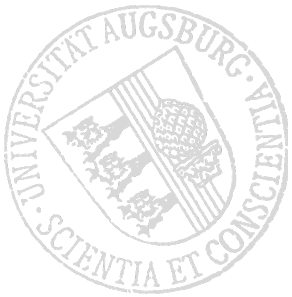
Seminar *Automotive Software Engineering*
Wintersemester 2019

AMALTHEA
Motivation, Profil, Vergleich und Ausblick

Dominic Beger
Matrikelnummer: 1530241
dominic.beger@student.uni-augsburg.de

Betreuer: Christoph Etzel
Softwaremethodik für verteilte Systeme (Prof. Bauer)
Universität Augsburg

Zusammenfassung Die Anforderungen an Hard- und Software in modernen Fahrzeugen wachsen stetig. Der Einsatz von eingebetteten Systemen auf Basis von Mehrkernprozessoren ist insofern keine Seltenheit mehr und bringt eine Menge Komplexität mit sich. Damit bedarf es entsprechenden Werkzeugen und Plattformen, die die zuständigen Entwickler beim kompletten Softwareentwicklungsprozess unterstützen. Diese Seminararbeit gibt einen Überblick über das dafür konzipierte AMALTHEA-Metamodell, die zugehörige Plattform APP4MC und deren Entwicklungsgeschichte. Zusätzlich wird diese mit den etablierten Plattformen ASAM MDX und AUTOSAR verglichen.



1 Einführung und Motivation

Ein Großteil der Funktionalitäten in modernen Fahrzeugen wird heutzutage softwarebasiert von eingebetteten Systemen gesteuert. Fruhner et al. erwähnen dabei in [1, S. 1], ähnlich wie die offizielle APP4MC-Dokumentation in [2, Sektion 1], die steigende Anzahl von Steuergeräten sowie deren größer werdendes Aufgabenfeld. Diese Faktoren sind verantwortlich für eine wachsende Beanspruchung von Hardwareleistung bei gleichzeitig gewünschter, besserer Effizienz, das heißt Senkung von Kosten und Energie. Beispielhaft zu nennen sind komplexe Softwareverbunde bei den Fahrzeugausstattungen, wie ausgereifte Informations-, Unterhaltungs- und Assistenzsysteme, die hierzu in einem entsprechend hohen Umfang beitragen. Dazu zählt unter anderem auch das autonome Fahren. (vgl. [3, S. 515]) Durch ihre Möglichkeit, diese Notwendigkeit zu erfüllen, werden verstärkt Mehrkernprozessoren in den Steuergeräten eingesetzt. Damit ergeben sich auch erhöhte Anforderungen an die Projektentwicklung selbst, da eine Vielzahl von Rahmenbedingungen beachtet werden muss und bestehende Softwarekonstrukte nicht für den Einsatz auf diesen Plattformen optimiert sind. (vgl. [2, Sektion 1]) Zusätzlich muss eine Vielzahl von Herstellern in den Erstellungsprozess integriert werden, da die fertigen Systeme derart komplex sind, dass mehrere Unternehmen an deren Entwicklung beteiligt sind. Zu nennen sind hier etwa die Erstausrüster und die weiteren Unternehmen innerhalb der Zulieferpyramide. An diesem Punkt kommt die modellbasierte Entwicklung mit AMALTHEA ins Spiel. Es bietet diesen innerhalb der Entwicklungskette einen Weg, ihre Artifakte (wie beispielsweise die bestehende Codebasis) zu integrieren und anzupassen, ohne konkrete Implementierungen der jeweiligen anderen Modell-Elemente zu kennen. (vgl. [1, S. 1f], [3, S. 515], [4, S. 1f])

Es stellt sich die Frage, wie das AMALTHEA-Metamodell und die zugehörige Plattform, die APP4MC (Application Platform For MultiCore), aufgebaut sind und welche Möglichkeiten sie anbieten, um komplexe Systeme vollständig beschreiben und den gesamten Softwareentwicklungsprozess unterstützen zu können. Der Begriff Metamodell beschreibt dabei den Aufbau der vorgegebenen Modellelemente (Klassen), aus denen dann konkrete Modelle (Instanzen) entstehen können. Des Weiteren sollte geklärt werden, welche Gründe die Entstehung der Plattform bedingt haben, da mit Standards wie AUTOSAR bereits ein etabliertes Umfeld in der automobilen Domäne vorhanden war.

Um diese Fragen zu beantworten und einen Überblick zu geben, soll diese Seminararbeit zunächst die Entwicklungsgeschichte der AMALTHEA-Plattform bzw. APP4MC aufgreifen und innerhalb dieser auch die Gründe für deren Notwendigkeit erläutern. Anschließend werden der Aufbau und die Eigenschaften der APP4MC erklärt, um aufzuzeigen, welche verschiedenen Werkzeuge darin für den Softwareentwicklungsprozess mit AMALTHEA verwendet werden können. Daraufhin wird das Metamodell selbst in seine Bestandteile gegliedert und diese in einer groben Zusammenfassung mit den jeweiligen Kernkomponenten