand verbunden wäre. Somit stehen diese dann im Projektverzeichnis (`pdir`) zur Verfügung. Anschließend kann der Wizard mit einem Rechtsklick über `SCA Tools > Generate AMALTHEA` geöffnet werden (vgl. Abbildung 2.15).

4.2 Softwaremodell

Der Pfad zu der sca-Executable sollte durch die Schritte im Kapitel Einrichtung des Plugins bereits eingetragen sein. Nun sollten noch die Pfade zu den Headerdateien angegeben werden. Einmal sollte hier daher das Projektverzeichnis selbst nochmals hinzugefügt werden (siehe hlist). Dann müssen die Header-Dateien der Standardblibothek lokalisiert werden. Deren Speicherort kann sich von System zu System unterscheiden. Hauptsächlich liegen diese in den Windows SDK-Ordnern sowie den `VC`-Ordnern der jeweiligen Visual Studio-Installation. Welche Headerdateien nicht gefunden wurden, kann nach der Generierung auch dem Fehlerlog entnommen werden, um diese jeweiligen Dateien dann noch ausfindig zu machen. Im für diese Arbeit verwendeten Rechner liegen die notwendigen Dateien beispielsweise unter `C:\\Program Files (x86)\\Windows Kits\\10` und in `C:\\Program Files (x86)\\Microsoft Visual Studio 14.0\\VC`.

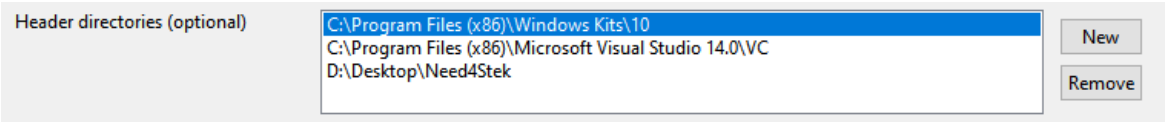


Abbildung 4.2: Header-Verzeichnisse

4.3 Scheduling- und Betriebssystemmodell

In dieser Gruppe können noch zusätzliche Informationen spezifiziert werden, die für das Betriebssystem- und Scheduling-Modell relevant sind. Die Inhalte müssen dabei jeweils im CSV-Format vorliegen.

4.3.1 OS Lock Function File

Die `OS lock function file` definiert die Funktionen, die Semaphorenaufrufe (Locks) sind oder innerhalb von Semaphorenfunktionen verwendet werden. Damit werden diese nicht als Runnables gekennzeichnet, sondern dem OS-Modell zugeordnet.

Das Format hat dabei folgende Form:

Tabelle 4.1: Lock-Datei-Format

Name	Akquierierungsfkt.	Freigabefunktion	Semaphoren-Typ
my_lock	get_my_lock	release_my_lock	
excl_lock	get_excl_lock	release_excl_lock	EXCLUSIVE_LOCK
# Kommentar	Zeilen mit #	werden ignoriert	

Nicht angegebene Semaphoren-Typen werden automatisch zu einem Spin-Lock (normaler Lock).

Ein sinnvoller Testfall ergibt sich im Beispiel nicht. Ersichtlich wäre jedoch, dass der Zugriff auf die Lock-Methode nicht als `RunnableCall`, sondern `SemaphoreAccess` im entsprechenden `ActivityGraph` ausgewiesen würde. Dies ist beispielhaft (unabhängig vom Testprojekt dieser Arbeit) in den Abbildungen 4.3 und 4.4 ersichtlich, für die ein Semaphor namens `memcpy_s_lock` definiert wurde. Die zugehörigen, in der Datei definierten Namen der Akquierierungs- und Freigabefunktionen werden dann automatisch beim Iterieren erkannt und an der entsprechenden Aufrufstelle als `SemaphoreAccess` mit dem jeweiligen Modus (`request` oder `release`) hinterlegt.

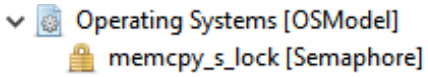


Abbildung 4.3: Semaphor im Betriebssystemmodell

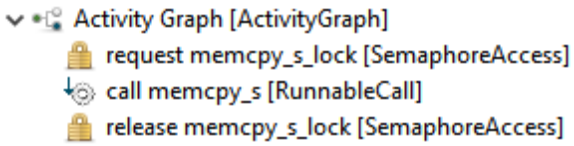


Abbildung 4.4: Zugriff auf einen Semaphor im `ActivityGraph`

4.3.2 Task Scheduling File

Die `Task scheduling file` wird benötigt, um ISR (Unterbrechungsroutinen) und Tasks zu definieren. Ein Task unterliegt dabei im Gegensatz zu einem Runnable direkt dem Betriebssystem und wird von diesem ausgeführt. (vgl. [10])
Dies ist relevant, da dies zeitlich gesteuert über einen Scheduler passiert und im Betriebssystemmodell entsprechend hinterlegt sein muss. Im Beispiel entspricht der Einstiegspunkt der Anwendung (`main`) einem Task. Alle Runnables (Methoden der Anwendung) werden dann geschachtelt, innerhalb dieses Tasks aufgerufen. Da Unterbrechungsroutinen auch speziell behandelt werden müssen, können diese hier ebenso spezifiziert werden. Für das gewählte Projekt soll dieser Schritt mangels Kontext jedoch vernachlässigt werden. Die Datei könnte daher im Beispiel so aussehen:

Tabelle 4.2: Scheduling-Datei-Format

Typ	Name	Zyklus
SOFTWARE	main	NONE
#ISR	service_routine	ONCE

Die mit dem Doppelkreuz versehene Zeile wird beim Einlesen der Datei wieder ignoriert. Sie soll aufzeigen, wie eine ISR definierbar wäre.

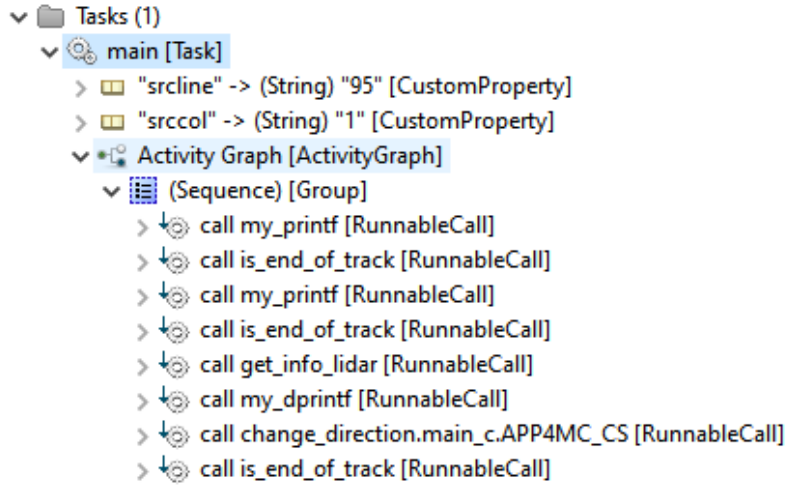


Abbildung 4.5: main-Task im Softwaremodell

Abbildung 4.5 zeigt, dass dem Softwaremodell eine neue Kategorie `Tasks` hinzugefügt wird, sobald Informationen mit dieser Datei spezifiziert werden. Im Beispiel wurde entsprechend die `main`-Methode als `Task` (`SOFTWARE`) gekennzeichnet und wird dann auch als solcher ausgewiesen. In den Runnables kommt sie dann nicht mehr vor.

4.4 Modell-Generierung

Nun kann die Modell-Generierung mit einem Klick auf `Finish` bereits angestoßen werden und je nach Größe des Projektes etwas Zeit in Anspruch nehmen. Im Beispiel waren dies ungefähr 10 Sekunden.

Abbildung 4.6: AMALTHEA Generierungs-Fortschritt

Sobald die Generierung abgeschlossen ist, erscheint eine Benachrichtigung. Das fertige Modell befindet sich im Ordner `_bin/sca2amalthea`. In Abbildung 4.7 ist der Aubau bereits erkennbar.

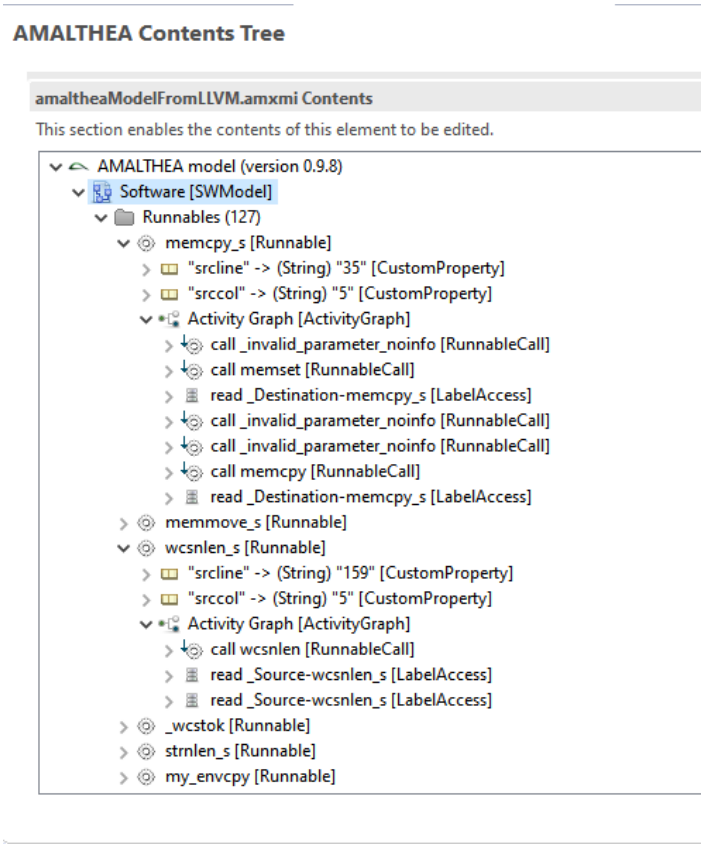


Abbildung 4.7: AMALTHEA-Softwaremodell

Er ist gruppiert nach Runnables (Funktionen), Labels und Typdefinitionen. Jedes Runnable hält dabei seinen zugehörigen ActivityGraph (vor APP4MC-Version 0.9.6 CallGraph genannt) mit den Zugriffen auf globale Variablen (`read` und `write`) und

andere Funktionen (`call`). Festzuhalten ist, dass die Runnable- und Label-Objekte im Softwaremodell nicht nach Dateien gruppiert sind. Der Quellcode wurde analysiert und dahingehend wie eine einzelne, große Klasse behandelt, die den gesamten Code enthält. Da jedoch in den sca-Datenstrukturen (Maps) zu den Funktionen und Variablen auch die zugehörigen Dateinamen gespeichert wurden, ist es durchaus möglich, diese zu gruppieren. Dies geschieht im darunterliegenden Komponentenmodell. Dort sind alle Quelldateien (Komponenten) gelistet und enthalten unter anderem die Listen von Runnable- und Label-Instanzen, die zu ihnen gehören (Reiter `Included SW Elements`, vgl. Abbildung 4.8).

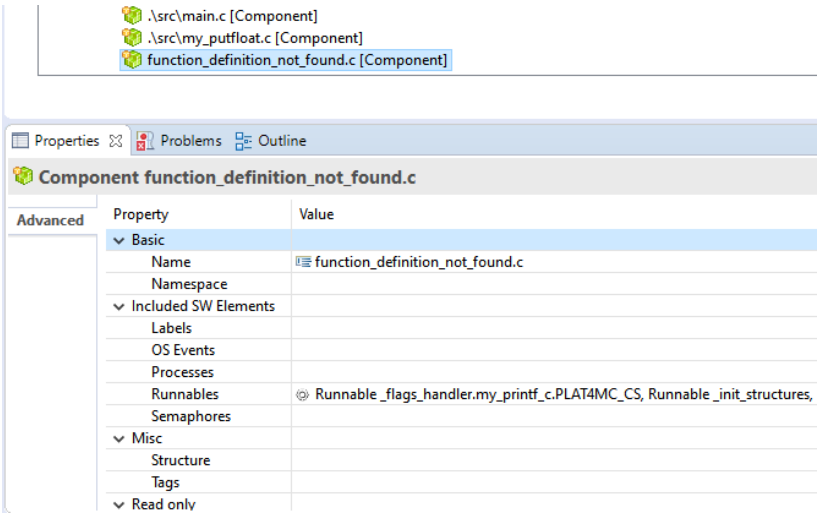


Abbildung 4.8: AMALTHEA-Komponentenmodell

In den Logdateien kann nachvollzogen werden, ob alle Quelldateien korrekt kompiliert und analysiert wurden. Oftmals kann es zu Fehlern bei den Includes kommen. Diese Komponenten werden dann übersprungen. Im Beispiel konnte `unistd.h` nicht lokalisiert werden (`fatal error: 'unistd.h' file not found`). Im Kontext der Arbeit wurde ein Windows-Rechner für den Praxistest verwendet. Da `unistd` ein POSIX-Header ist, wird dieser entsprechend unter Windows nicht gefunden. Es müsste nun also der Quellcode an diesem Punkt entsprechend angepasst werden, was jedoch im Rahmen der Forschung redundant wäre. Im Regelfall sollte ein kompilierender C-Quellcode auch in SCA2AMALTHEA ohne Probleme analysiert werden können, da clang ebenso wie andere Compiler auch dem gültigen Standard folgen.

Das nun vorhandene Modell kann als Basis für die Konfiguration verwendet werden. Viele Eigenschaften werden durch SCA2AMALTHEA nicht automatisch erkannt und gesetzt, da diese für die statische Codeanalyse unter anderem zu komplex sind oder aus semantischen Gründen aus dieser gar nicht hervorgehen. Diese müssen entsprechend manuell gesetzt werden. Es dient jedoch gerade in sehr großen Projekten als Hilfestellung, da damit ein großer Teil an Arbeit wegfällt.

5 Erweiterung

SCA2AMALTHEA liefert für die Modellgenerierung, wie bereits gezeigt, keine genauen Methodeninhalte, sondern lediglich Informationen zu aufgerufenen Untermethoden im Call-Stack sowie entsprechenden Zugriffen auf Parameter oder globale Variablen. Diese Arbeit soll nun das Plugin so erweitern, dass `if-else`-Zweige innerhalb von Methoden erkannt und auch im AMALTHEA-Modell entsprechend angelegt werden. Es zeigt sich bei näherer Betrachtung des C-Projektes, dass die grundlegenden Implementierungen für diese Funktionalitäten bereits vorgesehen sind, jedoch noch nicht ausgearbeitet wurden. Folglich werden sie zum Zeitpunkt dieser Arbeit auch noch nicht in der XML-Ausgabe berücksichtigt, wodurch auch im Java-Teil noch keine korrespondierenden Methoden zu finden sind. Im Rahmen der hier vorgenommenen Analyse wurde aufgrund der bestehenden Datenstrukturen im SCA-Projekt zunächst ein anderer Ansatz verfolgt, der jedoch nur mit hohem Aufwand bzw. sehr vielen Vorbehalten zum Ziel geführt hätte. Dieser wird der Vollständigkeit halber in Kapitel 6 noch detailliert ausgeführt und begründet, warum dieser nicht sinnvoll ist. Der Umstieg auf die folgende Methodik hat schließlich die vorhandene Codebasis bestmöglich genutzt und es mit moderatem Aufwand möglich gemacht, diese Funktionalität zu implementieren. Sie basiert auf der direkten Traversierung des abstrakten Syntaxbaumes, den clang aus dem C-Quellcode erzeugt und erlaubt es dadurch, jeweilige Codekonstrukte eindeutig zu identifizieren und komfortabel auszuwerten.

5.1 Darstellung in der SCA-IR

Um einen passenden Ansatz für die Darstellung von alternativen Ausführungspfaden in der Intermediate Representation des SCA-Projektes zu wählen, sollte zunächst betrachtet werden, wie das AMALTHEA-Modell diese darstellt. Dies ist von grundlegender Bedeutung, da sich jegliche Implementierungen nach dieser Struktur richten müssen. Zunächst existieren sogenannte `ActivityGraph`-Elemente (bis zu Version 0.9.7 `CallGraph` genannt, vgl. [11]), welche auch in Abbildung 4.7 zu sehen sind. Diese stellen den Ablauf einer Funktion, also deren Inhalt, dar. Die offizielle Dokumentation des APP4MC bietet auch eine Spezifikation des AMALTHEA-Metamodells, das heißt sie beschreibt dessen Aufbau. Relevant ist für diese Erweiterung nur der Teil des Softwaremodells, da dieses die übersetzten Codeeinheiten repräsentiert. Die folgende Abbildung zeigt dabei einen exemplarischen Ablauf mit integriertem Switch (alternativen Ausführungspfaden), welche auch den rekursiven Charakter der einzelnen Elemente gut darstellt. Auf diesen wird nachfolgend bei der Beschreibung der XML-Struktur noch zurückgegriffen, da er elementar für diese und alle weiteren Datenstrukturen ist.

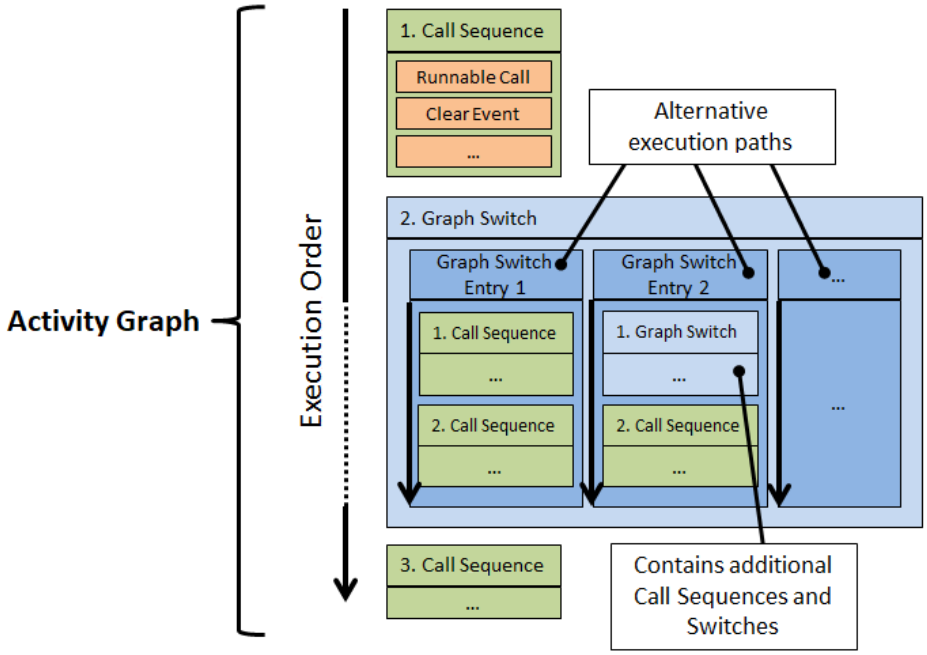


Abbildung 5.1: AMALTHEA Aktivitätsgraph [12]

Sucht man nun nach Informationen, wie diese Switches im Metamodell implementiert sind, so findet man zunächst die sogenannten `Mode Switch`-Instanzen.

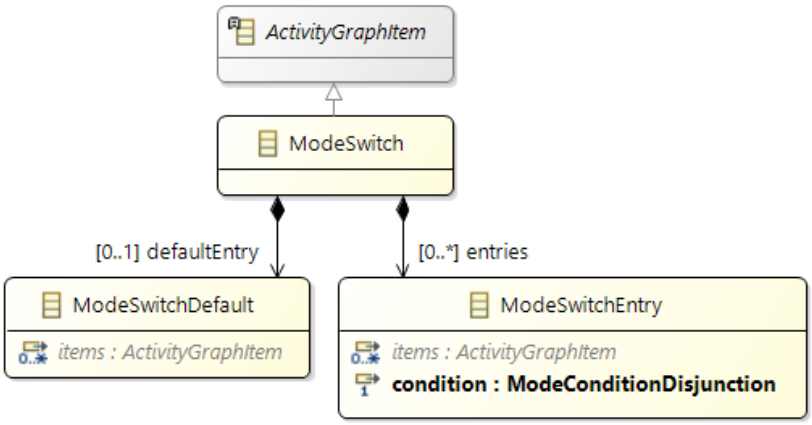


Abbildung 5.2: AMALTHEA Mode Switch [13]

Dabei handelt es sich, wie im Klassendiagramm ersichtlich, um Erben der Klasse `ActivityGraphItem`, wodurch diese auch den Aktvititätsgraphen hinzugefügt werden können. Sie enthalten eine Liste von (optionalen) `ModeSwitchEntry`-Objekten, die die einzelnen Ausführungspfade repräsentieren. Der `ModeSwitchDefault` ist für `switch-case`-Konstrukte interessant, wenn kein deklarierter Fall eintritt (`default`). (vgl. [14])

Nun stellt sich die Frage, ob diese `Mode Switch`es auch die passende Struktur sind, um generische `if-else`-Konstrukte im AMALTHEA-Modell darzustellen. In der Praxis werden Mode-Strukturen genutzt, um bestimmte Zustände (eines Fahrzeuges) zu hinterlegen. Etwa wäre es denkbar, dass die Stellung des Gaspedals mit einem solchen Mode abgefragt wird und das Motorsteuergerät dann abhängig von den konkreten Werten in den Kennfeldern unterschiedliche Drehmomentanforderungen weitergibt, die dann zu mehr oder weniger Kraftstoffeinspritzung führen. Auf diese Weise kann dann geregelt werden, wie stark das Fahrzeug beschleunigt. Ebenso könnte der Zustand des Bremspedals auf diese Weise geprüft werden, um bei starkem Bremsdruck zusätzliche Regelsysteme (z.B. das ABS) zu aktivieren. Derartige Abläufe können dann durch diese `Mode Switch`es modelliert werden.

Bei einer Übersetzung dieses Modells in zugehörigen Quellcode werden diese dann auch mit hoher Wahrscheinlichkeit in ein entsprechendes `if-else`-Konstrukt überführt. Eine generelle Umkehrung dieser Beziehung besteht folglich jedoch nicht. Zwar wären so gut wie alle Abläufe auch auf diese Weise repräsentierbar, sie wären jedoch rein formal gesehen nicht korrekt. Aus diesem Grund besteht auch noch eine Unterscheidung zu den `ProbabilitySwitch`es, die bestimme Pfade nur bei Eintreten einer gewissen Wahrscheinlichkeit ausführen. Wären `Mode Switch`es generische Konstrukte für alternative Ausführungspfade, so könnte man dies auch über diese darstellen.

Man würde an dieser Stelle folglich eine allgemeinere Struktur benötigen. Die in Abbildung 5.1 gezeigten `Graph Switch`es existieren als solche jedoch nicht im AMALTHEA-Metamodell. Da auch sonst keine adäquaten Alternativen zur Verfügung stehen, soll im Rahmen dieser Arbeit daher einfach das `Mode Switch` als Darstellungsvariante gewählt werden. Dies ist möglich, da jede Bedingung einer Abfrage auch als Zustand angesehen werden kann, der dann zu einem bestimmten Codepfad führt. Implizit besteht also zwischen den beiden Strukturen ein Isomorphismus, auch wenn die `Mode Switch`es einen konkreteren Zweck erfüllen sollen. Aufgrund dieser Abbildung ist es eben auch möglich, die generischen, alternativen Ausführungspfade auf diese Weise im AMALTHEA-Modell darzustellen. Lediglich bei der Darstellung der jeweiligen Bedingungen können hier unter Umständen Unterschiede eintreten, die im Rahmen dieser Arbeit jedoch nicht weiter erforscht und für die konkrete Erweiterung vernachlässigt wurden.

Sollte ab einem gewissen Zeitpunkt eine generische Datenstruktur bestehen, so stellt die Wahl der `Mode Switch`es keine Einschränkung dar, da die zugrundeliegende XML-Struktur sehr dynamisch einsetzbar ist und etwaige Modellelemente im Java-Teil des Plugins jederzeit ohne Aufwand ausgetauscht werden können. Die abstrakte Repräsentation der Elemente im Aktivitätsgraphen hilft dabei, da jeweilige Kinder bei allen Elementen auf die gleiche Weise hinzugefügt werden können.

Zu sehen ist in [14] auch, dass die jeweiligen Elemente der Ausführungspfade direkt als Kinder (`items`) zu den `ModeSwitchEntry`-Objekten hinzugefügt werden können. Entsprechend sind dies die für die Umsetzung dieser Funktionalität relevanten Elemente, welche aus einer passenden SCAIR erstellt werden müssen. Um sich ein Bild davon zu machen, wie die bisherigen Funktionsinhalte im XML-Format dargestellt werden, sollte eine beispielhafte, durch SCA erzeugte CallTree.xml-Datei betrachtet werden:

```
2 <scair:Project xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:scair="http://org.eclipse.app4mc.scair.
    core.model" location="D:\Downloads\org.eclipse.app4mc.
    platform-0.9.8-20200430-144456-win32.win32.x86_64\
    workspace\Test">
3  <containers>
4   <container name=".\lib\my\src\env\my_setenv.c">
5     <functions>
6       <function srcline="4" srccol="1" File="" Package="">
7         <name>get_new_env_line.my_setenv_c.APP4MC_CS</name>
8         <stmtseq xsi:type="scair:FunctionCall" calls="
    malloc?type=Function" srcline = "8" srccol="18"/>
9         <stmtseq xsi:type="scair:FunctionCall" calls="
    my_strlen?type=Function" srcline = "8" srccol="41"/>
10        <stmtseq xsi:type="scair:LabelAccess" label="
    to_change-get_new_env_line.my_setenv_c.APP4MC_CS?type=
    Label" Access="Read" srcline="8" srccol="51"/>
11        <stmtseq xsi:type="scair:FunctionCall" calls="
    my_strlen?type=Function" srcline = "9" srccol="8"/>
12        <stmtseq xsi:type="scair:FunctionCall" calls="
    my_exit?type=Function" srcline = "11" srccol="5"/>
13      </function>
14      <function srcline="17" srccol="1" File="" Package="">
15        <name>add_var.my_setenv_c.APP4MC_CS</name>
16        ...
```

Listing 5.1: Beispielhafte CallTree.xml

Ersichtlich ist, dass die einzelnen Funktionen innerhalb eines Containers abgelegt werden. Dieser repräsentiert dabei die analysierte Datei. Eine Funktion erhält verschiedene Attribute, unter anderem eine Liste von `stmtseq`-Objekten. Diese sind in chronologischer Reihenfolge angeordnet und beschreiben die einzelnen Funktions- oder Variablenaufrufe in einer Funktion (siehe `xsi:type`, welcher eine korrespondierende Klasse im Java-Teil erhält).

Für die Umsetzung der neuen Funktionalität müssen die bestehenden Einträge nun mittels weiterer Tags gruppiert werden, welche die zuvor angesprochene Rekursivität berücksichtigen. Die Vermischung der Liste von `stmtseq`-Einträgen mit den Mode-Switch-Elementen erfordert daher die Nutzung des Kompositum-Entwurfsmusters. Um sich dies klar zu machen, muss man diese XML-Struktur als Baum betrachten. Das heißt, dass die Mode-Switches Container darstellen, die (indirekt über dazwischenliegende `modeswitchentry`-Einträge) abschließende `stmtseq`-Objekte (Blätter) oder weitere Container enthalten können. Hierfür ist also eine entsprechende Oberklasse notwendig. `FunctionCall` und `LabelAccess` bilden dann entsprechend die Blätter. Die folgende Abbildung zeigt exemplarisch einen solchen Aufbau, wobei die direkten Kinder eines Knotens von links nach rechts evaluiert werden. Dies entspricht einer Tiefensuche mit Infix-Ordnung.

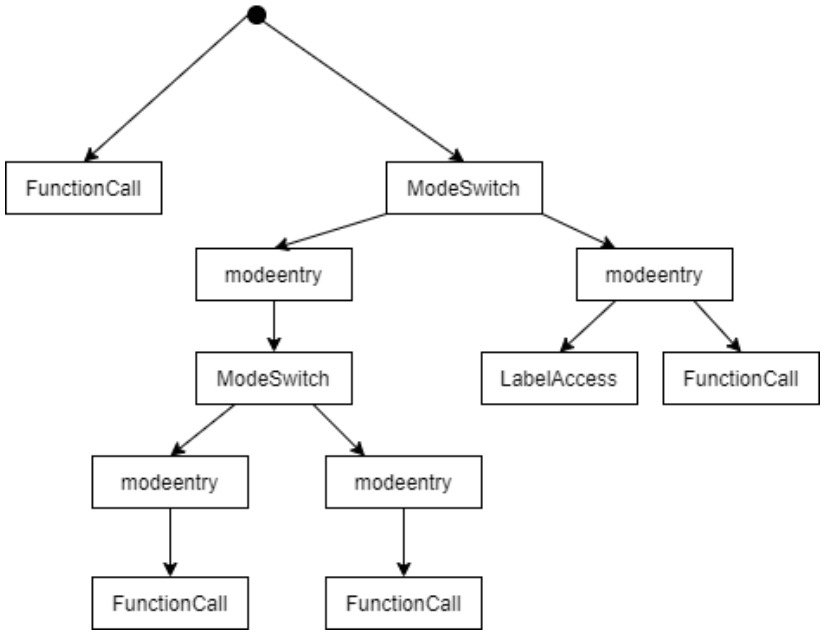


Abbildung 5.3: Exemplarische Repräsentation der XML-Struktur in Form eines Baums

Betrachtet man dann die Java-Implementierung für das Einlesen der XML-Datei im Paket `org.eclipse.app4mc.sca2amalthea.ir.scair`, so fällt auf, dass die Klasse `Function`, die eine Funktion repräsentiert (vgl. Zeile 7 im Ausschnitt 5.1), folgende Definition aufweist:

```
1 public interface Function extends IdentifiableElement {
2     // ...
3     EList<StmtCall> getStmtseq();
4     // ...
```

Listing 5.2: Function.java

Diese hält eine Liste von `StmtCall`-Objekten, welche unterschiedliche Arten von Aufrufen darstellen und über diese Klasse abstrahiert werden. Entsprechend könnte hier mit dem Kompositum direkt angesetzt werden, indem man ein `ModeSwitch` ebenfalls von dieser erben lässt und rekursiv weitere `StmtCall`-Instanzen in den jeweiligen `ModeSwitchEntry`-Instanzen speichert. Somit könnte man das XML-Format also wie folgt realisieren:

```
1 <function srcline="4" srccol="1" File="" Package="">
2     <name>get_new_env_line.my_setenv_c.APP4MC_CS</name>
3     <stmtseq xsi:type="scair:ModeSwitch">
4         <modeswitchentry>
5             <stmtseq xsi:type="scair:FunctionCall" calls="
    malloc?type=Function" srcline = "8" srccol="18"/>
6         </modeswitchentry>
7         <modeswitchentry>
8             <stmtseq xsi:type="scair:FunctionCall" calls="
    my_strlen?type=Function" srcline = "8" srccol="41"/>
9         </modeswitchentry>
10     </stmtseq>
11     <stmtseq xsi:type="scair:FunctionCall" calls="my_exit?
    type=Function" srcline = "11" srccol="5"/>
12 </function>
```

Listing 5.3: Beispielhafte CallTree.xml mit Mode Switches

Die Nutzung des `stmtseq`-Tags für Mode-Switches ist notwendig, da für die Nutzung des Kompositums eine Liste gleichartiger Tagnamen in der XML-Struktur erforderlich ist. Alternative Ausführungspfade sollen hierbei durch den Tag `modeswitchentry` deklariert werden. Die einzelnen `modeswitchentry`-Elemente könnten dabei noch mit weiteren Attributen versehen werden. Denkbar wäre beispielsweise die zugehörige Auswertungsbedingung. Da hierfür jedoch eine Vielzahl weiterer Anpassungen für die Konstruktion des AMALTHEA-Modells notwendig wäre, soll dies im Rahmen dieser Arbeit vernachlässigt werden.

Damit man die XML-Struktur auf diese Weise bauen kann, besteht logischerweise die Notwendigkeit auch die zu traversierende Struktur als Baum zu repräsentieren. Ideal ist hierfür bereits der abstrakte Syntaxbaum, den clang generiert, da dieser alle möglichen Arten von Kontrollstrukturen beinhaltet und das SCA-Projekt bereits basierend auf diesem seine Informationen generiert.

5.2 C++-Teil

Als Grundlage für die Erweiterungen müssen zunächst neue Modellklassen eingeführt werden. Gemäß der zuvor beschriebenen Struktur werden Klassen für einen `ModeSwitch` und `ModeSwitchEntry` benötigt. Zusätzlich müssen für das Kompositum Abstraktionsklassen definiert werden, die Container-Objekte von enthaltenen Objekten (Blättern) unterscheiden.
Entsprechend sei eine Klasse `CallGraphItemContainer` definiert:

```
1  #ifndef CALLGRAPHITEMCONTAINER_H
2  #define CALLGRAPHITEMCONTAINER_H
3
4  #include "CallGraphItem.h"
5  #include <list>
6
7  class CallGraphItemContainer {
8  private:
9    std::list<CallGraphItem*> children;
10 public:
11   void addChild(CallGraphItem *callGraphItem);
12   void removeChild(CallGraphItem *callGraphItem);
13   std::list<CallGraphItem*> getChildren();
14 };
15 #endif
```

Listing 5.4: CallGraphItemContainer.h

```
1  #include "CallGraphItemContainer.h"
2
3  void CallGraphItemContainer::addChild(CallGraphItem *
       callGraphItem) {
4          this->children.push_back(callGraphItem);
5  }
6
7  void CallGraphItemContainer::removeChild(CallGraphItem *
       callGraphItem) {
8          this->children.remove(callGraphItem);
9  }
10
11 std::list<CallGraphItem*> CallGraphItemContainer::
       getChildren() {
12         return this->children;
13 }
```

Listing 5.5: CallGraphItemContainer.cpp

Diese hält eine Liste von `CallGraphItems` und bietet Methoden an, um diese zu verwalten. Der zugehörige Header sieht wie folgt aus:

```
1  #ifndef CALLGRAPHITEM_H
2  #define CALLGRAPHITEM_H
3
4  #include "llvm/Support/Casting.h"
5
6  class CallGraphItem {
7  public:
8    enum CallGraphItemKind {
9      K_CallEntry,
10     K_GlobalVariableAccess,
11     K_ModeSwitch
12   };
13 private:
14   unsigned line;
15   unsigned col;
16   unsigned basicBlockID;
17   const CallGraphItemKind Kind;
18 public:
19     CallGraphItem(unsigned line, unsigned col, unsigned
     basicBlockID, CallGraphItemKind K) : line(line), col(col
     ), basicBlockID(basicBlockID), Kind(K) {}
20
21   unsigned getLine();
22   unsigned getColumn();
23   unsigned getBasicBlockID();
24   CallGraphItemKind getKind() const { return Kind; }
25 };
26 #endif
```

Listing 5.6: CallGraphItem.h

Neben den Modellinformationen befinden sich noch weitere Konstrukte in diesem Header. Auffällig sind dabei u.a. die `CallGraphItemKind`-Enumeration und zugehörige Attribute bzw. Zugriffsmethoden. Diese sind später für die dynamische Typauflösung und das somit notwendige Casten zwischen jeweiligen Ober- und Unterklassen relevant. Da clang bzw. LLVM intern statt der in C++ integrierten Runtime Type Information (*RTTI*) eine proprietäre Implementierung nutzt, müssen betroffene Klassen noch mit weiterer Funktionalität dekoriert werden. Es wird folglich auch nicht das in C++ integrierte `dynamic_cast` verwendet. Dadurch muss auch keine Methode mehr als `virtual` gekennzeichnet werden, da dies Voraussetzung für dessen Funktionalität ist: “If it was possible to use dynamic_cast to downcast non-polymorphic types, compliers would have to store run-time type information in every class type.” ([15])

Genau diesen Ansatz verfolgt jedoch die LLVM-RTTI und somit wurde dies entsprechend in der `CallGraphItem`-Klasse gemäß der offiziellen Dokumentation[1] umgesetzt. Folglich beinhaltet diese eine zusätzliche Enumeration, die die Art (d.h. die konkrete Unterklasse) dieses Items angibt, um später eine Typauflösung zu erlauben, ohne dass diese polymorph ist.

Die Implementierung ist dann simpel, da sehr viel Initialisierungslogik bereits in den Header integriert wurde.

```
1 #include "CallGraphItem.h"
2
3 unsigned CallGraphItem::getLine(){
4   return this->line;
5 }
6 unsigned CallGraphItem::getColumn(){
7   return this->col;
8 }
9
10 unsigned CallGraphItem::getBasicBlockID(){
11   return this->basicBlockID;
12 }
```

Listing 5.7: CallGraphItem.cpp

Die Klasse enthält genau die Metainformationen, die in jeder der `stmtseq`-Klassen (d.h. `CallEntry` und `GlobalVariableAccess`) entsprechend auftreten. Da diese dann durch die `CallGraphItem`-Klasse abstrahiert werden, erben sie die Attribute und Methoden und müssen diese nicht immer neu definieren, was zu Duplizierung führt. Auch die `ModeSwitches` werden dann entsprechend als `CallGraphItem` ausgewiesen, da sie in Containern als Elemente auftreten. Somit ergeben sich diese drei Arten von Unterklassen auch für die RTTI, wodurch man sie auch in der `CallGraphItemKind`-Definition vorfindet.

Die Klasse `FunctionAndCallees` ist dann für das Halten der `CallGraph`-Daten verantwortlich, da diese eine Funktion mit ihrem Ablauf repräsentiert. Betrachtet man die haltende Baumstruktur, so erkennt man, dass die `ModeSwitchEntry`-Instanzen und die Funktionsdefinitionen `FunctionAndCallees` selbst die `CallGraphItemContainer` -Objekte sind bzw. einen Zeiger auf ein solches halten. Initial ergibt sich somit für den ersten Geltungsbereich in der Funktion eine Liste von `CallGraphItems`, die direkt Blätter (`CallEntry` oder `GlobalVariableAccess`) oder weiter verschachtelte Elemente (`ModeSwitchEntry`-Elemente in einem `ModeSwitch`) sein können, da diese weitere `CallGraphItems` im jeweiligen Ausführungspfad (d.h. dessen Geltungsbereich) enthalten. Diese sind also auch Container.

[1] https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html

Da `FunctionAndCallees` eine allgemeinere Datenstruktur ist, die auch noch weitere Informationen sammelt, soll diese nicht von `CallGraphItemContainer` erben, sondern lediglich einen Zeiger auf ein solches Objekt speichern. Somit ergeben sich folgende Änderungen:

```
1 /* Data structure to hold the informations concerning a
      function and its callees*/
2 class FunctionAndCallees {
3 private:
4   CallGraphItemContainer *currentContainer;
5   // ...
6 public:
7   void addCallee(CallEntry *callEntry);
8   void addGlobalVariableAccess(GlobalVariableAccess *
      globalVariableAccess);
9   CallGraphItemContainer* getCurrentContainer();
10  void setCurrentContainer(CallGraphItemContainer *
      callGraphItemContainer);
11  // ...
12 };
13 #endif /* FUNCTION_AND_CALLEES_H */
```

Listing 5.8: FunctionAndCallees.h (gekürzt)

Zusätzlich wurden die Methoden `addGlobalVariableAccesses` und `addCallees` umbenannt, da jeweils nur ein Element hinzugefügt wird und somit der Singular für die Nomenklatur zu wählen ist.

```
1 #include "FunctionAndCallees.h"
2
3 FunctionAndCallees::FunctionAndCallees() {
4   this->currentContainer = new CallGraphItemContainer();
5 }
6
7 CallGraphItemContainer *FunctionAndCallees::
      getCurrentContainer() {
8   return currentContainer;
9 }
10
11 void FunctionAndCallees::setCurrentContainer(
12   CallGraphItemContainer *callGraphItemContainer) {
13   this->currentContainer = callGraphItemContainer;
14 }
15 // ...
```

Listing 5.9: FunctionAndCallees.cpp (gekürzt)

Der Ansatz, den `CallGraphItemContainer` in der Klasse `FunctionAndCallees` zu speichern und dennoch die alten Listen von Funktionsaufrufen und Variablenzugriffen zusätzlich beizubehalten, mag redundant erscheinen. Dieser erweist sich jedoch als sinnvoll. Zum einen können bestehende Methoden, die zusätzliche Informationen akquierieren und verarbeiten, ohne Anpassungen weiter genutzt werden. Etwa wird beim Sammeln der Funktionsaufrufe geprüft, welche der aufgerufenen Funktionen tatsächlich existieren und welche nicht definiert wurden. Auf diese Weise können für die folgende Repräsentation in der XML-Datei passende Vorkehrungen getroffen werden, um diesen invaliden Zustand darzustellen. Der Ansatz ermöglicht zusätzich eine hohe Kompatibilität mit der bestehenden Basis des SCAIR-Modells, falls man dieses zukünftig so verwenden/erweitern möchte, wie Bosch dies ursprünglich vorgesehen hat. Dies wird im Unterkapitel 5.3 noch genauer erläutert.

Nun müssen noch die neuen Strukturen für die Mode Switches definiert werden.

```
1  #ifndef MODESWITCH_H
2  #define MODESWITCH_H
3
4  #include <list>
5  #include "ModeSwitchEntry.h"
6
7  class ModeSwitch : public CallGraphItem {
8  private:
9      std::list<ModeSwitchEntry*> modeEntries;
10 public:
11     ModeSwitch(unsigned line, unsigned col, unsigned
       basicBlockID);
12     std::list<ModeSwitchEntry*>* getModeEntries();
13     void addModeSwitchEntry(ModeSwitchEntry *
       modeSwitchEntry);
14
15     static bool classof(const CallGraphItem *S) {
16         return S->getKind() == K_ModeSwitch;
17     }
18 };
19 #endif
```

Listing 5.10: ModeSwitch.h

Die `ModeSwitch`-Klasse erbt, wie zuvor erläutert, von `CallGraphItem`, speichert eine Liste von `ModeSwitchEntry`-Objekten und bietet Methoden für dessen Verwaltung an. Die Deklaration der `classof`-Methode ist notwendig für die Implementierung der LLVM-RTTI, damit der Typ der Klasse zur Laufzeit bestimmt/geprüft werden kann. Hierfür wird der Typ des übergebenen, allgemeinen `CallGraphItem`-Objekts mit dem der aktuellen Klasse verglichen.

```
#include "ModeSwitch.h"

ModeSwitch::ModeSwitch(unsigned line, unsigned col,
    unsigned basicBlockID) : CallGraphItem(line, col,
    basicBlockID, K_ModeSwitch) { }

std::list<ModeSwitchEntry *> *ModeSwitch::getModeEntries()
    {
     return &this->modeEntries;
}

void ModeSwitch::addModeSwitchEntry(ModeSwitchEntry *
    modeSwitchEntry) {
     this->modeEntries.push_back(modeSwitchEntry);
}
```

Listing 5.11: ModeSwitch.cpp

Wichtig ist, den Konstruktor der Oberklasse entsprechend mit den übergebenen Parametern aufzurufen, um die notwendigen Attribute auch zu initialisieren.

Die `ModeSwitchEntry`-Klasse erbt von `CallGraphItemContainer` und enthält damit automatisch alle notwendigen Attribute und Methoden. Da diese lediglich die Aufgabe eines Platzhalters ohne weitere Logik übernimmt, ist die Implementierung leer:

```
#ifndef MODESWITCHENTRY_H
#define MODESWITCHENTRY_H

#include "CallGraphItemContainer.h"
#include <list>

class ModeSwitchEntry : public CallGraphItemContainer {
};

#endif
```

Listing 5.12: ModeSwitchEntry.h

```
#include "ModeSwitchEntry.h"
```

Listing 5.13: ModeSwitchEntry.cpp

Würde man weitere Informationen, wie die zugehörige Bedingung des Pfades, mit aufnehmen wollen, so müsste die zugehörige Logik an dieser Stelle eingefügt werden.

Nun muss noch die Vererbungshierarchie des `CallGraphItems` in die Modellklassen `CallEntry` und `GlobalVariableAccess` eingearbeitet werden. Diese folgen der exakt selben Logik, um die Unterstützung der LLVM-RTTI zu gewährleisten. Ebenso muss die Oberklasse wieder mit den übergebenen Parametern aufgerufen werden. Hieraus folgt auch, dass die bisher hinterlegten Attribute `line`, `col` und `basicBlockID` nicht mehr notwendig sind, da diese nun aus `CallGraphItem` geerbt werden und man deren Initialisierung an diese Klasse delegiert. Somit können diese auch gelöscht werden. Der restliche Code bleibt von den Änderungen unberührt.

```
21 class CallEntry : public CallGraphItem {
22 private:
23   FunctionEntry* functionID;
24 public:
25   CallEntry(FunctionEntry* functionID, unsigned callLine,
      unsigned callColumn, unsigned basicBlockID);
26   FunctionEntry* getFunctionID();
27   static bool classof(const CallGraphItem *S) {
28     return S->getKind() == K_CallEntry;
29   }
30 };
```

Listing 5.14: CallEntry.h (gekürzt)

```
22 class GlobalVariableAccess : public CallGraphItem
23 {
24 private:
25   GlobalVariable* globalVariable;
26   std::string accessType;
27
28 public:
29   GlobalVariableAccess(GlobalVariable* globalVariable, std
      ::string accessType, unsigned line, unsigned col,
      unsigned basicBlockID);
30
31   GlobalVariable* getGlobalVariable();
32   std::string getAccessType();
33   void setGlobalVariable(GlobalVariable* gA);
34
35   static bool classof(const CallGraphItem *S) {
36     return S->getKind() == K_GlobalVariableAccess;
37   }
38 };
```

Listing 5.15: GlobalVariableAccess.h (gekürzt)

```
1 // ...
2 GlobalVariableAccess::GlobalVariableAccess(GlobalVariable*
      globalVariable, std::string accessType, unsigned line,
      unsigned col, unsigned basicBlockID)
3   : CallGraphItem(line, col, basicBlockID,
      K_GlobalVariableAccess) {
4   this->globalVariable = globalVariable;
5   this->accessType = accessType;
6 }
7 // ...
```

Listing 5.16: CallEntry.cpp (gekürzt)

```
1 // ...
2 CallEntry::CallEntry(FunctionEntry* functionID, unsigned
      callLine, unsigned callColumn, unsigned basicBlockID)
3   : CallGraphItem(callLine, callColumn, basicBlockID,
      K_CallEntry) {
4   this->functionID = functionID;
5 }
6 // ...
```

Listing 5.17: CallEntry.cpp (gekürzt)

Damit ist die Grundlage für die entsprechende Struktur gelegt, ohne dass bestehende Methoden überarbeitet werden müssen. Nun müssen diese Daten noch während der Traversierung des abstrakten Syntaxbaumes korrekt erstellt werden. clang verwendet für die Abarbeitung des Syntaxbaumes das Besucher-Entwurfsmuster. Im SCA-Projekt übernimmt die Klasse `FindInformationClassVisitor` diese Aufgabe. Sie ist in der Klasse `TraverseASTClassAction` definiert. In der Implementierung kann dann entsprechend eingegriffen werden. Die Methode `FindInformationClassVisitor ::VisitFunctionDecl` wird beim Antreffen einer Funktion im AST aufgerufen. Innerhalb dieser wird dann die zugehörige `FunctionAndCallees`-Instanz gebaut. In Zeile 369 wird ein `CallEntry`-Objekt erstellt. Dieses muss nun zusätzlich auch im zugehörigen `CallGraphItemContainer` gespeichert werden, wodurch sich folgender Codeausschnitt ergibt:

```
369 CallEntry *callEntry = new CallEntry(functionDeclaration,
      line, col, 0);
370 currentAnalyzedFunction->getCurrentContainer()->addChild(
      callEntry);
371 currentAnalyzedFunction->addCallee(callEntry);
```

Listing 5.18: TraverseClassAction.cpp (gekürzt)

Genau analog muss diese Logik in der Methode `addNewGlobalAccess` (Zeile 1262) hinzugefügt werden:

```
1262 currentAnalyzedFunction->getCurrentContainer()->addChild(
       globalVariableAccess);
1263 currentAnalyzedFunction->addGlobalVariableAccess(
       globalVariableAccess);
```

Listing 5.19: TraverseClassAction.cpp (gekürzt)

Das Parsen von Funktionsaufrufen wird auch durch eine Funktion `parseFunctionCall` übernommen. Auch hier muss eine Erweiterung in Zeile 1378 stattfinden:

```
1378 callEntry = new CallEntry(functionDeclaration, line, col,
       blockID);
1379 currentAnalyzedFunction->getCurrentContainer()->addChild(
       callEntry);
1380 currentAnalyzedFunction->addCallee(callEntry);
```

Listing 5.20: TraverseClassAction.cpp (gekürzt)

Der aktuelle Container, zu dem all diese Funktionen ihre erstellten Elemente hinzufügen, muss jedoch auch irgendwo gesetzt werden. Dieser kann sich dabei in einer beliebigen Schachtelungsstufe befinden, wenn man einzelne Ausführungspfade traversiert. Anfangs ist es der initiale Container des `FunctionAndCallees`-Objektes (siehe Zeile 3 in Codeausschnitt 5.9), sodass alle gefundenen Objekte diesem direkt als Kinder hinzugefügt werden. Dieser muss erst dann neu gesetzt werden, wenn ein Mode Switch Entry vorliegt. Entsprechend ist dies bei der Behandlung von alternativen Ausführungspfaden der Fall. Für diese muss eine neue Methode `parseIfStmt` eingeführt werden, die dies erledigt und sich nahtlos in die bestehende Struktur einfügt. Auch diese wird in der Klasse `TraverseClassAction` hinzugefügt:

```
911 void FindInformationClassVisitor::parseIfStmt(Stmt *stmt) {
912     auto ifStmt = llvm::cast<IfStmt>(stmt);
913     Stmt *condition = ifStmt->getCond();
914     Stmt *ifPart = ifStmt->getThen();
915     Stmt *elsePart = ifStmt->getElse();
916     CallGraphItemContainer *currentContainer =
      currentAnalyzedFunction->getCurrentContainer();
917
918     selectAndExecuteStatement(condition);
919
920     auto modeSwitch = new ModeSwitch(getLineNumberStmt(
      ifStmt), getColumnNumberStmt(ifStmt), getBlockID(ifStmt)
      );
```

```
921     const auto ifModeSwitchEntry = new ModeSwitchEntry();
922     modeSwitch->addModeSwitchEntry(ifModeSwitchEntry);
923     currentAnalyzedFunction->getCurrentContainer()->
      addChild(modeSwitch);
924     currentAnalyzedFunction->setCurrentContainer(
      ifModeSwitchEntry);
925     selectAndExecuteStatement(ifPart);
926
927     if (elsePart) {
928       const auto elseModeSwitchEntry = new ModeSwitchEntry
      ();
929       modeSwitch->addModeSwitchEntry(elseModeSwitchEntry);
930       currentAnalyzedFunction->setCurrentContainer(
      elseModeSwitchEntry);
931       selectAndExecuteStatement(elsePart);
932     }
933
934     // Reset the container to the one before the recursion
935     currentAnalyzedFunction->setCurrentContainer(
      currentContainer);
936 }
```

Listing 5.21: parseIfStmt

Hierzu wird zunächst das aktuelle `Stmt`-Objekt mittels `llvm:cast` (LLVM-RTTI) in ein `IfStmt`-Objekt (If-Abfrage) gecastet und dann die zugehörigen Daten (Bedingung, `then`-Ausdruck und `else`-Ausdruck) extrahiert. Da die Bedingungsanweisung noch vor der Abfrage ausgewertet wird und diese anschließend nur noch deren Ergebnis prüft, gehört der Ausdruck selbst noch zum aktuellen Geltungsbereich. Aus diesem Grund wird dieser mit `selectAndExecuteStatement(condition)` noch zuvor behandelt und dem aktuellen Container hinzugefügt. Ebenso wird dann ein neu erzeugtes `ModeSwitch` -Objekt an diesen angehängt. Für den `then`-Teil wird dann ein neuer Mode Switch Entry instanziiert und dem Mode Switch hinzugefügt. Der aktuelle Container wird dann auf den `ifModeSwitchEntry` gesetzt und somit alle weiteren Elemente dann rekursiv diesem hinzugefügt, da diese immer vom aktuellen Container ausgehen, welcher dann dieser Ausführungspfad ist. Das weitere, rekursive Ablaufen des AST übernimmt dabei wieder die Methode `selectAndExecuteStatement`, auf die gleich noch näher eingegangen wird. Dasselbe passiert dann für den `else`-Teil, falls dieser existiert. Sobald dann alle geschachtelten Elemente abgearbeitet wurden, wird der Container wieder auf den vorherigen zurückgesetzt. Dieser wurde in Zeile 916 zwischengespeichert. Alle auf den Switch folgenden Elemente kommen somit wieder in ihren korrekten Geltungsbereich.

Damit bleibt noch zu klären, wo diese Methode aufgerufen werden muss und was `selectAndExecuteStatement` im Kontext für eine Aufgabe hat. Dazu sollte zunächst betrachtet werden, wie die Traversierung einer Funktion genau gestartet wird.

Da der Eintrittspunkt (Body) der Funktion auch durch ein `clang::Stmt` repräsentiert wird, kann ausgehend von diesem begonnen werden. Die Besuchsfunktion für Funktionsdeklarationen erhält als Parameter ein `FunctionDecl`-Objekt, das dieses initiale `Stmt` über die Methode `getBody` zur Verfügung stellt, falls es sich um eine Funktionsdefinition (d.h. nicht nur einen Aufruf) handelt. Da dieses `Stmt`-Objekt keine direkten Inhalte, sondern nur den Einstiegspunkt bereitstellt, müssen die einzelnen Komponenten (Kontrollstrukturen, Aufrufe, Anweisungen, ...) der Funktion erst über dieses abgerufen werden. Diese sind entsprechend als dessen Kindelemente hinterlegt. Eine Methode `parseStmtGeneric` erledigt genau diese Aufgabe. Sie nimmt ein beliebiges `Stmt` entgegen und behandelt dieses generisch, d.h. ohne darauf zu achten, um welche Art von Ausdruck es sich dabei handelt und ohne spezifische Daten zu diesen zu sammeln. Hierfür iteriert diese lediglich über alle vorhandenen Kinder und ruft für jedes die Methode `selectAndExecuteStatement` auf:

```
635 void FindInformationClassVisitor::parseStatementGeneric(
      Stmt *stmt) {
636   //stmt->dumpColor();
637   for (Stmt::child_iterator I = stmt->child_begin(), E =
      stmt->child_end();
638        I != E; ++I) {
639     Stmt *child = *I;
640     selectAndExecuteStatement(child);
641   }
642 }
```

Listing 5.22: parseStatementGeneric

Diese wiederrum muss sich dann um die spezifische Behandlung der `Stmt`-Instanzen kümmern, d.h. diese nicht mehr nur generisch behandeln, da dort dann die für die detaillierten Funktionsabläufe relevanten Daten hinterlegt sind. Entsprechend finden innerhalb dieser nur Typprüfungen und Aufrufe von spezifischen Methoden statt:

```
661 void FindInformationClassVisitor::selectAndExecuteStatement
      (Stmt *child) {
662   try {
663
664     if (child) {
665       MemberExpr *memberExpr = nullptr;
666       switch (child->getStmtClass()) {
667       case Stmt::BinaryOperatorClass:
668         parseBinaryOperator(child);
669         break;
670       case Stmt::UnaryOperatorClass:
671         parseUnaryOperator(child);
```

```
672           break;
673         case Stmt::ReturnStmtClass:
674         case Stmt::DeclStmtClass:
675         case Stmt::CompoundStmtClass:
676         case Stmt::ForStmtClass:
677         case Stmt::SwitchStmtClass:
678         case Stmt::WhileStmtClass:
679         case Stmt::DoStmtClass:
680         case Stmt::CaseStmtClass:
681         case Stmt::DefaultStmtClass:
682         case Stmt::ParenExprClass:
683         case Stmt::IfStmtClass:
684           parseStatementGeneric(child);
685           break;
686         case Stmt::CompoundAssignOperatorClass:
687           parseBinaryOperator(child);
688           break;
689         case Stmt::CallExprClass:
690           parseCallExprOperator(child);
691           break;
692         case Stmt::ImplicitCastExprClass:
693           parseImplicitCastExpression(child);
694           break;
695         case Stmt::MemberExprClass:
696           memberExpr = llvm::cast<MemberExpr>(child);
697           memberExprName = memberExpr
698                              ->getMemberNameInfo().getName().
      getAsString();
699           parseStatementGeneric(child);
700           memberExprName = "";
701           break;
702         default:
703           parseStatementGeneric(child);
704           break;
705         }
706       }
707     } catch (...) {
708       cerr << "An error happened in selectAndExecute  " <<
      child->
709           getStmtClassName() << std::endl;
710       //child->dumpColor();
711     }
712 }
```

Listing 5.23: selectAndExecuteStatement

Alle relevanten Ausdrücke können hier passend erkannt und behandelt werden. Zum Stand dieser Arbeit ist die entsprechende Implementierung noch nicht abgeschlossen, sodass eine Reihe von Ausdrücken nur generisch behandelt wird und durch die entstehende Rekursion nur die Kindelemente mit in die Ausgabe aufgenommen werden, die dort definiert sind. Aus diesem Grund enthält das bisherige AMALTHEA-Modell nur alle Funktionsaufrufe und Variablenzugriffe, da lediglich diese hier bereits intern behandelt und weitere Informationen zu Geltungsbereichen oder Schachtelungen ignoriert werden. Auch die `IfStmt`-Instanzen resultieren in einem `parseStatementGeneric`-Aufruf, sodass noch keine genaue Implementierungslogik zu Mode Switches vorliegt (siehe Zeile 683). Genau an dieser Stelle muss folglich angesetzt werden, die zuvor definierte `parseIfStmt`-Methode aufzurufen. Es ergibt sich dann also folgende Änderung:

```
661 void FindInformationClassVisitor::selectAndExecuteStatement
      (Stmt *child) {
662   try {
663     if (child) {
664       // ...
665       case Stmt::DoStmtClass:
666       case Stmt::CaseStmtClass:
667       case Stmt::DefaultStmtClass:
668       case Stmt::ParenExprClass:
669         parseStatementGeneric(child);
670         break;
671       case Stmt::IfStmtClass:
672         parseIfStmt(child);
673         break;
674       // ...
675       }
676     }
677   } catch (...) {
678     // ...
679   }
680 }
```

Listing 5.24: TraverseClassAction.cpp (gekürzt)

Abschließend muss noch die Ausgabe des Plugins überarbeitet werden, damit die eingeführte Struktur auch passend in die XML-Datei geschrieben wird. Die vorgenommenen Änderungen betreffen lediglich die Inhalte der Funktionstags. Der Rest bleibt unberührt. Aus diesem Grund muss auch nur die dafür zuständige Funktion `printFunctionContentInOrderAdvanced` angepasst werden. Diese befindet sich in der Klasse `Output`. Die initiale Implementierung iteriert dabei über alle Variablenzugriffe und Funktionsaufrufe und ordnet diese dabei gemäß ihrer Zeilennummern chronologisch an, um sie dann sequentiell in die Funktion zu schreiben. Da nun die jeweiligen Objekte bereits strukturiert und geordnet in der `CallGraph`-Struktur vorlie-

gen, ist dies nicht mehr notwendig und der bestehende Ansatz kann komplett verworfen werden. Die neue Implementierung ergibt sich dann wie folgt:

```
261 void printFunctionContentInOrderAdvanced(
262   CallGraphItemContainer *callGraphItemContainer, ofstream
      *outputFile,
263   ofstream *errorsFile, string &filename) {
264
265   list<CallGraphItem*> callGraphItems =
      callGraphItemContainer->getChildren();
266   for (list<CallGraphItem*>::iterator callGraphItemIterator
      = callGraphItems.
267            begin(); callGraphItemIterator != callGraphItems
      .end(); ++
268        callGraphItemIterator) {
269     printCallGraphItem(*(callGraphItemIterator), 4,
      outputFile, errorsFile,
270                        filename);
271   }
272 }
```

Listing 5.25: Output.cpp (gekürzt)

Die Methode iteriert über alle Elemente des initialen Containers und gibt dann deren Inhalte aus. Die aufgerufene Funktion `printCallGraphItem` arbeitet dann rekursiv, falls das übergebene Item ein Mode Switch ist.

```
232 void printCallGraphItem(CallGraphItem *callGraphItem, int
      indentationLevel,
233                        ofstream *outputFile, ofstream *
      errorsFile,
234                        string &filename) {
235
236   CallEntry *itemAsCallEntry = llvm::dyn_cast<CallEntry>(
      callGraphItem);
237   GlobalVariableAccess *itemAsGlobalVariableAccess = llvm::
      dyn_cast<
238     GlobalVariableAccess>(callGraphItem);
239   ModeSwitch *itemAsModeSwitch = llvm::dyn_cast<ModeSwitch
      >(callGraphItem);
240
241   if (itemAsCallEntry) {
242     printCallAdvanced(itemAsCallEntry, outputFile,
      indentationLevel);
243   } else if (itemAsGlobalVariableAccess) {
244     printAccessAdvanced(itemAsGlobalVariableAccess,
```

```
      outputFile, errorsFile,
245                           filename, indentationLevel);
246   } else if (itemAsModeSwitch) {
247     printModeSwitchBegin(indentationLevel, outputFile);
248     for (auto modeSwitchEntry : *itemAsModeSwitch->
      getModeEntries()) {
249       printModeSwitchEntryBegin(indentationLevel + 1,
      outputFile);
250       for (auto item : modeSwitchEntry->getChildren()) {
251         printCallGraphItem(item, indentationLevel + 2,
      outputFile, errorsFile, filename);
252       }
253       printModeSwitchEntryEnd(indentationLevel + 1,
      outputFile);
254     }
255     printModeSwitchEnd(indentationLevel, outputFile);
256   } else {
257     cout << "Unknown call graph item occured." << endl;
258   }
259 }
```

Listing 5.26: Output.cpp (gekürzt)

Durch entsprechendes Casting (mittels LLVM-RTTI) wird der genaue Typ des übergebenen Items bestimmt. Je nachdem wird dann eine passende Untermethode aufgerufen, die das jeweilige Objekt ausgibt.

Dabei ist den bestehenden Methoden `printCallAdvanced` und `printAccessAdvanced` ein neuer Parameter `indentationLevel` hinzugefügt worden. Dieser hat den Zweck, die Einrückung der Tags im XML passend anzupassen, da diese durch die Mode Switches beliebig tief geschachtelt werden können. Die übergebene Zahl ist dabei ein Multiplikator und wird mit 2 multipliziert. Ein Level von 1 entspricht also einer Einrückung um 2 Leerzeichen. Beträgt dieses 2, werden 4 Leerzeichen vor dem XMLTag angefügt. Dieses Verhalten wird durch eine separate Methode `printIndentation` erzielt, die intern jeweils in den Untermethoden mit dem übergebenen Level aufgerufen wird:

```
123 void printIndentation(int indentationLevel, ofstream *
      outputFile)
124 {
125   for (int i = 0; i < indentationLevel; ++i)
126   {
127     *outputFile << "  ";
128   }
129 }
```

Listing 5.27: Output.cpp (gekürzt)

Die Untermethoden werden wie folgt ergänzt:

```
131 void printAccessAdvanced(GlobalVariableAccess *gvE,
    ofstream *outputFile, ofstream *errorsFile, string &
    filename, int indentationLevel) {
132   try {
133     if (gvE && gvE != nullptr && gvE->getGlobalVariable()
    && gvE->
134         getGlobalVariable() != nullptr) {
135       // ...
136       if (!isLocalPointerOrParameter(variableName,
    allTraversingData.at(0)) &&
137           isGlobalVariableDeclared(gvE, allTraversingData.
    at(0))) {
138         if (cleanedName.compare(name) == 0 || (
139             name.find(cleanedName) >= 0 && cleanedName.
    find("#") == string::
140             npos)) {
141           printIndentation(indentationLevel, outputFile);
142           *outputFile <<
143             "<stmtseq xsi:type=\"scair:LabelAccess\"
    label=\"" <<
144             cleanedName;
145           *outputFile << "?type=" << type << "\" Access=\""
     << gvE->
146             getAccessType() << "\" srcline=\"" << gvE->
    getLine() << "\"" <<
147             " srccol=\"" << gvE->getColumn() << "\"/>" <<
     endl;
148         }
149       } else if (
150         isLocalPointerOrParameter(variableName,
    allTraversingData.at(0)) || (
151           name.find_first_of("-FPTR") == name.length() - 5)
    ) {
152         printIndentation(indentationLevel, outputFile);
153         *outputFile << "<stmtseq xsi:type=\"scair:
    LabelAccess\" label=\"" << variableName;
154         // ...
155       }
156       // ...
157 }
```

Listing 5.28: Output.cpp (gekürzt)

Die Aufrufe wurden im Codeauszug 5.28 entsprechend in Zeile 141 und 152 eingefügt.

```
204 void printCallAdvanced(CallEntry *fE, ofstream *outputFile,
       int indentationLevel) {
205   printIndentation(indentationLevel, outputFile);
206   *outputFile << "<stmtseq xsi:type=\"scair:FunctionCall\""
       ;
207   *outputFile << " calls=\"" << fE->getFunctionID()->
       getFunctionName() <<
208       "?type=Function\"" << " srcline = \"" << fE->getLine
       () << "\" srccol=\""
209       << fE->getColumn() << "\"/>" << endl;
210 }
```

Listing 5.29: Output.cpp (gekürzt)

Nun müssen nur noch Ausgaben für die Mode Switches erfolgen. Da es sich bei diesen um Containerelemente handelt, umschließen sie die anderen, inneren Elemente. Daher müssen Start- und End-Tags jeweils einzeln ausgegeben werden. Die Methoden `printModeSwitchBegin` und `printModeSwitchEnd` ergeben sich wie folgt:

```
212 void printModeSwitchBegin(int indentationLevel, ofstream *
       outputFile) {
213     printIndentation(indentationLevel, outputFile);
214     *outputFile << "<stmtseq xsi:type=\"scair:ModeSwitch\">
       " << endl;
215 }
```

Listing 5.30: Output.cpp (gekürzt)

```
217 void printModeSwitchEnd(int indentationLevel, ofstream *
       outputFile) {
218     printIndentation(indentationLevel, outputFile);
219     *outputFile << "</stmtseq>" << endl;
220 }
```

Listing 5.31: Output.cpp (gekürzt)

Auch hier wird die Einrückung wieder berücksichtigt, da auch die Mode Switches selbst geschachtelt werden können und dann in der Hierarchie weiter unten auftreten. Codeauszug 5.26 zeigt in Zeile 248 dann die Iteration über alle in diesem Switch enthaltenen Mode Switch Entries. Für das aktuelle Element wird dann zunächst analog ein Start-Tag gesetzt und daraufhin für jedes Item in diesem Mode Switch Entry die `printCallGraphItem`-Methode rekursiv mit erhöhter Einrückung aufgerufen. Abschließend wird noch der zugehörige End-Tag ausgegeben.

Die Implementierung der Ausgaben für die Mode Switch Entries ist analog implementiert:

```
222 void printModeSwitchEntryBegin(int indentationLevel,
      ofstream *outputFile) {
223     printIndentation(indentationLevel, outputFile);
224     *outputFile << "<modeswitchentry>" << endl;
225 }
```

Listing 5.32: Output.cpp (gekürzt)

```
227 void printModeSwitchEntryEnd(int indentationLevel, ofstream
      *outputFile) {
228     printIndentation(indentationLevel, outputFile);
229     *outputFile << "</modeswitchentry>" << endl;
230 }
```

Listing 5.33: Output.cpp (gekürzt)

Damit ist die Implementierung im C++-Teil abgeschlossen, da die restliche Codelogik bereits angepasst wurde und die `parseIfStmt`-Methode die relevanten Aufgaben für die strukturelle Darstellung von Mode Switches übernimmt. Würde man nun weitere Konstrukte unterstützen wollen, so müsste man ebenso an dieser Stelle ansetzen. Eine Erweiterung des Plugins ist somit jederzeit gut und flexibel möglich. Für solche Fälle müssten dann nur weitere, passende Modellklassen eingeführt werden. Der hier vorgestellte Ansatz mit den Containern kann jedoch auch für diese direkt genutzt werden, da geschachtelte Strukturen jeglicher Art damit darstellbar sind und somit auch auf Schleifen und andere Konstrukte übergreifen.

5.3 Java-Teil

Um nun die entsprechenden Datenstrukturen der SCA-IR auch passend im Java-Teil repräsentieren zu können, muss das entsprechende Modell im Java-Plugin angepasst werden. Das Paket `org.eclipse.app4mc.sca2amalthea.ir` enthält dabei die entsprechende Repräsentation, die dann später automatisiert aus der vorliegenden XML-Datei erzeugt werden kann. Der Modellcode wird dabei nicht händisch entwickelt, sondern mithilfe des Eclipse Modeling Frameworks (kurz *EMF*) erzeugt. Hierbei steht ein Editor bereit, der es erlaubt, die einzelnen Klassen mit ihren Attributen, Assoziationen und Vererbungshierarchien in Form eines UML-Diagramms zu erstellen. Das Modell selbst wird im XML-Format in einer `.ecore`-Datei gespeichert. Innerhalb dieser muss auch für die Erweiterung angesetzt werden. Eine händische Implementierung der neuen Klassen ist wenig sinnvoll, da alle Komponenten im Zusammenspiel betrachtet werden müssen und viele weitere Metadaten benötigt werden.

Die im Auszug 5.3 gezeigte XML-Struktur kann direkt übernommen und in das Modell eingepflegt werden. Die Liste von `StmtCall`s kann in der `Function`-Klasse entsprechend beibehalten werden, da die `ModeSwitch`es auch als solche behandelt werden sollen. Folglich werden diese als entsprechende Unterklasse von `StmtCall` implementiert. Ein `ModeSwitch` selbst speichert dann eine Liste von `ModeSwitchEntry` -Objekten, die wiederum weitere `StmtCall`s zur Verfügung stellen. Auf diese Weise kann das Kompositum-Softwaremuster passend umgesetzt werden. Der `scair.ecore` -Datei müssen hierfür nach Zeile 93 folgende Zeilen angefügt werden:

```
94 <eClassifiers xsi:type="ecore:EClass" name="ModeSwitchEntry
     ">
95     <eStructuralFeatures xsi:type="ecore:EReference" name="
     stmtseq" upperBound="-1" eType="#//StmtCall" containment
     ="true"/>
96 </eClassifiers>
97 <eClassifiers xsi:type="ecore:EClass" name="ModeSwitch"
     eSuperTypes="#//StmtCall">
98     <eStructuralFeatures xsi:type="ecore:EReference" name="
     modeswitchentry" upperBound="-1" eType="#//
     ModeSwitchEntry" containment="true"/>
99 </eClassifiers>
```

Listing 5.34: ModeSwitchEntry- und ModeSwitch-Deklaration im Modell

Die Datei kann dann gespeichert werden und sollte automatisch die beiliegende `scair .genmodel`-Datei aktualisieren. Innerhalb dieser kann durch einen Rechtsklick auf den Knoten `scair` mit dem Menüeintrag `Generate Model Code` der zugehörige Java-Code erzeugt werden. Die Schnittstellendefinitionen `ModeSwitch` und `ModeSwitchEntry` sowie die jeweiligen Implementierungen im Ordner `impl` sollten dann vorhanden sein.

Da bestehende Implementierungen von zusätzlich deklarierten, eigenen Operationen in den Modellklassen auf diese Weise verloren gehen und daher initial zu einer Ausnahme für fehlende Methodenimplementierung (`UnsupportedOperationException`) führen, müssen diese manuell wieder eingepflegt werden. Dies beschränkt sich hier auf die Klasse `IdentifiableElementImpl.java`, deren bestehenden Inhalt man aus dem offiziellen git-Repository[2] kopieren kann. Möglich ist dies, da sich durch die Anpassungen im Rahmen dieser Arbeit für diese keine Änderungen ergeben.

Betrachtet man die `.ecore`-Datei genauer, so fällt eine Deklaration von `CallGraph` auf, die jedoch nicht weiter initialisiert wurde. Bosch hat offenbar an dieser Stelle die Darstellung der jeweiligen, detaillierten Funktionsabläufe im XML-Format vorgesehen. Die Nutzung des alten Klassennamens zeigt auf, dass der Ansatz an dieser Stelle vorgesehen war, aber dann nicht mehr ausgearbeitet wurde. Kapitel 4 hat gezeigt, dass die gefundenen Funktionsaufrufe und Variablenzugriffe jedoch auch mit dem aktuellen Format in einem `ActivityGraph`-Container aufgelistet werden. Entsprechend ist der in dieser Arbeit gewählte Ansatz, die genauen Ablaufsinformationen bezüglich alternativer Ausführungspfade ebenso direkt in die Ausdruckssequenzen aufzunehmen, sinnvoll gewählt. Da der restliche Code im SCA-Projekt jedoch nicht verändert, sondern nur um einen additiven `CallGraphItemContainer` erweitert wurde, könnte man eine alternative Implementierung über einen eigenen `ActivityGraph`-Tag ebenso problemlos umsetzen, da alle Elemente in diesem Container enthalten sind und die Wahl des Ausgabeformates in der Klasse `Output.cpp` flexibel gewählt werden kann.

Sobald das zugrundeliegende Modell existiert, muss nur noch die Übersetzung in den Softwareteil des AMALTHEA-Modells implementiert werden. Diese erfolgt im Paket `org.eclipse.app4mc.sca2amalthea.exporter` in der Klasse `SwModelTransformer`. Die einzelnen `ActivityGraph`-Elemente werden im AMALTHEA-Modell jeweils den einzelnen `Task`s und `Runnable`s hinzugefügt, sodass diese später in der darstellenden Struktur des APP4MC eingesehen werden können.

Die Methode `transform` extrahiert dabei alle vorhandenen und definierten Funktionen aus der beiliegenden Ressource, die die modellbasierte Instanz der übergebenen SCAIR repräsentiert. Dazu werden alle vorhandenen Container durchlaufen und dann deren zugehörige Funktionen gesammelt. Eine folgende Iteration über die extrahierten Funktionen prüft dann jeweils den zugehörigen Typ (Task, Runnable oder Unterbrechungsroutine) und ruft eine passende, interne Transformationsmethode auf, die dann die Erstellung des Aktivitätsgraphen übernimmt. Der Quellcode dieser Methode ist im nachfolgenden Auszug 5.35 zu sehen.

[2] https://git.eclipse.org/r/plugins/gitiles/app4mc/org.eclipse.app4mc.tools/+/13baf17cfe40fef2b3a58f27ee044e3afa255cfa/eclipse-tools/sca2amalthea/plugins/org.eclipse.app4mc.sca2amalthea.ir/src/org/eclipse/app4mc/sca2amalthea/ir/scair/impl/IdentifiableElementImpl.java

```
85 public void transform(final Amalthea amaltheaModel, final
       SCAResource resource) {
86         SWModel swModel = AmaltheaFactory.eINSTANCE.
       createSWModel();
87         amaltheaModel.setSwModel(swModel);
88 
89         this.typeExporter.transform(resource, swModel);
90         transformLabels(resource, swModel);
91 
92         Set < Function > allFunctions = getAllfunctions(
       resource);
93         Set < Function > ignoreList = prepareIgnoreList(
       allFunctions);
94         allFunctions.removeAll(ignoreList);
95         EList < Runnable > runnables = swModel.getRunnables
       ();
96         EList < Task > tasks = swModel.getTasks();
97         EList < ISR > isrs = swModel.getIsrs();
98         transformFirstLevel(allFunctions, runnables, tasks,
        isrs);
99 
100        for (Function func: allFunctions) {
101            try {
102                if (func.getType().getLiteral().
       equalsIgnoreCase(EFunctionTypeEnum.RUNNABLE.getLiteral()
       )) {
103                    transformRunnableInternals(func);
104                } else if (func.getType().getLiteral().
       equalsIgnoreCase(EFunctionTypeEnum.TASK.getLiteral())) {
105                    transformTaskInternals(func);
106                } else if (func.getType().getLiteral().
       equalsIgnoreCase(EFunctionTypeEnum.ISR.getLiteral())) {
107                    transformISRInternals(func);
108                }
109            } catch (Exception e) {
110                LogUtil.log(LLVMLogUtil.LOG_MSG_ID,
       Severity.DEBUG, "**Function transformation problem" +
       func.getName(),
111                    this.getClass(), Activator.PLUGIN_ID);
112                LogUtil.logException(LLVMLogUtil.LOG_MSG_ID
       , e.getClass(), e, Activator.PLUGIN_ID);
113            }
114        }
115     }
```

Listing 5.35: transform-Methode der SwModelTransformer-Klasse (gekürzt)

Um nun in diesen Prozess eingreifen und die Mode Switches hinzufügen zu können, muss betrachtet werden, welche Aufgaben die einzelnen, aufgerufenen Methoden jeweils erledigen.

Betrachtet man die Transformation der Runnables, so sieht man, dass dort die Liste der `stmtseq`-Objekte durchlaufen und für jedes Element eine Typprüfung vollzogen wird.

```
245 private void transformRunnableInternals(final Function
246         function) {
247         String name = function.getName();
248         Runnable runnable = this.data.getRunnableMap().get(
    name);
249 
250         EList < StmtCall > stmtseq = function.getStmtseq();
251         for (StmtCall stmtCall: stmtseq) {
252             if (stmtCall instanceof FunctionCall) {
253                 FunctionCall fc = (FunctionCall) stmtCall;
254                 String calledFunc = fc.getCalls().getName()
    ;
255 
256                 if (this.data.getSemaMap().keySet().
    contains(fc.getCalls().getName())) {
257                     String srcLine = fc.getSrcline();
258                     String srcCol = fc.getSrccol();
259                     checkForSemaphoreCall(runnable,
    calledFunc, srcLine, srcCol);
260                 } else {
261                     handleRunnableCall(name, runnable, fc);
262                 }
263             } else if (stmtCall instanceof LabelAccess) {
264                 handleLabelAccess(runnable, stmtCall);
265             }
266         }
267      }
```

Listing 5.36: transformRunnableInternals-Methode der SwModelTransformer-Klasse

Zum Stand dieser Arbeit können Funktionsaufrufe und Variablenzugriffe erkannt und behandelt werden. An dieser Stelle muss folglich angesetzt werden, um ein `ModeSwitch` berücksichtigen zu können, da dieses, wie zuvor erläutert, ebenso von der `StmtCall`-Klasse erbt. Dies geschieht über ein weiteres `else if`, welches auf den Typ `ModeSwitch` prüft. Die weiteren Schritte werden dann in eine neue Methode `handleModeSwitch(Runnable, StmtCall)` ausgelagert, welche im weiteren Verlauf dieses Unterkapitels noch definiert werden soll.

Nun stellt sich die Frage, ob man diese Methode `handleModeSwitch(Runnable, StmtCall)` auch direkt in den anderen Transformationsmethoden für Tasks und Unterbrechungsroutinen anwenden kann. Die Verwaltung von Tasks erfolgt in einer Funktion `transformTaskInternals`, welche zur Erstellung des Aktivitätsgraphens noch eine weitere Methode `transformStatements` aufruft. Interessant ist bei dieser, dass diese analog zu der der Runnables arbeitet, jedoch nur auf Funktionsaufrufe prüft und diese behandelt. Variablenzugriffe (`Label Access`) werden nicht beachtet. Aus diesem Grund sind bei der Darstellung von Tasks im APP4MC auch nur Funktionsaufrufe aufgelistet (vgl. nachfolgende Abbildung 5.4)

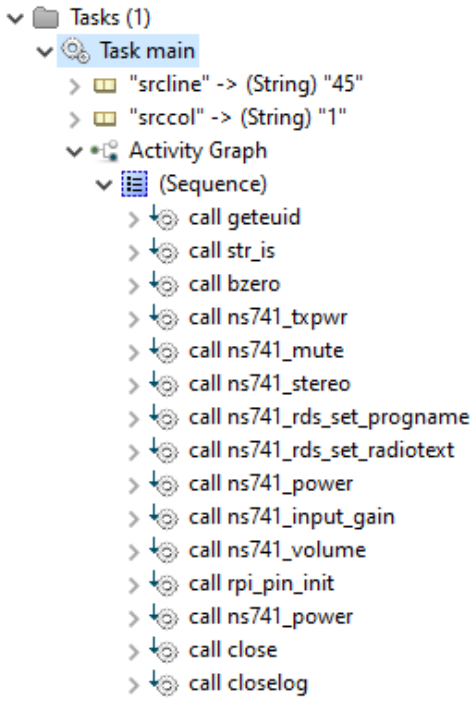


Abbildung 5.4: Auflistung von Funktionsaufrufen bei Tasks

Entsprechend sollen bei dieser die Mode Switches vernachlässigt werden und keine Erweiterung der Funktion stattfinden. Genau dasselbe gilt auch für Unterbrechungsroutinen (ISR), welche komplett analog behandelt werden. Man kann sich bei der Erweiterung daher komplett auf die Runnables beschränken.

Zunächst müssen hierfür die notwendigen Modellklassen importiert werden:

```
1 import org.eclipse.app4mc.sca2amalthea.ir.scair.
    ModeSwitchEntry;
2 import org.eclipse.app4mc.sca2amalthea.ir.scair.ModeSwitch;
```

Die Klassen des AMALTHEA-Modells selbst werden nicht importiert, da sie bei Verwendung immer mit dem vollständigen Paketnamen angegeben werden und sonst aufgrund der gleichen Klassennamen im IR-Paket eine Kollision auftritt. Die überarbeitete Methode `transformRunnableInternals` ergibt sich dann wie folgt:

```
245 private void transformRunnableInternals(final Function
246     function) {
247     String name = function.getName();
248     Runnable runnable = this.data.getRunnableMap().get(name
      );
249     EList < StmtCall > stmtseq = function.getStmtseq();
250     for (StmtCall stmtCall: stmtseq) {
251         if (stmtCall instanceof FunctionCall) {
252             FunctionCall fc = (FunctionCall) stmtCall;
253             String calledFunc = fc.getCalls().getName();
254             if (this.data.getSemaMap().keySet().contains(fc
      .getCalls().getName())) {
255                 String srcLine = fc.getSrcline();
256                 String srcCol = fc.getSrccol();
257                 checkForSemaphoreCall(runnable, calledFunc,
       srcLine, srcCol);
258             } else {
259                 handleRunnableCall(name, runnable, fc);
260             }
261         } else if (stmtCall instanceof LabelAccess) {
262             handleLabelAccess(runnable, stmtCall);
263         } else if (stmtCall instanceof ModeSwitch) {
264             handleModeSwitch(runnable, stmtCall);
265         }
266     }
267 }
```

Listing 5.37: Neue transformRunnableInternals-Methode der SwModelTransformer-Klasse

Nun muss diese Methode entsprechend implementiert werden. Zuvor ist es jedoch relevant, sich nochmals über die Struktur der `CallGraphItems` Gedanken zu machen. Da ein `ModeSwitch` selbst wieder `ModeSwitchEntry`-Instanzen enthält und diese rekursiv weitere `CallGraphItems` speichern, müssen die bestehenden Methoden auf Kompatibilität mit diesem neuen Ansatz geprüft werden. Bei Betrachtung der jeweiligen Methoden `handleLabelAccess` bzw. `handleRunnableCall` fällt auf, dass diese die jeweiligen Items nur zu einem `Runnable`-Objekt hinzufügen können. Möchte man diese stattdessen nun einem `ModeSwitchEntry` anhängen, so ist dies mit dem bisherigen Ansatz nicht möglich.

Eine Idee wäre, `Runnable` und `ModeSwitchEntry` erneut über eine Schnittstelle zu abstrahieren (bspw. `ActivityGraphContainer`) und diese als Typ in die jeweiligen Methoden aufzunehmen. Um jedoch nicht weitere Anpassungen am Modell vorzunehmen, sollen stattdessen jeweils zwei verschiedene Methoden deklariert werden, die dies für jeden der beiden Typen (`Runnable` und `ModeSwitchEntry`) übernehmen. Wäre in Zukunft geplant, diese Items auch zu Tasks und Unterbrechungsroutinen hinzuzufügen, so wäre eine direkte Abstraktion im Modell sinnvoller. Um das Ziel dieser Erweiterung zu erreichen, reicht dieser Ansatz jedoch völlig aus. Als Konvention sollen die Methoden, die Items zu `ModeSwitch`-Instanzen hinzufügen, nach der Struktur `handle...OnModeSwitch` benannt werden. Somit müssen alle relevanten, intern aufgerufenen Methoden der Funktion `transformRunnableInternals` dahingehend angepasst werden. Um das DRY-Prinzip (*Do Not Repeat Yourself*) nicht zu verletzen, wird die Erstellungslogik der jeweiligen AMALTHEA-Modellelemente in eine separate Methode ausgelagert, die dann in diesen Methoden aufgerufen wird. Der Aufbau für die Behandlung von `RunnableCalls` ergibt sich dann wie folgt:

```
276 private org.eclipse.app4mc.amalthea.model.RunnableCall
    buildRunnableCall(final Runnable calledRunnable, final
    FunctionCall fc) {
277     org.eclipse.app4mc.amalthea.model.RunnableCall
    runnableCall = AmaltheaFactory.eINSTANCE.
    createRunnableCall();
278     runnableCall.setRunnable(calledRunnable);
279     CustomPropertiesAdder.addSourceLineInformation(
    runnableCall, fc.getSrcline(), fc.getSrccol());
280     return runnableCall;
281 }
282
283 /**
284  * @param name
285  * @param runnable
286  * @param fc
287  */
288 private void handleRunnableCall(final String name, final
    Runnable runnable, final FunctionCall fc) {
289     String calledFunc = fc.getCalls().getName();
290     Runnable calledRunnable = this.data.getRunnableMap().
    get(calledFunc);
291     if (calledRunnable != null) {
292         runnable.getRunnableItems().add(buildRunnableCall(
    calledRunnable, fc));
293     } else {
294         LogUtil.log(
295             LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG, "Called
     Runnable does not exist for ***" + calledFunc + "*** in
```

```
        *" +
296                 name + "*Possibly it appeared in ignored
    hierarchy of Lock Functions",
297                 this.getClass(), Activator.PLUGIN_ID);
298         }
299 }
300
301 private void handleRunnableCallOnModeSwitch(final String
    name, final ModeSwitchEntry modeSwitchEntry, final
    FunctionCall fc) {
302         String calledFunc = fc.getCalls().getName();
303         Runnable calledRunnable = this.data.getRunnableMap().
    get(calledFunc);
304         if (calledRunnable != null) {
305             modeSwitchEntry.getItems().add(buildRunnableCall(
    calledRunnable, fc));
306         } else {
307             LogUtil.log(
308                 LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG, "Called
     Runnable does not exist for ***" + calledFunc + "*** in
     *" +
309                 name + "*Possibly it appeared in ignored
    hierarchy of Lock Functions",
310                 this.getClass(), Activator.PLUGIN_ID);
311         }
312 }
```

Listing 5.38: Behandlung von Funktionsaufrufen

Die bestehende Architektur wird somit nicht zerstört und die initialen Aufrufe werden weiterhin gleich behandelt. Sie wird lediglich um eine weitere Methode erweitert, die im Verlauf der rekursiven Struktur dann noch für die `ModeSwitches` verwendet werden kann. Variablenzugriffe (`LabelAccess`-Instanzen) werden dann wie folgt behandelt:

```
314 private org.eclipse.app4mc.amalthea.model.LabelAccess
    buildLabelAccess(LabelAccess labelAccess) {
315         org.eclipse.app4mc.amalthea.model.LabelAccess
    amLabelAccess = AmaltheaFactory.eINSTANCE.
    createLabelAccess();
316         amLabelAccess.setData(this.data.getLabelMap().get(
    labelAccess.getLabel().getName()));
317         if ("Read".equalsIgnoreCase(labelAccess.getAccess().
    getLiteral())) {
318             amLabelAccess.setAccess(LabelAccessEnum.READ);
319         } else {
320             amLabelAccess.setAccess(LabelAccessEnum.WRITE);
```

```
321     }
322
323     CustomPropertiesAdder.addSourceLineInformation(
        amLabelAccess, labelAccess.getSrcline(), labelAccess.
        getSrccol());
324     return amLabelAccess;
325 }
326
327 /**
328  * transform SCAIR labelAccess to AMALTHEA Label Access in
        a runnable
329  *
330  * @param runnable
331  * @param stmtCall
332  */
333 private void handleLabelAccess(final Runnable runnable,
        final StmtCall stmtCall) {
334     LabelAccess labelAccess = (LabelAccess) stmtCall;
335     if ((labelAccess != null) && (labelAccess.getLabel() !=
        null)) {
336         runnable.getRunnableItems().add(buildLabelAccess(
        labelAccess));
337     } else {
338         LogUtil.log(LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG,
339             "**LabelAccess transformation problem in
        function " + runnable.getName(), this.getClass(),
340             Activator.PLUGIN_ID);
341     }
342 }
343
344 private void handleLabelAccessOnModeSwitch(final
        ModeSwitchEntry modeSwitchEntry, final StmtCall stmtCall
        ) {
345     LabelAccess labelAccess = (LabelAccess) stmtCall;
346     if ((labelAccess != null) && (labelAccess.getLabel() !=
        null)) {
347         modeSwitchEntry.getItems().add(buildLabelAccess(
        labelAccess));
348     } else {
349         LogUtil.log(LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG,
350             "**LabelAccess transformation problem in mode
        switch entry", this.getClass(),
351             Activator.PLUGIN_ID);
352     }
353 }
```

Listing 5.39: Behandlung von Variablenzugriffen

Analog müssen nun noch die `ModeSwitches` selbst behandelt werden. Es folgt entsprechend die Implementierung der neu eingeführten Methode `handleModeSwitch`, die zuvor genannt wurde. Um zu wissen, wie die Schnittstelle der `ModeSwitches` und `ModeSwitchEntry`-Objekte im AMALTHEA-Modell (d.h. der entsprechenden API im Plugin) zu nutzen ist, muss man sich deren Quellcode im offiziellen git-Repository ansehen. Definiert ist das Modell im Paket `org.eclipse.app4mc.amalthea.model`. Es basiert ebenso auf dem EMF und enthält daher auch eine Modell-Datei und die aus dieser generierten Codedateien. Letztere befinden sich unter `xcore-gen/org/eclipse/app4mc/amalthea/model`[3]. Analog findet man dort die entsprechenden Schnittstellendefinitionen zusammen mit den zugehörigen Implementierungen im Unterordner `impl`. Basierend auf den dort zu entnehmenden Informationen kann man die `ModeSwitch`-Elemente nun bauen. Wichtig ist dabei die volle Paketangabe für die AMALTHEA-Modellelemente, um diese von den Klassen des SCAIR-Modells zu unterscheiden.

```
355 private org.eclipse.app4mc.amalthea.model.ModeSwitch
    buildModeSwitch(final String name, final ModeSwitch
    modeSwitch) {
356     org.eclipse.app4mc.amalthea.model.ModeSwitch
    amModeSwitch = AmaltheaFactory.eINSTANCE.
    createModeSwitch();
357     for (ModeSwitchEntry me: modeSwitch.getmodeswitchentry
    ()) {
358         org.eclipse.app4mc.amalthea.model.ModeSwitchEntry
    amModeSwitchEntry = AmaltheaFactory.eINSTANCE.
    createModeSwitchEntry();
359         EList < StmtCall > modeSwitchEntryStmtseq = me.
    getStmtseq();
360         for (StmtCall stmtCall: modeSwitchEntryStmtseq) {
361             if (stmtCall instanceof FunctionCall) {
362                 FunctionCall fc = (FunctionCall) stmtCall;
363                 String calledFunc = fc.getCalls().getName()
    ;
364                 if (this.data.getSemaMap().keySet().
    contains(fc.getCalls().getName())) {
365                     String srcLine = fc.getSrcline();
366                     String srcCol = fc.getSrccol();
367                     checkForSemaphoreCallOnModeSwitch(
    amModeSwitchEntry, calledFunc, srcLine, srcCol);
368                 } else {
369                     handleRunnableCallOnModeSwitch(name,
    amModeSwitchEntry, fc);
370                 }
371             } else if (stmtCall instanceof LabelAccess) {
```

[3] https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.git/tree/plugins/org.eclipse.app4mc.amalthea.model/xcore-gen/org/eclipse/app4mc/amalthea/model

```
372                 handleLabelAccessOnModeSwitch(
    amModeSwitchEntry, stmtCall);
373             } else if (stmtCall instanceof ModeSwitch) {
374                 handleModeSwitchOnModeSwitch(name,
    amModeSwitchEntry, stmtCall);
375             }
376         }
377         amModeSwitch.getEntries().add(amModeSwitchEntry);
378     }
379     return amModeSwitch;
380 }
381
382 private void handleModeSwitch(final String name, final
    Runnable runnable, final StmtCall stmtCall) {
383     ModeSwitch modeSwitch = (ModeSwitch) stmtCall;
384     if ((modeSwitch != null)) {
385         runnable.getRunnableItems().add(buildModeSwitch(
    name, modeSwitch));
386     } else {
387         LogUtil.log(LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG,
388             "**ModeSwitch transformation problem in
    function " + runnable.getName(), this.getClass(),
389             Activator.PLUGIN_ID);
390     }
391 }
392
393 private void handleModeSwitchOnModeSwitch(final String name
    , final ModeSwitchEntry modeSwitchEntry, final StmtCall
    stmtCall) {
394     ModeSwitch modeSwitch = (ModeSwitch) stmtCall;
395     if ((modeSwitch != null)) {
396         modeSwitchEntry.getItems().add(buildModeSwitch(name
    , modeSwitch));
397     } else {
398         LogUtil.log(LLVMLogUtil.LOG_MSG_ID, Severity.DEBUG,
399             "**ModeSwitch transformation problem in mode
    switch", this.getClass(),
400             Activator.PLUGIN_ID);
401     }
402 }
```

Listing 5.40: Behandlung von Mode Switches

Da diese Methode etwas mehr Codelogik enthält, um die Containerstruktur passend zu berücksichtigen, soll diese kurz erläutert werden. Betrachtet man diese genauer, fällt auf, dass sie relativ ähnlich zu der Methode `transformRunnableInternals` ist.

Dies ist logisch, da die `Runnables` ebenso als (initiale) Container dienen. Zunächst wird in Zeile 356 ein neues AMALTHEA-`ModeSwitch`-Objekt für die aktuelle SCAIR-`ModeSwitch`-Instanz erzeugt. Anschließend wird über zugehörigen `ModeEntries` iteriert und jeweils ein SCAIR-`ModeSwitchEntry`-Objekt erzeugt. Da dieses weitere Ausdruckssequenzen (`stmtseq`/Items) enthält, werden jene in Zeile 359 mit der im SCAIR-Modell definierten Methode abgerufen und dann jeweils eine analoge Typprüfung vorgenommen. Anders als in der `transformRunnableInternals`-Methode werden hier für jeden Typen jedoch nun die zuvor definierten, für `Mode Switches` spezifischen Funktionen aufgerufen. Auf diese Weise wird in Zeile 372 eine Rekursion erzeugt, da die Funktion `handleModeSwitch` selbst die Methode `buildModeSwitch` aufruft. Somit ist die Struktur darstellbar, wenn `Mode Switches` in anderen `Mode Switches` geschachtelt werden. Der Abbruch der Rekursion ist implizit durch das Finden und Behandeln von Variablenzugriffen und Funktionsaufrufen gegeben, da diese Abschlusselemente des Kompositums darstellen und keine weiteren Rekursionen initiieren. Durch die rekursiven Aufrufe werden dem jeweiligen, übergebenen `ModeSwitchEntry` die Kindelemente Schritt für Schritt hinzugefügt. Sobald dieser vollständig ist, wird er dem `ModeSwitch` selbst hinzugefügt (Zeile 377).

Abschließend müssen noch die Semaphorenaufrufe angepasst werden, da auch diese in `transformRunnableInternals` vorkommen.

```
405 private SemaphoreAccess buildSemaphoreAccess(final String
    calledFunc, final String srcLine,
406     final String srcCol) {
407     Semaphore semaphore = this.data.getSemaMap().get(
    calledFunc);
408     SemaphoreAccess semaAcc = AmaltheaFactory.eINSTANCE.
    createSemaphoreAccess();
409     if (((StringObject) semaphore.getCustomProperties().get
    (CustomPropertiesAdder.GET_LOCK_FUNC_NAME)).getValue()
410         .equalsIgnoreCase(calledFunc)) {
411
412         if (((StringObject) semaphore.getCustomProperties()
    .get(CustomPropertiesAdder.LOCK_TYPE)).getValue()
413             .equalsIgnoreCase(LockType.EXCLUSIVE_LOCK.name
    ())) {
414             semaAcc.setAccess(SemaphoreAccessEnum.EXCLUSIVE
    );
415         } else {
416             semaAcc.setAccess(SemaphoreAccessEnum.REQUEST);
417         }
418     } else if (((StringObject) semaphore.
    getCustomProperties().get(CustomPropertiesAdder.
    RELEASE_LOCK_FUNC_NAME))
419         .getValue().equalsIgnoreCase(calledFunc)) {
```

```
420             semaAcc.setAccess(SemaphoreAccessEnum.RELEASE);
421         }
422         semaAcc.setSemaphore(semaphore);
423         CustomPropertiesAdder.addSourceLineInformation(semaAcc,
            srcLine, srcCol);
424         return semaAcc;
425     }
426
427     private void checkForSemaphoreCallOnModeSwitch(final
            ModeSwitchEntry modeSwitchEntry, final String calledFunc
            , final String srcLine,
428         final String srcCol) {
429         modeSwitchEntry.getItems().add(buildSemaphoreAccess(
            calledFunc, srcLine, srcCol));
430     }
431
432     /**
433      * @param runnable
434      * @param calledFunc
435      */
436     private void checkForSemaphoreCall(final Runnable runnable,
            final String calledFunc, final String srcLine,
437         final String srcCol) {
438         runnable.getRunnableItems().add(buildSemaphoreAccess(
            calledFunc, srcLine, srcCol));
439     }
```

Listing 5.41: Behandlung von Semaphorenaufrufen

Weitere Anpassungen in dieser Klasse sind nicht mehr notwendig.

Das Hinzufügen von `Mode Switches` funktioniert mit diesem Stand jedoch noch nicht und diese bleiben leer, da keine Mode Switch Entries erkannt werden. Dies ist auf den proprietären XML-Serialisierer für das Einlesen der SCAIR zurückzuführen. Wichtig zu wissen ist, dass listenartige Attribute (d.h. Referenzen, die Container sind) in EMF-Modellklassen beim Einlesen aus einem passenden Format (etwa XML) über ihren Namen identifiziert werden. Dies bedeutet, dass die XML-Tagnamen für jeden Listeneintrag genau gleich heißen müssen. Hierbei ergeben sich Schwierigkeiten mit den Pluralformen der verschiedenen Klassennamen. Etwa möchte man in der Modellklasse `Container` die Liste von enthaltenen Funktionen `functions` benennen, um den Array-Datentyp zu implizieren. Folglich müssten allerdings auch alle Einträge im XML-Schema mit dem Tag `functions` angegeben werden, obwohl es sich bei den einzelnen Elementen nur um **eine** Funktion und keine Kollektion handelt. Dies ist für potentielle Leser unintuitiv. Das Problem ist exemplarisch im folgenden Codebeispiel ersichtlich.

```
<container>
    <functions src="..." col="...">
        ...
    </functions>
    <functions src="..." col="...">
        ...
    </functions>
</container>
```

Die `stmtseq`-Objekte stellen dies auch gut dar. Eine *Statement Sequence* ist eine Folge (d.h. eine Liste) von Ausdrücken. Ein einzelner Ausdruck wird in der XML-Datei jedoch auch über den Tag `stmtseq` identifiziert, was nicht ganz intuitiv ist. Die Problematik besteht aufgrund der Äquivalenz auch umgekehrt: Sind alternativ die XML-Tags sinnvoll benannt, muss das entsprechende Listenattribut in der Modellklasse im Singular angegeben werden, wodurch die intuitive Typerkennung bei der Nutzung des Attributs (bzw. dessen Getter-Methode) erschwert wird.

Um dieses Problem nun zu lösen, hat Bosch hierzu im Parser eine entsprechende Behandlung hinzugefügt. Die einzelnen Objekte sollen im XML-Schema im Singular benannt sein und dann automatisiert beim Einlesen in die Pluralform des Modellattributs gebracht werden. Das Paket `org.eclipse.app4mc.sca2amalthea.serialization` ist dabei für das Parsen verantwortlich. Intern wird dabei auf ein `SAXParser`-Objekt zurückgegriffen und die notwendigen Methoden überschrieben. Die genannte Anpassung findet in der Datei `SCAXMLHandler` statt. Beim Einlesen eines beginnenden Tags wird die Methode `startElement` mit den gefundenen Daten aufgerufen. Diese wird wie folgt überschrieben:

```
@Override
public void startElement(final String uri, final String
    localName, final String qName, final Attributes
    attributes)
throws SAXException {
    setAttributes(attributes);
    if (!qName.endsWith("s") && !qName.contains(PROJECT) &&
    !qName.endsWith("me") && !qName.endsWith("seq")) {
        startElement(uri, localName + "s", qName + "s");
    } else if (qName.contains(PROJECT) || qName.endsWith("
    me") || qName.endsWith("el") || qName.endsWith("seq")) {
        startElement(uri, localName, qName);
    }
}
```

Listing 5.42: SCAXMLHandler.java

Es ist ersichtlich, dass an bestimmte Tags automatisch ein *s* für die Pluralform angehängt wird. Auf diese Weise wird aus `container` der Name `containers` und aus `function` dann `functions`. Tagnamen, die bereits im Plural vorliegen, werden regulär behandelt (Zeile 54). An dieser Stelle erklärt sich nun, warum die `modeswitchentry` -Tags nicht korrekt eingelesen werden und dann später folglich auch nicht zur Verfügung stehen. Da alle Bedingungen der Abfrage in Zeile 51 zutreffen, wird an diesen Tag automatisch ein *s* angehängt und aus `modeswitchentry` wird fälschlicherweise `modeswitchentrys`. Dies muss bei allen neu eingeführten Tags beachtet werden. Die Lösung ist dann, eine entsprechende Prüfung für diese zu hinterlegen. Dazu kann theoretisch der ganze Tagname geprüft werden. Es reicht an sich jedoch für den Zweck dieser Erweiterung auch aus, nur einen Teil (etwa das Ende) zu vergleichen:

```
47 @Override
48 public void startElement(final String uri, final String
       localName, final String qName, final Attributes
       attributes)
49 throws SAXException {
50     setAttributes(attributes);
51     if (!qName.endsWith("s") && !qName.contains(PROJECT) &&
       !qName.endsWith("me") && !qName.endsWith("seq") && !
       qName.endsWith("entry")) {
52         startElement(uri, localName + "s", qName + "s");
53     } else if (qName.contains(PROJECT) || qName.endsWith("
       me") || qName.endsWith("el") || qName.endsWith("seq") ||
       qName.endsWith("entry")) {
54         startElement(uri, localName, qName);
55     }
56 }
```

Listing 5.43: SCAXMLHandler.java

Sobald dies erledigt ist, werden die `modeswitchentry`-Einträge unverändert eingelesen und auch hinzugefügt. Der Java-Teil muss dann nur noch gemäß der Anleitung in Kapitel 2 (neu) kompiliert und installiert werden.

5.4 Ergebnis

Es kann im fertigen AMALTHEA-Modell vorkommen, dass Mode Switch Entries ohne Inhalte vorkommen. Dies liegt nicht an fehlerhaftem Code, sondern der Tatsache, dass das Plugin zum Stand dieser Arbeit nur Funktionsaufrufe und Variablenzugriffe erkennt und auch listet. Falls keines dieser beiden Elemente innerhalb eines Mode Switch Entries auftritt, wird dieser als leer ausgewiesen. Würde man das Plugin nun um weitere Items erweitern und jegliche Codekonstrukte in dieses aufnehmen, so wären diese analog dort zu sehen.

Die folgende Abbildung zeigt einen Ausschnitt eines Runnables des in Kapitel 4 gezeigten Need4Stek-Projekts nach der Erweiterung des Plugins im Rahmen dieser Arbeit. Dieses verdeutlich die geschachtelte Struktur und weist gleichzeitig `Mode Switch Entry`-Einträge auf, die, wie eben beschrieben, keinen Inhalt haben. Alleinige `case`-Elemente in einem `Mode Switch` repräsentieren dabei `if`-Abfragen ohne `else`-Teil.

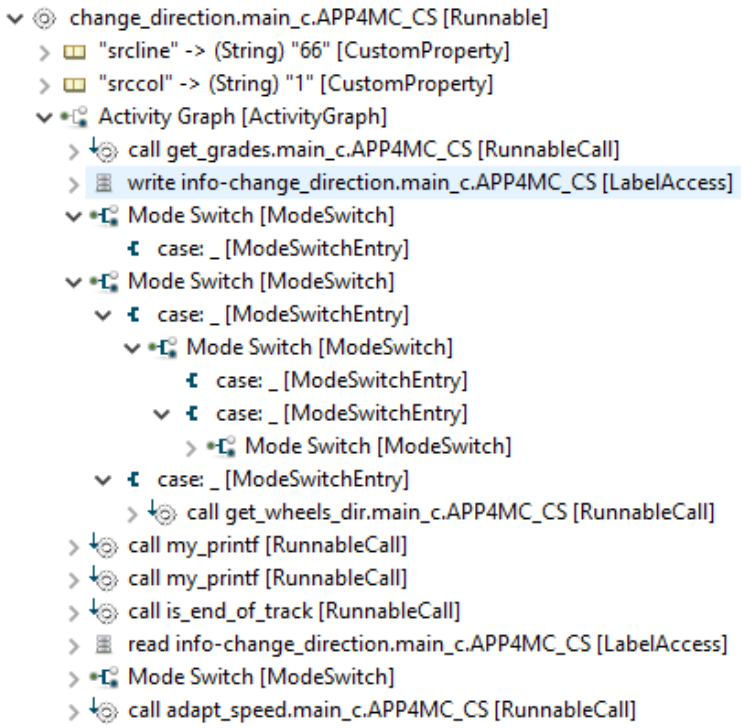


Abbildung 5.5: Auflistung von Mode Switches im AMALTHEA-Modell

Die einzelnen Fälle (`case` in den `Mode Switch`-Einträgen) haben dabei keine genauereren Spezifikationen. Diese ließen sich jedoch relativ leicht implementieren, indem man die spezifischen Informationen aus den zugehörigen clang-Klassen mit in die SCAIR aufnimmt und passende Attribute in den Java-Klassen einführt. So könnte man etwa die jeweilige Fallbedingung abrufen und mit der passenden Klasse im AMALTHEA-Modell hinterlegen.

Die Art und Weise, wie das Plugin für diese Funktionalität erweitert wurde, lässt sich auch auf alle anderen Arten von Elementen übertragen. Viele einfache Items können sogar direkt als `stmtseq`-Objekte eingepflegt werden, sodass diese sich unmittelbar in die Struktur integrieren. Für etwaige andere Konstrukte wie Schleifen oder allgemeine Sprungbefehle können analoge Modelle, XML-Strukturen und Funktionen eingeführt werden, um diese passend in das Modell aufzunehmen. Die AMALTHEA- und clang-Dokumentation sowie die jeweiligen git-Repositories liefern alle notwendigen Informationen, um das Plugin passend zu erweitern.

6 Alternative Idee für einen Erweiterungsansatz

Bei der Suche nach einer passenden Lösung wurde zunächst ein anderer Ansatz verfolgt, da Bosch im C++-Teil des Plugins auch schon Unterstützung für Kontrollflussgraphen (*CFG*) vorgesehen hat. Da zusätzliche Informationen zu verschiedenen Zyklen (Schleifen) und alternativen Ausführungspfaden vorhanden waren und die Nutzung der clang-spezifischen Klassen beim Ablaufen des Syntaxbaumes zunächst sehr komplex schien, haben im Rahmen dieser Arbeit Überlegungen stattgefunden, die bestehenden Kontrollflussgraphen für eine solche Erweiterung heranzuziehen und somit keine direkten Änderungen in den Besuchermethoden vorzunehmen zu müssen. Da alternative Ausführungspfade in so einem Graphen gut und schnell einsehbar sind, schien dieser Ansatz für eine On-Top-Erweiterung simpel.

Im Verlauf der Analyse und genaueren Betrachtung der CFG-Theorie hat sich jedoch herausgestellt, dass sich zu viele Mehrdeutigkeiten und Spezialfälle ergeben und folglich zu komplexe Algorithmen notwendig wären, um dies adäquat umzusetzen. Der Ansatz, die Informationen direkt beim Traversieren des abstrakten Syntaxbaums zu speichern, ist tatsächlich weitaus sinnvoller und einfacher. Um jedoch zu erläutern, warum genau ein solcher Ansatz nicht geeignet ist, sollen die gefundenen Ergebnisse nachfolgend trotzdem präsentiert werden. Sie haben jedoch keine Praxisrelevanz für die konkrete Erweiterung des Plugins.

6.1 Kontrollflussgraph und Basic Blocks

Um zunächst den zugrundeliegenden Aufbau der betroffenen Strukturen nachvollziehen zu können, sollen zunächst die relevanten Codeteile im C++-Projekt betrachtet werden.

```
112 list<BasicBlock>* FindInformationClassVisitor::
      buildCFGModel(clang::FunctionDecl* decl, Stmt *stmt) {
113   list<BasicBlock>* basicBlockList = new list<BasicBlock>()
      ;
114
115   if (decl && stmt) {
116     std::unique_ptr< CFG > srcCFG = CFG::buildCFG(decl,
      stmt, Context, clang::CFG::BuildOptions());
```

```
117
118     CFG* otherCFG = srcCFG.release();
119     parentMap = new ParentMap(decl->getBody());
120     mapOfCFGStmts = CFGStmtMap::Build(otherCFG, parentMap);
121
122     for (CFG::const_reverse_iterator it = otherCFG->rbegin
      (); it != otherCFG->rend(); ++it) {
123       const CFGBlock *CB = *it;
124       BasicBlock bb(CB->getBlockID());
125
126       for (CFGBlock::const_pred_iterator it_pred = CB->
      pred_begin(); it_pred != CB->pred_end(); ++it_pred) {
127         bb.addPred(it_pred->getReachableBlock()->getBlockID
      ());
128       }
129       for (CFGBlock::const_succ_iterator it_succ = CB->
      succ_begin(); it_succ != CB->succ_end(); ++it_succ) {
130         bb.addSucc(it_succ->getReachableBlock()->getBlockID
      ());
131       }
132       basicBlockList->push_back(bb);
133     }
134     delete otherCFG;
135   }
136   return basicBlockList;
137 }
```

Listing 6.1: TraverseASTClassAction.cpp

```
229 bool FindInformationClassVisitor::VisitFunctionDecl(clang::
      FunctionDecl* decl) {
230   // ...
231   try {
232     if (decl && decl->isThisDeclarationADefinition() &&
      decl->hasBody()) {
233       Stmt *stmt = decl->getBody();
234
235       if (stmt) {
236         list<BasicBlock>* basicBlockList;
237         if (isCFGActivated) {
238           basicBlockList = buildCFGModel(decl, stmt);
239         }
240       //   ...
```

Listing 6.2: TraverseASTClassAction.cpp (gekürzt)

Die Methode `buildCFGModel` gibt eine Liste von `BasicBlock`-Objekten zurück, die einen Funktionsablauf repräsentiert. Sie wird beim Besuchen einer Funktionsdeklaration aufgerufen und die zugehörige Liste dann später mit dieser assoziiert.

6.1.1 Allgemeine Basic-Block-Definition

Ein `BasicBlock` ist dabei eine Sammlung von sequentiell ausgeführten Instruktionen/Ausdrücken, die der Compiler während der Optimierungsphase aus dem Quellcode konstruiert. Diese bestehen in der Standarddefinition aus folgenden Teilen: Einer Menge von Ausdrücken, die selbst wiederum rekursiv Ausdrücke enthalten können, einer finalen Instruktion, welche meistens den weiteren Kontrollfluss angibt (`if-then-else`, `while`, `for`, ...) und einer Liste von Vorgänger- und Nachfolgerblöcken.

Die Verteilung von Ausdrucken auf Blöcke erfolgt dabei nach vorgegebenen Regeln. Betrachtet man einen `BasicBlock` als eine Menge solcher Ausdrücke ($B_{ID} = \{E_1, E_2, ...\}$, wobei ID die ID des Blocks darstellt), müssen alle vorhandenen Ausdrucksmengen $B(T)$ innerhalb einer Translation Unit T (zu analysierenden Datei) disjunkt sein: $\forall B_{ID1}, B_{ID2} \in B(T) : (ID1 \neq ID2) \Rightarrow B_{ID1} \cap B_{ID2} = \emptyset$. Das heißt, dass ein Ausdruck nur in einem `BasicBlock` vorkommen darf. Blockeinträge dürfen jedoch auch Ausdrücke anderer Blöcke referenzieren. Da die Instruktionen einen aufeinanderfolgendenden Ablauf garantieren müssen, müssen Sprünge (durch Schleifen, Abfragen, logische Operatoren oder ähnliche Konstrukte) speziell behandelt werden.

Hierfür ergeben sich die folgenden, grundlegenden Vorschriften/Regeln:

1. Wird ein Sprungziel gefunden, wird ein `BasicBlock` erstellt und das Sprungziel als Instruktion in dessen Menge hinzugefügt.
2. Wird ein Sprungbefehl gefunden, wird dieser als finale Instruktion (Terminator) im aktuellen Block gesetzt. Der unmittelbar darauffolgende Ausdruck muss dann als Sprungziel betrachtet werden ($\rightarrow$ vorheriger Punkt).
3. Alle anderen Ausdrücke werden direkt der Menge des aktuellen Blocks hinzugefügt, wobei dieser neu erstellt wird, falls er noch nicht existiert oder der aktuelle Block einen anderen Gültigkeitsbereich repräsentiert.

(vgl. [16], Basic Blocks)

Zusätzlich können bei gewissen Konstrukten (bspw. `do-while`-Schleifen) noch diverse Sonderfälle auftreten. Etwa, dass ein zusätzlicher, leerer Block zwischen Sprungbefehl und Sprungziel eingefügt wird. Über genaue Verbindungen zwischen den Blöcken wird hier also keine Aussage getätigt.

Abbildung 6.1 zeigt einen beispielhaften Aufbau nach Abarbeitung dieser Regeln.

Die nummerierten Einträge sind dabei die Elemente der Instruktionsmenge und `T` repräsentiert den Terminator.

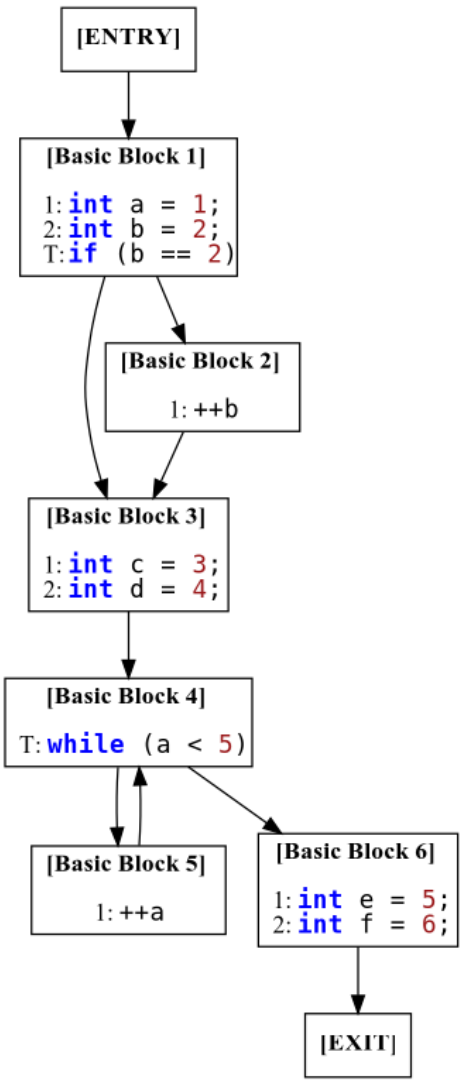


Abbildung 6.1: Basic Block-Beispiel

Der zugehörige Quellcode ist wie folgt definiert:

```
1 int a = 1;
2 int b = 2;
3
4 if (b == 2) {
5   ++b;
6 }
7
8 int c = 3;
```

```
9  int d = 4;
10
11 while (a < 5) {
12   ++a;
13 }
14
15 int e = 5;
16 int f = 6;
```

Listing 6.3: Quellcode für die BasicBlocks

- Die initialen Variablenzuweisungen befinden sich nach Regel 3 in Block 1.
- Zeile 4 definiert mit der If-Abfrage einen Sprungbefehl, da bei positiver Bedingung der Inhalt dieser ausgeführt wird und hierfür zu diesem Bereich im Programm gesprungen werden muss. Dieser Ausdruck wird somit gemäß der ersten Regel als Terminator innerhalb von Block 1 gesetzt.
- Zeile 6 ist dann ein entsprechendes Sprungziel und erhält somit durch Regel 2 einen Platz in der Instruktionsmenge eines neuen Blocks (im Beispiel Nummer 2).
- Die Zuweisungen in Zeilen 9 und 10 kämen gemäß Regel 3 in den aktuellen Block (2). Dieser bezieht sich jedoch auf einen anderen Gültigkeitsbereich (den Inhalt der If-Abfrage), der mit den Codeteilen um ihn herum nichts zu tun hat, sodass ein neuer Block erstellt wird (Block 3).
- Die while-Schleife in Zeile 12 ist gleichzeitig Sprungbefehl und -ziel. Hier greifen die Regeln 1 und 2 zusammen. Der Ausdruck wird wegen seiner Befehlseigenschaft als Terminator im aktuellen Block gesetzt, welcher aufgrund der Sprungzieleigenschaft ein neuer Block (4) sein muss.
- Zeile 14 wird analog wie Zeile 6 behandelt.
- Die letzten beiden Zuweisungen erhalten ihren eigenen Block aus analogen Gründen zu denen in Zeile 9 und 10.

6.1.2 Basic-Block-Implementierung in clang und SCA

Auch in clang befindet sich eine entsprechende Implementierung, die diesen Vorgaben folgt. Die entsprechende Klasse heißt dabei `CFGBlock`. Hierbei sei noch angemerkt, dass die Liste von Vorgängerblöcken dabei beliebig ist, während die Nachfolger einer genauen Ordnung folgen. (vgl. [17], Detailed Description) Diese ist in Abbildung 6.2 zu sehen.

Terminator	Successor Ordering
if	Then Block; Else Block
? operator	LHS expression; RHS expression
logical and/or	expression that consumes the op, RHS
vbase inits	already handled by the most derived class; not yet

Abbildung 6.2: Ordnung für Nachfolgerblöcke eines `CFGBlock`s

Nun ist in Zeile 128 von Codeausschnitt 6.1 zu sehen, dass aus jeder `CFGBlock`-Instanz des kompletten Kontrollflussgraphen ein eigenes `BasicBlock`-Objekt erstellt wird, das mit clang nichts zu tun hat. SCA2AMALTHEA hat eine spezielle, abgespeckte Datenstruktur eingeführt, die lediglich die Block-ID sowie Vorgänger- und Nachfolgerinformationen speichert. Die in 6.1 genutzte, umgekehrte Iteration mittels des `CFG::const_reverse_iterator` ist dabei sinnvoll, da clang die `BasicBlock`-Struktur von unten nach oben erzeugt (vgl. beispielhafte Ausgabe in [18]) und man in der praktischen Auswertung jedoch bevorzugt mit der chronologischen Reihenfolge (erste Instruktion zuerst) arbeiten möchte. Die Vorgänger- und Nachfolgerstruktur bleibt trotz dieser spezifischen Implementierung in clang jedoch konsistent und korrekt, sodass die regulären Iteratoren für diese genutzt werden können, um die passenden Informationen zu erhalten. Warum clang diese Anordnung für die Blöcke wählt, konnte im Rahmen dieser Arbeit nicht genauer erforscht werden.

Die relevanten Attribute der Datenstruktur für Funktionen sind im folgenden Codeausschnitt augelistet.

```
28 class FunctionAndCallees {
29 private:
30   FunctionEntry* functionID;
31   std::list<CallEntry*> callees;
32   std::list<GlobalVariableAccess*> globalVariableAccesses;
33   std::list<BasicBlock>* basicBlockList;
34   std::list<std::list<unsigned int>*>* CFGPaths = nullptr;
35   std::list<std::list<unsigned int>*>* CFGCycles = nullptr;
36   // ...
```

Listing 6.4: FunctionAndCallees.h

Die Reihenfolge der einzelnen BasicBlocks in der Liste `basicBlockList` ergibt sich dann aus Codebeispiel 6.1. Es zeigt sich, dass die Liste immer mit dem zur chronologisch ersten Instruktion gehörigen Block beginnt. Dies ist die einzig wichtige Information, um im weiteren Verlauf die `CFGPaths` und `CFGCycles` zu erzeugen, da alle benötigten Nachfolgerinformationen in diesem Block mitgespeichert werden und dadurch ausgehend von diesem rekursiv traversiert werden können.

6.2 Ermittlung von Pfaden und Zyklen

Es ist in der zuständigen Methode `findAllPaths` (vgl. Codeausschnitt 6.5) ersichtlich, dass aufgrund der Rekursion eine Tiefensuche im Kontrollflussgraphen stattfindet. Ein Pfad wird solange nach unten verfolgt (Rekursion in Zeile 235), bis ein Blatt auftritt (Abfrage in Zeile 237, ob keine Nachfolger mehr vorhanden sind) oder andernfalls ein Knoten gefunden wird, der bereits in der durchsuchten Menge vorhanden ist (alternativer Zweig in Zeile 241). Die Methode war im aktuellen Zustand nicht optimal implementiert bzw. enthielt Stellen, die keinen Sinn ergaben, wodurch sie im Rahmen der Arbeit nochmal überarbeitet wurde. Das Codebeispiel, auf das sich die folgenden Informationen beziehen, ist somit nicht das der Originalfassung im Plugin.
Übergeben wird eine Liste an `nodes`, die alle Elemente des aktuell traversierten Pfades in chronologischer Reihenfolge enthält, der aktuelle `BasicBlock`, an dem sich der Durchlauf gerade befindet, die Menge aller verfügbaren `BasicBlock`-Objekte im Kontrollflussgraphen sowie das aktuell analysierte Funktionsobjekt (`functionAndCallees`), dessen Inhalt der Graph repräsentiert. Der Algorithmus arbeitet dann wie folgt:

1. Prüfe, ob die Menge der traversierten Knoten leer ist (erste Bedingung) oder, falls dies nicht der Fall ist, der aktuell analysierte Knoten noch nicht in der Menge der traversierten Knoten enthalten ist

2. Falls dies wahr ist:
   a) Füge den aktuellen Knoten (über seine ID) zur traversierten Menge hinzu
   b) Iteriere über die IDs aller Nachfolgerknoten des aktuellen Knotens
      i. Ermittle aus der Liste aller verfügbaren Blöcke den Block, dessen ID mit der aktuellen Nachfolger-ID übereinstimmt
      ii. Rufe die Methode rekursiv mit einer Kopie der neuen, aktuellen Liste traversierter Knoten und dem ermittelten Nachfolgerblock (als aktueller Knoten) auf
   c) Prüfe, ob die Liste der Nachfolgerknoten leer ist (d.h. es wurde ein Blatt gefunden)
   d) Falls dies wahr ist:
      i. Erzeuge eine neue Liste, füge Ihr alle gefundenen Pfadelemente hinzu und hänge sie an die Liste der `CFGPaths` der aktuellen Funktion an

3. Falls dies nicht wahr ist und die Menge der bereits traversierten Knoten nicht leer ist und der aktuelle Knoten bereits abgelaufen wurde/in der Menge der traversierten Knoten vorhanden ist
   a) Erzeuge eine neue Liste, füge Ihr alle gefundenen Pfadelemente hinzu und hänge sie an die Liste der `CFGCycles` der aktuellen Funktion an

Würde man statt der reinen Identifikationsnummern die ganzen `BasicBlock`-Objekte (bzw. einen Zeiger auf diese) innerhalb eines `BasicBlock`s speichern, könnte man die Iteration mit anschließender Prüfung in den Zeilen 229 und 230 weglassen und bräuchte zusätzlich nicht die Menge aller verfügbaren Blöcke an die Methode übergeben. Aus Gründen der Speichereffizienz wurde dies scheinbar von Bosch jedoch auf diese Weise gelöst.

```
223 void findAllPaths(list<unsigned int>* nodes, BasicBlock*
      newNode, list<BasicBlock>* completeList,
      FunctionAndCallees* functionAndCallees) {
224   if (nodes->size() == 0 || (nodes->size() > 0 && std::find
      (nodes->begin(), nodes->end(), newNode->getID()) ==
      nodes->end())) {
225     //new node is not already in the list or the first need
      to be inserted
226     nodes->push_back(newNode->getID());
227     int i = 0;
228     for (unsigned int succ : *newNode->getSuccs()) {
229       for (BasicBlock bb : *completeList) {
230         if (bb.getID() == succ){
231           i++;
232           //cout << i << endl;
233           findAllPaths(copyList(nodes), &bb, completeList,
      functionAndCallees);
234         }
235       }
236     }
237     if (newNode->getSuccs()->size() == 0) {
238       functionAndCallees->addCFGPath(copyList(nodes));
239     }
240   }
241   else if (nodes->size() > 0 && std::find(nodes->begin(),
      nodes->end(), newNode->getID()) != nodes->end()) {
242     list<unsigned int>* myCycle = copyList(nodes);
243     myCycle->push_back(newNode->getID());
244     functionAndCallees->addCFGCycle(myCycle);
245   }
246 }
```

Listing 6.5: Helpers.cpp (angepasst)

Betrachtet man nun Abbildung 6.1 und führt für diese Basic-Block-Struktur den eben gezeigten Algorithmus aus, ergeben sich die folgenden Aufrufe.
Hierbei ist `completeList = [BB1, BB2. BB3, BB4, BB5, BB6]`.

1. `nodes = [], newNode = BB1`

2. `nodes = [1], newNode = BB2` (nicht BB3, gemäß der Reihenfolge in Abbildung 6.2, wobei der `else`-Block wegfällt)

3. `nodes = [1, 2], newNode = BB3`

4. `nodes = [1, 2, 3], newNode = BB4`

5. `nodes = [1, 2, 3, 4], newNode = BB5`

6. `nodes = [1, 2, 3, 4, 5], newNode = BB4,`
   → `CFGCycles = [[1, 2, 3, 4, 5, 4]]`

7. `nodes = [1, 2, 3, 4], newNode = BB6`
   → `CFGPaths = [[1, 2, 3, 4, 6]]`

8. `nodes = [1], newNode = BB3`

9. `nodes = [1, 3], newNode = BB4`

10. `nodes = [1, 3, 4], newNode = BB5`

11. `nodes = [1, 3, 4, 5], newNode = BB4,`
    → `CFGCycles = [[1, 2, 3, 4, 5, 4], [1, 3, 4, 5, 4]]`

12. `nodes = [1, 3, 4], newNode = BB6`
    → `CFGPaths = [[1, 2, 3, 4, 6], [1, 3, 4, 6]]`

6.3 Idee für den Ansatz

Nun wäre eine Idee, diese Informationen zu nutzen und basierend auf diesen die XML-Struktur zu bauen. Bei der reinen Verwendung dieser Listenform ergeben sich jedoch Schwierigkeiten. Einmal müsste für die Analyse der `CFGCycle`- und `CFGPath`-Attribute ein komplexer Algorithmus mit vielen Iterationen entworfen werden. Das Problem bei dieser Struktur ist, dass (wie im vorherigen Unterkapitel erläutert) rekursive Informationen auf mehrere Listen verteilt abgespeichert werden. Eine entsprechende Rekonstruktion des Aufbaus und eine performante Suche von Wegen wird damit enorm erschwert und gestaltet sich ineffizient. Am Ende muss, wie bereits in Kapitel 5 beschrieben, eine passende Baumstruktur vorliegen, die traversiert werden kann. Diese muss somit entsprechend aus den bestehenden CFG-Daten erzeugt werden.

6.3.1 Angedachter Aufbau der Baumstruktur

Abbildung 6.3 zeigt einen Ausschnitt aus einer Basic-Block-Struktur mit einem `if-else`-Zweig. Die zugehörige Baumstruktur, die für die Generierung der XML-Inhalte als Basis dient, ist in Abbildung 6.4 zu sehen.

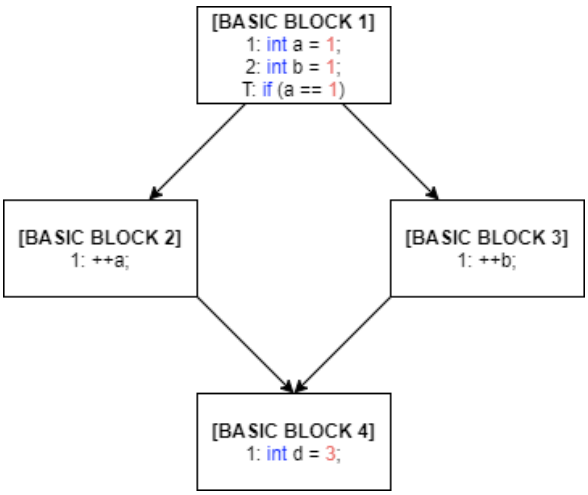


Abbildung 6.3: Basic Block-Beispiel mit `if-else`-Zweig

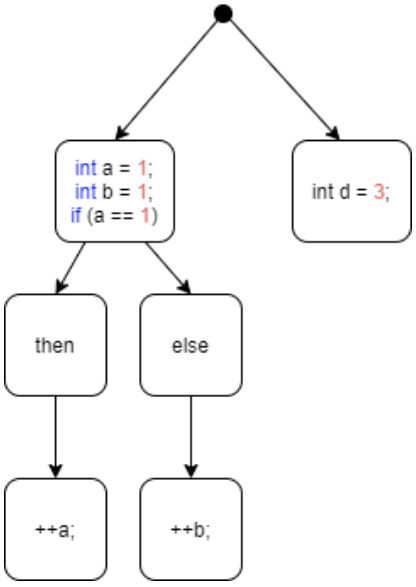


Abbildung 6.4: Zugehörige Baumstruktur

Der schwarze Punkt repräsentiert dabei den ENTRY-Block, der in Abbildung 6.3 oben weggelassen wurde. Es ist aus diesen ersichtlich, dass man die Basic-Block-Elemente auch direkt innerhalb der Baumstruktur integrieren kann, da diese nur Folgen von Ausrücken zusammenfassen und die Aufrufsreihenfolgen innerhalb dieser auch beim späteren Evaluieren konsistent bleiben müssen. Sie werden nur unterschiedlich angeordnet, um die rekursive Abarbeitung zu ermöglichen. Diese Baumstruktur ist dabei, auch wenn sie gemeinsame Eigenschaften aufweist, nicht ganz ähnlich zu der eines abstrakten Syntaxbaumes. Sie besitzt ein proprietäres Format, da die zugrundeliegende Basic-Block-Struktur mehrere Ausdrücke zusammenfassen kann, während ein AST diese komplett kapselt.

Ein Gültigkeitsbereich wird durch eine Ebene im Baum repräsentiert. Der ENTRY-Block stellt somit das Wurzelelement dar. Er hat Verbindungen zu allen Blöcken des Hauptausführungspfades, also allen Aufrufen im direkten Gültigkeitsbereich der Methode. Ausdrücke, die einen neuen, eigenen Anwendungsbereich öffnen, erhalten dann die jeweiligen Pfade als Nachfolger. Da ein `if-else` beispielsweise einen neuen Bereich eröffnet, muss deren jeweiliger Inhalt, unterteilt in einen `then`- und optional mehrere `else`-Pfade, eine Ebene tiefer hinzugefügt werden. Die `then` und `else`-Signalblöcke sind dabei leer. Beim späteren Durchlaufen werden daher innerhalb dieser keine Variablenzugriffe oder Funktionsaufrufe gefunden, sodass diese lediglich als Brücke zu den jeweiligen Nachfolgern dienen. Sie sind jedoch zwingend notwendig, da sonst bei verschachtelten `if`s nicht klar wäre, ob eine auf die innere Abfrage folgende Instruktion zum `else`-Teil des oberen `if`s gehört oder tatsächlich einen nachfolgenden Ausdruck im `then`-Teil der inneren Abfrage darstellt. Das Problem entsteht, da eine `if`-Abfrage immer die letzte Anweisung eines Blocks (der Terminator) ist und nachfolgende Instruktionen in einen neuen Block wandern. Die folgende Abbildung 6.5 zeigt einen solchen Fall auf.

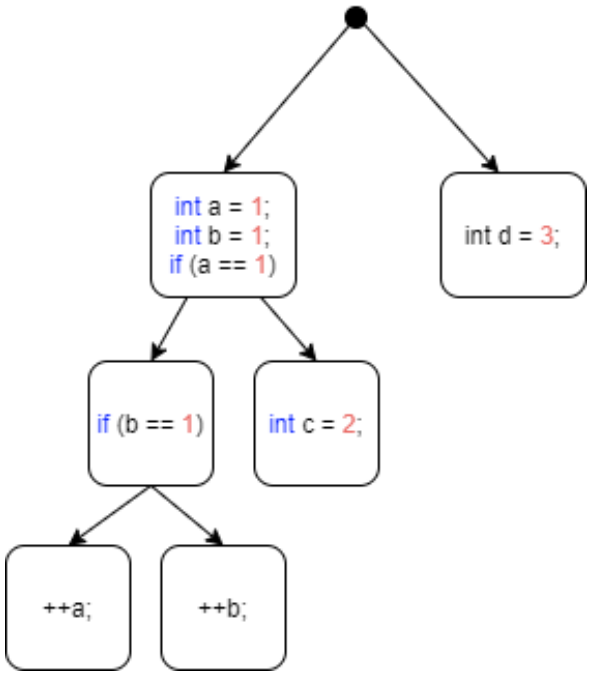


Abbildung 6.5: Nicht-eindeutige Baumstruktur

Dieser Baum könnte zwei verschiedene Codeausschnitte als Basis haben. Ein derartiger Fall darf jedoch nocht eintreten, da dann keine eindeutige Rekonstruktion und Erkennung des Ablaufs mehr möglich wäre. Eine Funktion, die einen Quellcode auf einen solchen Baum abbildet, muss immer bijektiv sein. Die beiden Codebeispiele für diesen Baum sind nachfolgend angegeben.

```
1  int a = 1;
2  int b = 1;
3  if (a == 1) {
4      if (b == 1) {
5          ++a;
6      }
7      else {
8          ++b;
9      }
10 }
11 else {
12     int c = 2;
13 }
```

Listing 6.6: Codeausschnitt 1

```
1  int a = 1;
2  int b = 1;
3  if (a == 1) {
4      if (b == 1) {
5          ++a;
6      }
7      else {
8          ++b;
9      }
10
11     int c = 2;
12 }
```

Listing 6.7: Codeausschnitt 2

Eine Unterscheidung zwischen Folgeausdruck und `else` (`if`)-Zweig wäre nicht mehr möglich. Den aus diesem Grund gewählten Ansatz kann man folglich auch für `switch` -Konstrukte verwenden, indem man eine beliebige Anzahl von `else`-Pfaden zulässt. Man könnte für diese zur besseren Abgrenzung von `if`-Abfragen auch einen neuen Brückenknoten `case` einführen. Da beide Kontrollstrukturen später jedoch mittels Mode Switches zusammengefasst werden und keine detaillierteren Informationen in diesen hinterlegt werden, ist dies für die Überführung gleichgültig.

Um die Basic-Block-Struktur in einen solchen Baum zu überführen, muss man unter anderem die einzelnen Pfade filtern und den entsprechenden Ebenen hinzufügen. Ebenso müssen Überlegungen für andere Kontrollstrukturen (`for`, `while`, ...) stattfinden, da auch diese adäquat überführt oder anderweitig berücksichtigt werden müssen. Hierfür muss jedoch zunächst der Aufbau der Basic-Block-Struktur bekannt sein.

6.3.2 Aufbau der Basic-Block-Struktur

Zunächst sollte die Basic-Block-Struktur daher induktiv definiert werden, da diese für die gewünschten Daten als Quelle dient und man somit jeweilige Ansätze für die Überführung besser nachvollziehen kann. Die Namen der entsprechenden Regeln sind dabei eigene Bezeichnungen im Rahmen dieser Arbeit, um diese im späteren Textverlauf kennzeichnen zu können.

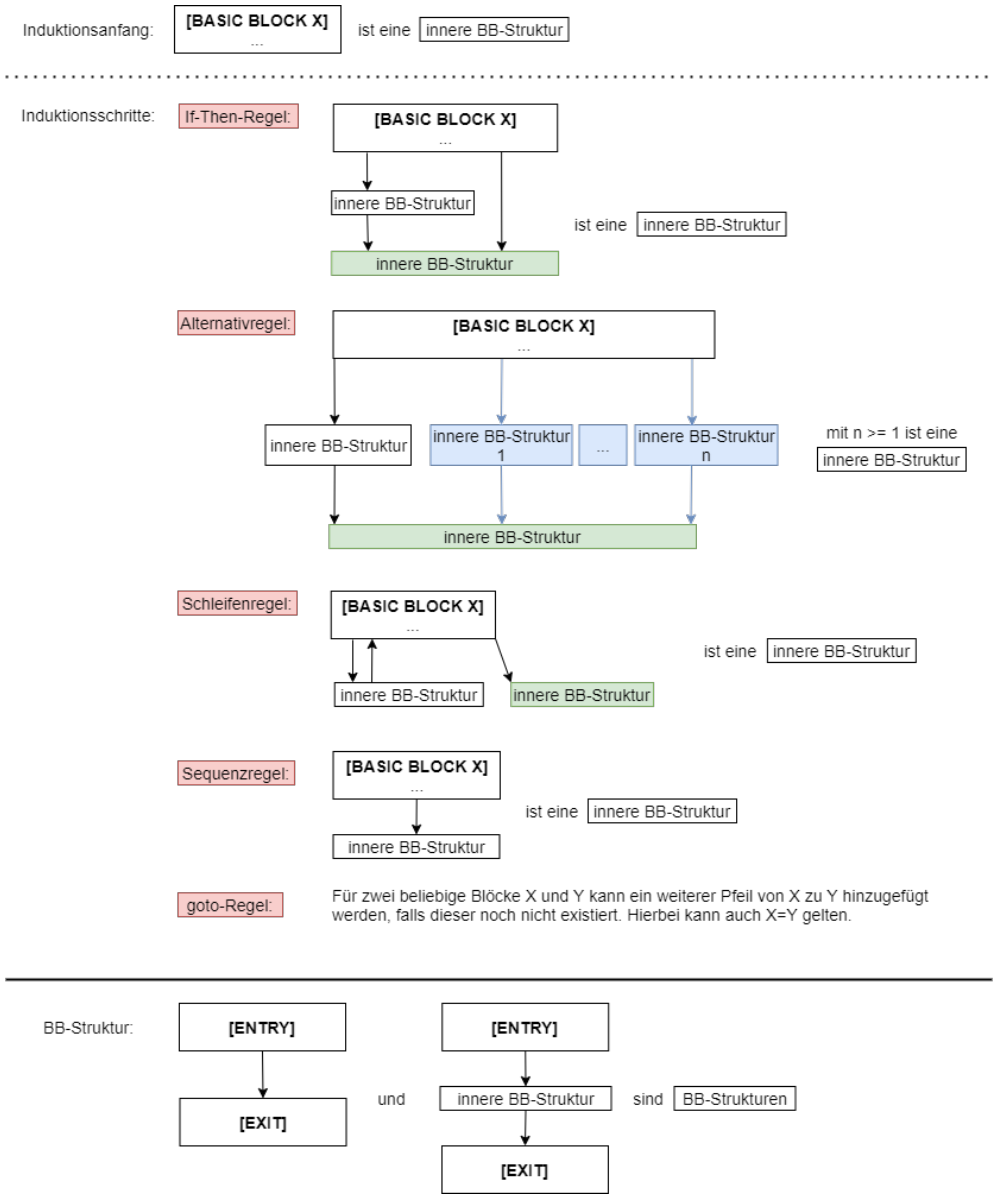


Abbildung 6.6: Induktivte Definition der Basic-Block-Struktur

Es erfolgt eine Aufteilung in die Basic-Block-Struktur selbst und deren Inhalt. Den Rahmen bilden gemäß [18] immer die ENTRY- und EXIT-Blöcke. Dies gilt auch, wenn eine Methode nie das Ausführungsende erreicht (etwa durch Endlosschleifen).

Diese beiden alleine bilden bereits eine gültige Struktur, wenn eine leere Methode vorliegt. Bei einer Methode mit Inhalt kommt dann die induktive Definition der inneren Struktur hinzu. Der Induktionsanfang ist dabei ein einzelner Block. Alle weiteren Verbindungen ergeben sich dann aus Anwendung der Induktionsschritte. Die gezeigte Existenz von ENTRY- und EXIT-Blöcken ist auch für deren Korrektheit im eingebetteten Kontext relevant, da mit diesen immer sichergestellt wird, dass sich am Ende einer Methodenausführung noch mindestens ein Block auf dem Hauptpfad befinden muss. Es kann nicht vorkommen, dass eine Verzweigung nie mehr zurückführt bzw. einfach endet. Die ausgehenden Pfeile eines inneren Blocks sind also *immer* anzubringen und müssen in einen Zielblock führen.

Dieser Aspekt ist auch für die grünen Blöcke von Relevanz, da diese optional sind und man somit denken könnte, dass bei Auslassen dieser die entsprechenden Pfeile ebenso wegfallen. Ist dies der Fall, muss dennoch zwingend eine Verbindung mit einem oder mehreren anderen, beliebigen Block/Blöcken der Struktur existieren. Es müssen jedoch nicht alle Pfeile in denselben Block führen. Diese können auch jeweils unterschiedliche Verbindungen aufweisen.

Die grünen Blöcke selbst existieren unter anderem, damit die einzelnen Pfade nicht zwingend zuerst in einem separaten Block zusammengeführt werden müssen. Dies ist notwendig, um beispielsweise die Darstellung von geschachtelten Alternativpfaden, die direkt in einen Zusammenführungsknoten eines höheren Geltungsbereiches (zum Beispiel in den EXIT-Block) münden, zu erlauben. Ein solcher Fall ist in Abbildung 6.7 dargestellt, da Block 4 und 5 nicht noch in einem vorherigen Knoten zusammengeführt werden, sondern direkt in Block 5 münden, welcher sich wieder auf dem Hauptpfad befindet.

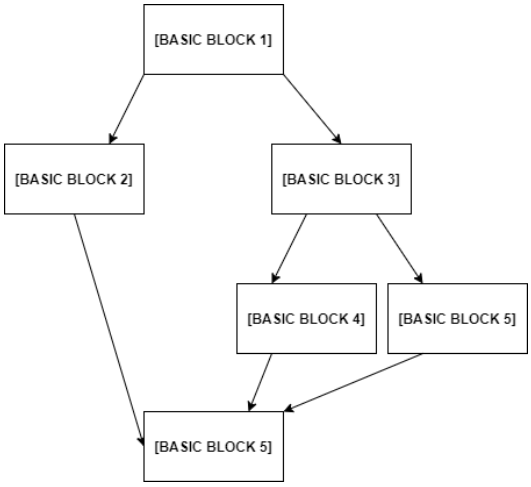


Abbildung 6.7: Geschachtelte Blöcke mit keinem direkten Zusammenführungsknoten

Ebenso werden somit `goto`-Befehle berücksichtigt, die Verbindungen zwischen einzelnen Ausführungspfaden erzeugen (vgl. Abbildung 6.14).

Diese horizontale Verbindung von Pfaden und das damit einhergehende Fehlen eines gemeinsamen Zusammenführungsblocks ist auch für `switch case`-Ausdrücke interessant, bei denen kein `break` verwendet wird.

Zunächst erhält jeder `case`-Block immer einen eigenen Ausführungspfad. Wenn bei jedem von diesen ein `break` eingesetzt wird, führen diese analog zu `if-else`-Konstrukten wieder in einen eigenen, gemeinsamen Nachfolgerblock. Ebenso gibt es eine direkte Verbindung vom initialen Block in diesen zusammenführenden. Dies ist in Abbildung 6.8 zu sehen.

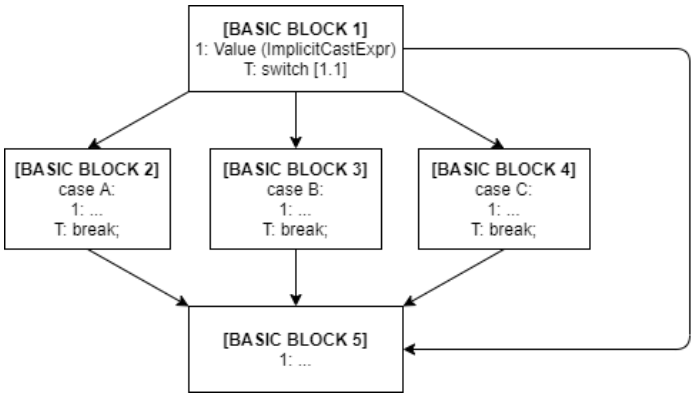


Abbildung 6.8: Repräsentation von `switch case`-Konstruktren mit Nutzung von `break`

Wird das `break` jedoch weggelassen, sind die einzelnen `case`s auch untereinander verbunden, da somit von Anweisung zu Anweisung gesprungen werden kann. Jeder Pfad führt somit am Ende in den letzten `case`-Block und ausgehend von diesem setzt sich dann die Methodenausführung fort. Die folgende Abbildung 6.9 zeigt diesen Fall.

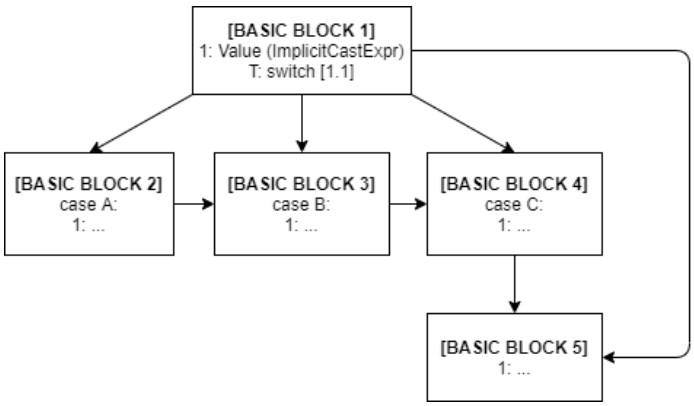


Abbildung 6.9: Repräsentation von `switch case`-Konstruktren bei fehlender Nutzung von `break`

Die **if-then-Regel** ist für `if`-Abfragen ohne `else`-Teil konzipiert, welche hier den einzigen Spezialfall darstellen. Die **Alternativregel** funktioniert analog, erlaubt jedoch die Verwendung von `else`-Pfaden. Die dort zu sehenden, blauen Elemente stellen dabei eine Menge von beliebig vielen Zwischenstrukturen dar. Bei If-Abfragen ist `n` hierbei immer 1. Ein praktischer Test gemäß der in [18] gezeigten Methodik hat gezeigt, dass `else if`-Vorkommen nicht als eigener Pfad angesehen werden, sondern als Unterbedingung in den `else`-Teil kommen bzw. dort geschachtelt werden (vgl. Abbildung 6.10).

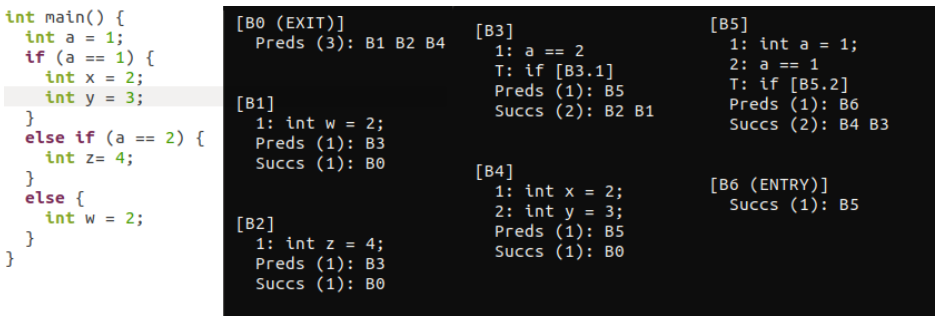


Abbildung 6.10: Repräsentation von `else if`-Zweigen (erzeugt mit dem Code aus [18])

`n` kann jedoch nicht einfach auf 1 beschränkt werden, da es, wie bereits erläutert, bei einem `switch case` mehrere Nachfolger ergibt. Die Regel ist somit korrekt. Bei `switch`-Konstrukten muss zusätzlich noch die `goto`-Regel angewandt werden.

Die **Alternativregel** ist außerdem für Ausdrücke mit binären Operatoren relevant, da diese auch in unterschiedliche Blöcke aufgeteilt werden. Existiert also bspw. eine `if`-Abfrage, die mehrere durch ein logisches `&&` oder || verbundene Bedingungen enthält, so werden diese nicht auf einmal, sondern einzeln ausgewertet. Praktisch bedeutet dies, dass eine einzelne Abfrage in mehrere Teilabfragen unterteilt wird, die je nach Operator dann zur nächsten Bedingung oder direkt in einen resultierenden Block (`then`-Block bei ||, `else`-Block bei `&&`) führen. Hierbei ist egal, wie viele Operatoren aneinandergereiht werden, da diese immer nach dem Teile-und-Herrsche-Prinzip auf zwei Bedingungen reduziert werden können. Dieselbe Logik gilt auch für alle anderen Deklarationen, in denen binäre Ausdrücke auftreten.

`do while`-Schleifen haben einen besonderen Aufbau, welcher in Abbildung 6.11 zu sehen ist. Hierbei enthält Block 1 den Inhalt des `do`-Teils und Block 2 lediglich die entsprechende Abfrage für den `while`-Teil. Es existiert zudem immer ein inhaltloser Bindeblock (Block 3), der nur wieder zurück zum ersten führt. Der rechte Pfad repräsentiert den restlichen Teil der Methode, der ausgeführt wird, wenn die Bedingung nicht länger erfüllt ist. Betrachtet man diesen Aufbau, so kann man folgern, dass diese ebenso durch die **Alternativregel** gebaut werden können, wobei kein eigener Zusammenführungsknoten vorhanden ist.

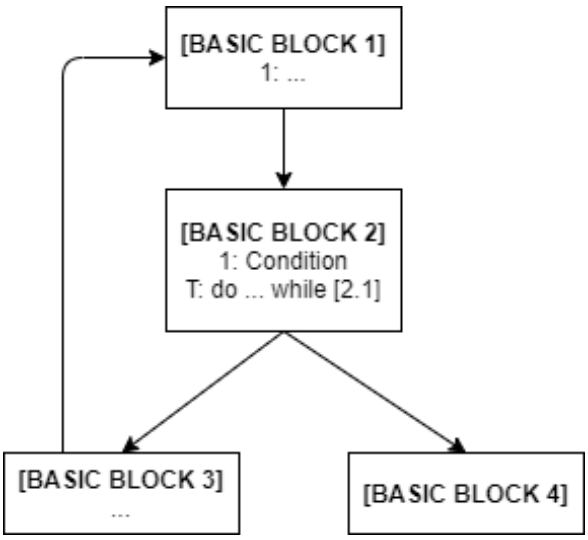


Abbildung 6.11: Basic Block-Darstellung einer `do while`-Schleife

`for`- und `while`-Schleifen werden durch die **Schleifenregel** repräsentiert. Der Block B, der den Schleifenausdruck als Terminator besitzt, hat dabei immer zwei Nachfolger. Einen, der den Inhalt der Schleife wiedergibt und einen, der zum weiteren Ausführungspfad der Methode führt. Der linke Nachfolger führt dabei zurück zu B und erzeugt damit den Zyklus. Bei verschachtelten Schleifen führt der rechte Pfeil einer inneren Schleife dabei immer zur darüberliegenden zurück, wenn keine `goto`s verwendet werden (vgl. oranger Pfeil in Abbildung 6.12).

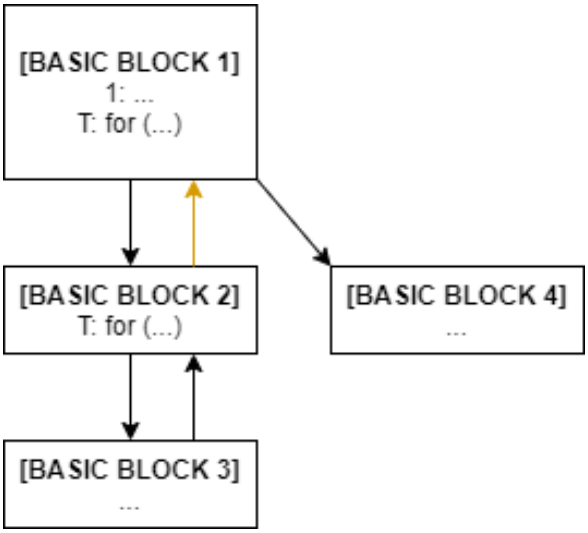


Abbildung 6.12: Verschachtelte Schleifen in der Basic-Block-Struktur

Sequenzregel und **goto-Regel** können (unter anderem) für `goto`-Befehle genutzt werden. Mit ersterer ist es möglich, direkt aufeinanderfolgende Sequenzen, die über ein `goto` verbunden sind, auszudrücken. Ein solcher Fall ist in Abbildung 6.13 zu sehen. Praktisch macht dieser wenig Sinn. Formal müssen jedoch alle denkbaren Fälle abgedeckt sein, damit die Regeln vollständig sind.

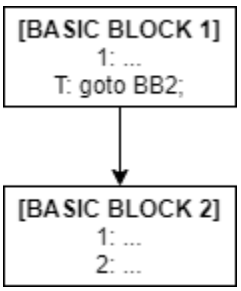


Abbildung 6.13: Direkt aufeinanderfolgende Blöcke mit `goto`

Die letzte Regel erlaubt es allgemein, beliebige Blöcke zu verbinden. Es ist insofern egal, wohin ein Sprung führt. Mit der Regel kann also auch die Verbindung zwischen `switch`- und Zusammenführungsblock bei `switch case`s hinzugefügt werden.

6.3.3 Vorbereitung der Basic-Block-Struktur

Bevor nun der eigentliche Blockbaum aus dieser Struktur erzeugt wird, müssen noch ein paar Vorkehrungen getroffen werden. Da beispielsweise Zyklen für die Ausarbeitung irrelevant sind, sollen diese zunächst entfernt werden. Ebenso muss eine Überlegung bezüglich der Behandlung von `goto`-Befehlen und Binäroperatoren erfolgen.

Erkennung von Zyklen Die Frage ist, wie man Zyklen in der Blockstruktur algorithmisch erkennt. Ein Ansatz wäre, beim Traversieren eine Liste der bereits besuchten Blöcke zu führen und bei erneutem Antreffen eines Blocks die entsprechende Kante zu ignorieren. Diese Methodik hat jedoch zwei Probleme. Zum Einen kann nicht zwischen einem Zusammenführungsknoten und einem Knoten mit Zyklus unterschieden werden, da ersterer ebenso einen Eingangsgrad > 1 besitzt und somit auch beim Ablaufen aller Pfade mehrmals besucht werden würde. Zweitens ist die Wahl der notwendigen Kanten stark vom Verlauf des Graphen abhängig, da unter Umständen eine nichtrelevante Kante zuerst gefunden wird und die eigentlich wichtige dann im weiteren Schritt ignoriert wird. Diese Informationen können also beim reinen Traversieren mit einem passenden Verfahren wie Breiten- oder Tiefensuche nicht herausgefiltert werden und müssen aus anderer Quelle bezogen werden. Hier kommen die von Bosch bereits angedachten `CFGCycle`-Listen wieder ins Spiel. Da zur Determinierung dieser alle Pfade einzeln durchlaufen werden, können Zyklen auch eindeutig bestimmt werden. Geht man alle vorhandenen Elemente in dieser Liste durch, kann man jeweils durch Extrahierung der letzten beiden Block-IDs die Kante finden, die den Zyklus verursacht und

entsprechend entfernen. Hierzu kann etwa eine `map` definiert werden, deren Schlüssel die Quellblöcke sind. Beim Traversieren kann dann auf Existenz der aktuellen Block-ID in der `map` geprüft werden und mit dem jeweiligen Wert der nächste Knoten, der zusammen mit dem aktuellen einen Zyklus erzeugt, ausfindig gemacht werden.

Erkennung von `goto`-Befehlen `goto`-Ausdrücke können praktisch ein großes Problem darstellen, da sie den chronologischen Ablauf einer Methode vollständig abändern können. Abbildung 6.14 zeigt die Nutzung von `goto` zwischen zwei alternativen Ausführungspfaden.

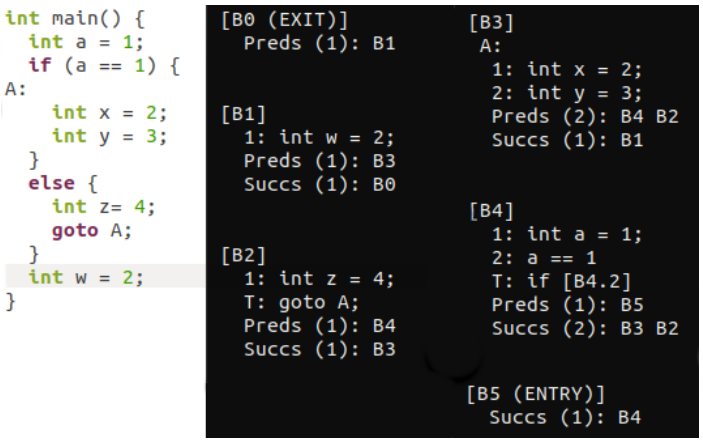


Abbildung 6.14: Nutzung von `goto` in einem parallelen Ausführungspfad

In diesem Fall werden beide Pfade jeweils wieder in einem Knoten/Block zusammengeführt, wodurch hier eine Erkennung der Switch-Grenzen noch möglich ist. Anders verhält es sich jedoch bei folgendem Beispiel:

```
1  int a = 4;
2  int b = 2;
3  if (a > b) {
4      foo();
5      goto D;
6  }
7  else {
8      bar();
9      return b;
10 }
11 int c = 5;
12 D:
13 int d = 4;
```

Listing 6.8: Nicht erkennbare Zusammenführung der Ausführungspfade

Für die zum Code zugehörige, in Abbildung 6.15 gezeigte Struktur ermittelt der (im nächsten Unterkapitel vorgestellte) Algorithmus, dass die Zusammenführung in Block 5 erfolgt. Ersichtlich ist aber, dass der `if-else`-Zweig eigentlich schon in Block 4 (darüber) endet.

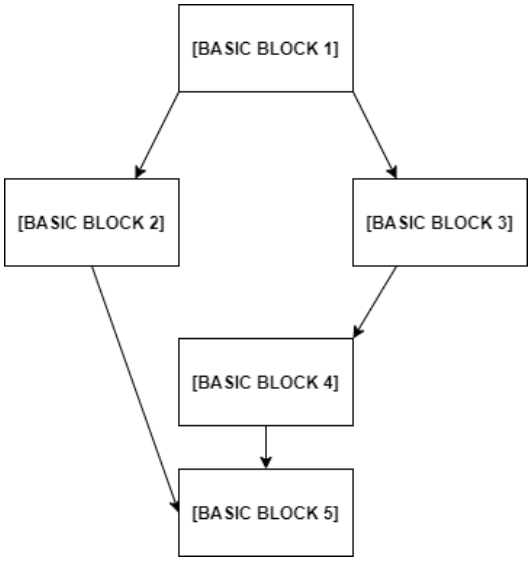


Abbildung 6.15: Problematik bei der Erkennung des richtigen Zusammenführungsblocks bei Verwendung von `goto`

Tatsächlich macht es im konkreten Fall semantisch keinen Unterschied, ob man den Inhalt von Block 4 auf dem Hauptpfad oder im `else`-Zweig ausführt, da er vom `then` -Teil ohnehin übersprungen wird. Dies gilt auch, wenn ein `else if` vorhanden ist, da dieses im `else`-Zweig geschachtelt wird und somit auf die äußere Umgebung keinen Einfluss nimmt. Die Deklaration selbst wäre dann jedoch nicht korrekt wiedergegeben, was im CallGraph nicht geschehen sollte.

Ein Ansatz für die Erkennung wäre bereits beim Iterieren über die `clang::CFGBlock` -Instanzen, die Blöcke, die einen `goto`-Ausdruck als Terminator besitzen, entsprechend zu markieren. Hierfür wäre also ein weiteres Attribut `ends_with_goto` in der `BasicBlock`-Klasse notwendig. Beim Traversieren könnte dieses dann geprüft und die nachfolgende Kante entfernt/ignoriert werden.

Ersetzung durch passende Kanten

Somit bleibt noch die Frage, wie die gelöschten Kanten ersetzt werden müssen. Dies ist in jedem Fall notwendig, da die Struktur sonst gemäß der induktiven Definition inkorrekt wäre. Hierunter fällt beispielsweise der in Abbildung 6.15 gezeigte Fall. Dasselbe gilt für die `while`-Schleife in Abbildung 6.1, da Block 5 nach Entfernung des Zyklus' keinen Nachfolger mehr hat und dann eigentlich auf Block 6 zeigen sollte.

`for`- und `while`-Schleifen Da der Aufbau dieser in der Struktur konsistent ist, kann man hier eine spezifische Methode verfolgen. Ignoriert man die Wiederholungen der Schleife und stellt sich vor, deren Inhalt würde nur einmal ausgeführt werden, könnte deren Fragment als gewöhnliche, alternative Ausführung angesehen werden, wobei ein `if` ohne `else` vorliegt. Dies gilt analog für `for`-Schleifen. Die Frage, die sich hinter diesem Ansatz versteckt, ist, ob man die in den Schleifen vorhandenen Bedingungen auch als Mode-Switch in die XML-Struktur aufnehmen möchte. Rein deklarativ gesehen handelt es sich dabei nicht um solche, weshalb sie auch in der Ausarbeitung nicht auf diese Weise behandelt werden. Stattdessen ist es sinnvoller, wenn man deren Inhalt als reguläre Sequenz von Funktionsaufrufen und Variablenzugriffen ansieht, die sich in keinerlei Kontrollstruktur befinden. In der originalen Fassung des Plugins ist dies auch so gelöst. Für die Repräsentation in der Basic-Block-Struktur bedeutet dies, dass der rechte Nachfolger des `for`- bzw. `while`-Blocks SB direkt auf jeden Block des linken Pfades, der zu SB zurückführt, folgen muss. Diese betroffenen Blöcke können dabei über die zuvor beschriebene `map` während der Traversierung ausfindig gemacht werden. Um diesen Ansatz anwenden zu können, muss noch in Erfahrung gebracht werden, wann eine echte Schleife gemäß der Schleifenregel in 6.6 vorliegt. Da `goto`-Befehle ausgeschlossen wurden, liegen diese vor, wenn der letzte Block eines gefundenen Zyklus' (entspricht SB) genau zwei Nachfolger hat.

`do while`-Schleifen Wenn man das Ziel verfolgt, mit dem Inhalt von `do while`-Schleifen analog wie bei `for`- und `while`-Schleifen zu verfahren, können diese relativ einfach erkannt und verarbeitet werden, indem man den Brückenblock (Block 3 in Abbildung 6.11) ignoriert/weglässt, da die entsprechende Zykluskante von diesem ausgeht. Entfernt man diesen mit seinem ein- und ausgehenden Pfeil, ist die Behandlung bereits abgeschlossen, da dieser leer ist und die chronologische Reihenfolge des Codes nicht beeinflusst wird.

`goto`-Befehle Die Behandlung von `goto`-Befehlen gestaltet sich schwierig, da hier keine festen Muster existieren. Um alle möglichen Fälle abzudecken, müsste man einen komplexen Algorithmus entwerfen, der alle Zusammenhänge prüft und passend auf diese reagieren kann. Dies würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund soll ab nun die Voraussetzung gelten, dass keine `goto`s im Code auftreten. Die Praxis zeigt, dass diese so gut wie immer äquivalent durch Nutzung anderer Kontrollstrukturen ausgedrückt werden können. Bereits hier wird jedoch deutlich, warum sich der Ansatz mit den Kontrollflussgraphen als ungeeignet erweist.

Binäre Ausdrücke Dasselbe gilt auch für binäre Ausdrücke. Da hier beliebige, komplexe Aneinanderreihungen von Ausdrücken möglich sind und diese auf jeweils unterschiedliche Blöcke verteilt würden, müsste man praktisch auch noch alle Speicheradressen der jeweiligen Blöcke bzw. Kindelemente einer Instruktion vergleichen, um diese über die verschiedenen Blöcke hinweg genau identifizieren und die Struktur somit nachvollziehen zu können.

Dies würde erfordern, dass die `BasicBlock`-Struktur um weitere Informationen erweitert wird und man im Endeffekt wieder auf die Informationen des bestehenden AST zurückgreifen müsste. Außerdem ist eine direkte Unterschiedung zwischen diesen und regulären `if-else`-Konstrukten nicht ohne Weiteres möglich, was auch hier noch weitere Flags notwendig machen würde. An dieser Stelle zeigt sich, dass zu viele Edge Cases entstehen, die niemals sauber abgedeckt werden können. Das Problem ist dann, dass nur ein einzelner Ausnahmefall, der nicht berücksichtigt wurde, die gesamte Struktur zunichte machen würde.

6.3.4 Überführung in die Baumstruktur

Würde man nun dennoch davon ausgehen, dass eine solche Präparation der bestehenden Struktur vollständig und mit moderatem Aufwand möglich ist, so könnte man sich einen passenden Überführungsalgorithmus überlegen. Dazu muss zunächst klar sein, wie das zum Baum zugehörige Modell aussieht. Da die Baumstruktur über die einzelnen Knoten definiert wird, besteht keine Notwendigkeit diese komplett abzuspeichern. Stattdessen reicht es aus, einen Zeiger auf das Wurzelelement zu erstellen und ausgehend von diesem dann den Baum traversieren zu können, wenn jeder Knoten Informationen zu seinen Nachfolgern speichert. Dies kann in einer Liste (`list< BlockTreeElement*>`) geschehen. Zusätzlich zu dieser muss noch der entsprechende Inhalt, welcher über einen Basic Block repräsentiert wird, gespeichert werden. Dazu kann man einen Zeiger auf die jeweilige `BasicBlock`-Instanz hinterlegen.

Überführungsalgorithmus Für die Überführung in die Baumstruktur sei eine Funktion `buildBlockTree` definiert, die einen Startblock (`source`), einen Endblock (`final`) und einen Zeiger auf ein `BlockTreeElement`-Objekt (`currentNode`), welches den aktuellen Knoten im Baum repräsentiert, erhält. Zusätzlich wird eine weitere Instanz der `BlockTreeElement`-Klasse (`predecessorNode`), die den Vorgängerknoten des aktuellen Knotens darstellt, übergeben. Dieser ist notwendig, um den Zugriff auf die zugehörige Baumebene des aktuellen Knotens zu erlauben, da deren Knoten von jenem Vorgänger ausgehen. Der finale Block ist bei Alternativpfaden wichtig, um den lokalen Arbeitsbereich des Algorithmus' dann beim Zusammenführungsknoten zu begrenzen und somit zu verhindern, dass die komplette Struktur bis zum Ende als Pfad im Baum gespeichert wird.

Der erste Aufruf erfolgt mit dem ENTRY-Block als `source` und dem EXIT-Block als `final`. Der aktuelle und der initiale Vorgängerknoten sind jeweils das leere Wurzelelement (d.h. eine `BlockTreeNode`-Instanz ohne Inhalt), da dieser der einzige in der Baumstruktur ist. Der folgende Algorithmus, der ausgehend von diesem Stand dann abgearbeitet werden kann, funktioniert nach dem Teile- und Herrsche-Prinzip und baut nach und nach den Blockbaum auf.

1. Enthält der momentane Basic Block `source` nur genau einen Nachfolger N und gilt $N \neq$ `final`, dann:
   a) Füge N über ein neues `BlockTreeNode`-Objekt BT dem Vorgängerknoten `predecessorNode` als Nachfolger hinzu.
   b) Rufe die Funktion rekursiv auf, wobei `source` auf den Nachfolger von BB, `currentNode` auf BT und `predecessorNode` auf `currentNode` gesetzt wird. Der Wert für `final` wird aus dem aktuellen Eingabeparameter übernommen.

2. Hat der Basic Block `source` $n \geq 2$ Nachfolger N_i (mit $i \in [2, n] \cap \mathbb{N}$), deutet dies auf mehrere Alternativpfade hin. Dann:
   a) Markiere den aktuellen Knoten `currentNode` als Gabelungsknoten.
   b) Finde den zusammenführenden Basic Block ZB, der die Gabelung wieder aufhebt, damit klar ist, welche Blöcke in den Switch gehören.
   c) Ist der gefundene Block ZB selbst ein Nachfolgerelement von `source`, so handelt es sich um ein `switch case` ohne `break` oder ein `if` ohne `else`. Setze dann den zusammenführenden Block auf den letzten Nachfolger von `source` und dekrementiere n um 1. *(vgl. Abbildung 6.9 beim* `switch`*; bei* `if` *ohne* `else` *bleibt der Block dadurch gleich und der Schritt korrekt)*
   d) Iteriere über alle Pfade (d.h. n mal) und mache für den jeweils aktuellen Pfad folgendes:
      i. Ermittle den ersten Block PB dieses Pfades. Ist $PB =$ `final`, überspringe diese Iteration.
      ii. Hänge einen neuen Brückenknoten BK als Nachfolger an den aktuellen Knoten `currentNode` im Baum an.
      iii. Füge PB über einen neuen Baumknoten PK dem Brückenknoten BK als Nachfolger hinzu.
      iv. Rufe die Funktion rekursiv auf, wobei `source` auf den ersten Block in diesem Pfad und `final` auf ZB gesetzt wird. `currentNode` erhält als Wert PK und `predecessorNode` wird mit dem Brückenknoten dieses Pfades initialisiert.
   e) Ist $ZB =$ `final`, gehört der Zusammenführungsblock zu einem höheren Geltungsbereich, der die restlichen Aufgaben übernehmen muss. Beende in diesem Fall daher die Methodenausführung.
   f) Baue einen neuen Baumknoten ZK für den Zusammenführungsblock ZB und füge diesen dem `currentNode` als Nachfolger hinzu.
   g) Rufe die Funktion rekursiv auf, wobei `source` auf ZB und `currentNode` auf ZK gesetzt wird. `final` wird aus dem Eingabeparameter übernommen und `predecessorNode` auf `currentNode` gesetzt.

Zusammenführungsblock ermitteln Schritt 2.b ist nicht so trivial, wie man es eventuell vermuten würde. Um diesen zu erkennen, müssen zwangsläufig alle vorhandenen Pfade, die an der Gabelung entstehen, durchlaufen werden, um deren ersten gemeinsamen Nachfolger zu finden. Dieser ist dann der gesuchte Block. Hierzu kann die vorhandene Idee der `CFGPath`-Liste genutzt werden. Die Verwendung der bestehenden Angaben zu dieser ist dabei jedoch nicht mehr möglich, da die Struktur zuvor angepasst wurde und die beiden somit nicht mehr übereinstimmen. Entsprechend muss man mit der neuen Basic-Block-Struktur zunächst eine neue Pfadermittlung initiieren. Dazu kann erneut die Methode `findAllPaths` der Klasse `Helper` angewandt werden. Als Funktionsmodell (Parameter `functionAndCallees`) dient dabei dann ein Platzhalterobjekt, das lediglich die resultierenden Daten halten soll und keinen weiteren, relevanten Einsatzzweck hat. Dieser Ansatz funktioniert auch bei `switch`-Konstrukten mit untereinander verbundenen `case`s, da diese dann immer in den letzten `case`-Block führen müssen, welcher somit den Zusammenführungsblock darstellt. Um den Arbeitsbereich des ermittelnden Algorithmus' auf die betroffenen Teile einzuschränken, muss zunächst klar sein, bei welchem Block zu beginnen ist. Ausgangspunkt ist dabei immer die ID des Blocks, der die `if`- bzw. `switch`-Anweisung als Terminator enthält. Da nicht zwingend alle Pfade der Basic-Block-Struktur von dieser Gabelung betroffen sind, kann bzw. muss man zunächst auch die hierfür relevanten Pfade bestimmen. Dazu iteriert man über alle Pfade und sucht innerhalb dieser nach der Startblock-ID. Ist diese existent, fügt man den aktuellen Pfad einer speziellen Auswahlliste hinzu und notiert sich zusätzlich (etwa in einer `map`) den zugehörigen Positionsindex des Blocks innerhalb von diesem. Das gesuchte Element wird sich dabei nie ganz am Ende befinden, da dort der EXIT-Block steht und jegliche Alternativpfade schon zuvor abgeschlossen sein müssen. Es kann also maximal zwei Elemente vor dem finalen Block liegen. Man nehme an, es seien im kompletten Graphen n Pfade vorhanden und diese haben maximal die Länge m (m Einträge in der Liste). Dann ergibt sich eine Laufzeit von $O(n * (m - 2))$, wobei m beliebige Größe relativ zu n annehmen kann. Somit ergeben sich nicht genauer spezifizierbare Laufzeiten. Ist m beispielsweise so groß wie n, hätte man bereits quadratische Komplexität zu erwarten.

Um dann den gemeinsamen Nachfolger zu finden, muss jeder Eintrag des ersten Pfades in jedem anderen Pfad gesucht werden, bis dieser überall vorkommt. Man kann sich dann sicher sein, dass es sich um den ersten gemeinsamen Block handelt, da die Pfadlisten in sich chronologisch geordnet sind. Die Problemlösung nutzt dabei Backtracking. Hierbei wird Schritt für Schritt versucht, aus jeweiligen Teillösungen eine Gesamtlösung zu konstruieren. Man überprüft solange alle Pfade, bis in jedem die gesuchte Block-ID auftritt. Andernfalls muss ein alternativer Weg ausprobiert werden. Das Backtracking erfolgt durch geschachtelte Iterationen, d.h. erneute Nutzung von Rekursion. An diesem Punkt verliert man ebenso die Kontrolle über die Effizienz der Implementierung, da im Voraus nicht entscheidbar ist, wie komplex der einzulesende Code ist. Dies gilt allerdings auch analog für bereits implementierte Methoden des Plugins, welche praktisch keine Probleme bereitet haben. Da die AMALTHEA-Modellgenerierung ohnehin ein Prozess ist, der basierend auf der Eingabe länger be-

nötigen kann (so wie dies auch in einer Hinweisnachricht bei Nutzung des Plugins beschrieben wird), soll dies kein Hindernis darstellen. Das Problem ist bei zwei Pfaden beispielsweise, dass für jeden einzelnen Eintrag in der ersten Liste, im schlechtesten Fall alle Einträge in der zweiten Liste durchsucht werden müssen. Bei noch mehr Ausführungspfaden schachtelt sich dies immer weiter, sodass sich für p Alternativpfade mit jeweils m_i Einträgen eine Komplexität von $O(\prod_{i=1}^{p} m_i)$ ergibt, wobei $p \leq n$ gilt. Für einfache `if-else`-Zweige ist p 2, also die Laufzeit für durchschnittlich konstant angenommene Pfadlängen m_i quadratisch. Schachtelt man aber nun noch weitere Abfragen in jeden der beiden Pfade, ist p bereits 4. Praktisch wird man einen bestimmten Grad an Schachtelung vermutlich alleine aus Gründen der Lesbarkeit nicht überschreiten. Gerade bei der Nutzung von `switch-case` kann die Laufzeit schnell ansteigen.

Diese Idee muss nun algorithmisch festgehalten werden. Der jeweilige Methodenaufruf soll dabei eine gefundene Block-ID zurückgeben, wenn diese im aktuellen und in allen weiteren Pfad vorhanden ist. Falls nicht, soll per Definition -1 zurückgegeben werden. Auf diese Weise kann man die Information, ob ein gemeinsamer Zusammenführungsknoten in allen beteiligten Pfaden vorhanden ist, durchreichen und auswerten.

Die entsprechende Funktion `determineMergeBlockID` erhält dabei als Parameter zunächst die zuvor ermittelten, zugehörigen Pfadlisten und deren Startindizes s_i ($i \in [1, \#Pfade \cap N]$), bei denen die Gabelung innerhalb dieser jeweils beginnt. Dieser geht dann zunächst nur vom ersten Pfad aus und nutzt für die weitere Prüfung eine weitere, interne Funktion, die dann die Rekursion durchführt. Diese erhält ebenso die Liste aller Pfade und die Startindizes. Zusätzlich werden ihr die aktuelle Pfad-ID (nachfolgend p) und die zu prüfende Block-ID übergeben.

`determineMergeBlockID` arbeitet dann wie folgt:

1. Iteriere über alle Elemente in der ersten Liste, beginnend beim zugehörigen Startindex s_1. b sei die aktuelle Block-ID der Iteration:
   a) Rufe die interne Funktion mit den Pfadlisten, den Startindizes, Pfad-ID (p) 2 und Block-ID b auf.

Die innere Funktion macht dann folgendes:

1. Iteriere über alle Elemente in der p-ten Liste, beginnend beim Startindex s_p.
   b sei die aktuelle Block-ID der Iteration:
   a) Ist b gleich der gesuchten Block-ID, rufe die innere Funktion erneut rekursiv mit den Pfadlisten, den Startindizes, Pfad-ID $p + 1$ und der zu prüfenden Block-ID auf. Gebe dann den von diesem Methodenaufruf zurückgegebenen Wert zurück und beende damit die Methodenausführung.
2. Gebe -1 zurück.

Auf diese Weise wird die gesuchte Block-ID an den ersten Aufruf durchgereicht. Damit sind alle notwendigen Daten vorhanden, um die Baumstruktur zu bauen.

6.4 Zusammenfassung und Bewertung

Der angedachte Ansatz ist an der Komplexität der Basic-Block-Struktur gescheitert, da es keine effiziente Möglichkeit gibt, alle auftretenden Kombinationen von Kontrollstrukturen und Ausdrücken zu erkennen und entsprechend zu behandeln. Wäre es möglich, die Vorbereitung ohne großen Mehraufwand oder nur mit vielen Einschränkungen umzusetzen, so könnte man diesen Ansatz jedoch auch wählen, um die Erweiterung im Rahmen dieser Arbeit umzusetzen. Diese ist jedoch dann nicht ausreichend flexibel und macht das Plugin praktisch unnutzbar. Würde man später auch noch Unterstützung für Schleifen implementieren wollen, so müsste man die gesamte Codelogik überarbeiten und viele weitere Prüfungen einbauen, um die verschiedenen Konstrukte passend zu erkennen und zu überführen. Dies würde dann die bestehende Problematik weiterführen und immer komplexer werden lassen, da bestehende Codeteile sich gegenseitig beeinflussen und blockieren würden. In jedem Fall ist daher die Traversierung des abstrakten Syntaxbaumes die sauberste und flexibelste Lösung, da diese eine eindeutige Identifizierung der jeweiligen Elemente zulässt und mit diesem bereits eine passende Baumstruktur vorliegt.

Nachfolgend finden sich Diff-Dateien, die die notwendigen Codeanpassungen für die Einrichtung des Plugins bzw. der SCA-Executable beschreiben.

```
1629c1629
<   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getLocStart());
---
>   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getBeginLoc());
1632c1632
<     return srcMgr.getExpansionLineNumber(stmt->
    getLocStart());
---
>     return srcMgr.getExpansionLineNumber(stmt->
    getBeginLoc());
1638c1638
<   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getLocStart());
---
>   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getBeginLoc());
1642c1642
<     return srcMgr.getExpansionColumnNumber(stmt->
    getLocStart());
---
>     return srcMgr.getExpansionColumnNumber(stmt->
    getBeginLoc());
1647c1647
<   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getLocStart());
---
>   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getBeginLoc());
1650c1650
<     return srcMgr.getExpansionLineNumber(stmt->
    getLocStart());
---
>     return srcMgr.getExpansionLineNumber(stmt->
    getBeginLoc());
1656c1656
<   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getLocStart());
---
>   FullSourceLoc fullLocation = Context->getFullLoc(stmt->
    getBeginLoc());
1660c1660
```

```
30 <     return srcMgr.getExpansionColumnNumber(stmt->
   getLocStart());
31 ---
32 >     return srcMgr.getExpansionColumnNumber(stmt->
   getBeginLoc());
33 1721,1723c1721,1723
34 <   string str = srcMgr.getFilename(stmt->getLocStart()).
   str();
35 <   //string str3 = srcMgr.getSpellingLoc(stmt->getLocStart
   ()).printToString(srcMgr);
36 <   string str2 = srcMgr.getFileLoc(stmt->getLocStart()).
   printToString(srcMgr);
37 ---
38 >   string str = srcMgr.getFilename(stmt->getBeginLoc()).
   str();
39 >   //string str3 = srcMgr.getSpellingLoc(stmt->getBeginLoc
   ()).printToString(srcMgr);
40 >   string str2 = srcMgr.getFileLoc(stmt->getBeginLoc()).
   printToString(srcMgr);
```

Listing 9: Diff für TraverseASTClassAction.cpp

```
1 15c15
2 < #include <filesystem>
3 ---
4 > #include <experimental/filesystem>
5 22,23d21
6 < using namespace std::tr2::sys;
7 <
8 79c77
9 <       std::string fileName = i->path().filename();
10 ---
11 >       std::string fileName = i->path().filename()->string
   ();
12 143c141
13 <       std::string fileName = i->path().filename();
14 ---
15 >       std::string fileName = i->path().filename()->string
   ();
```

Listing 10: Diff für Helpers.cpp

```
1 24d23
2 < using namespace std::tr2::sys;
```

```
413c412
<
---
>    cl :: ResetCommandLineParser ();
```

Listing 11: Diff für TraverseASTMainHelpers.cpp

Literatur

[1] Daniel Fruhner u. a. „Partitioning and Mapping for Embedded Mutlicore System Utilization in Context of the Model Based Open Source Development Environment Platform AMALTHEA“. en. In: *International Research Conference Dortmund 2014*, S. 11.

[2] *Quellcode von `CommonOptionsParser.cpp` in Version 10.x*. 2019. URL: https://github.com/llvm/llvm-project/blob/release/10.x/clang/lib/Tooling/CommonOptionsParser.cpp (besucht am 30. 07. 2020).

[3] *Quellcode von `CommonOptionsParser.cpp` in Version 4.x*. 2016. URL: https://github.com/llvm/llvm-project/blob/release/4.x/clang/lib/Tooling/CommonOptionsParser.cpp (besucht am 30. 07. 2020).

[4] *Quellcode der `ParseCommandLine`-Methode*. 2020. URL: https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Support/CommandLine.cpp#L1308 (besucht am 30. 07. 2020).

[5] *The LLVM Compiler Infrastructure*. 2020. URL: https://llvm.org (besucht am 17. 01. 2021).

[6] *The LLVM Target-Independent Code Generator*. 2020. URL: https://llvm.org/docs/CodeGenerator.html (besucht am 17. 01. 2021).

[7] *Überladung des opt-Zuweisungsoperators*. 2019. URL: https://llvm.org/doxygen/classllvm_1_1cl_1_1opt.html#a7c51fb5b0045e8c5acc7cc13f57ff016 (besucht am 04. 07. 2020).

[8] *Clang Command Line Manual*. 2020. URL: https://clang.llvm.org/docs/UsersManual.html (besucht am 20. 07. 2020).

[9] *Definition einer Translation Unit nach C99 ISO-Standard*. 2007. URL: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf (besucht am 21. 07. 2020).

[10] *APP4MC-Dokumentation von Tasks und Unterbrechungsroutinen*. 2020. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.8/index.html#section3.14.6 (besucht am 28. 07. 2020).

[11] *Version APP4MC 0.9.6 to APP4MC 0.9.7*. 2020. URL: https://www.eclipse.org/app4mc/help/app4mc-1.0.0/index.html#section4.5.4.4 (besucht am 31. 12. 2020).

[12] *Activity Graph*. 2020. URL: https://www.eclipse.org/app4mc/help/app4mc-1.0.0/images/model_sw_activitygraph_structure.png (besucht am 31. 12. 2020).

[13] *Mode Switch*. 2020. URL: https://www.eclipse.org/app4mc/help/app4mc-1.0.0/images/model_sw_activity_graph_items___mode_switch.png (besucht am 31. 12. 2020).

[14] *Mode Switches*. 2020. URL: https://www.eclipse.org/app4mc/help/app4mc-1.0.0/index.html#section3.13.10.2 (besucht am 31. 12. 2020).

[15] *The reason why dynamic_cast doesn't work with non-polymorphic types*. 2016. URL: https://stackoverflow.com/questions/35729540/the-reason-why-dynamic-cast-doesnt-work-with-non-polymorphic-types (besucht am 10. 01. 2021).

[16] *Matching source-level CFG basic blocks to LLVM IR basic blocks*. 2020. URL: https://adamrehn.com/articles/matching-cfg-blocks-to-basic-blocks/#basic-blocks (besucht am 23. 09. 2020).

[17] *CFGBlock-Dokumentation*. 2020. URL: https://clang.llvm.org/doxygen/classclang_1_1CFGBlock.html (besucht am 23. 09. 2020).

[18] *Generating source-level Control Flow Graph using Clang 4.0*. 2019. URL: http://s4.ce.sharif.edu/blog/2019/12/31/clang/ (besucht am 14. 10. 2020).

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat.
Alle Ausführungen der Arbeit, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Augsburg, 24. Februar 2021

Dominic Beger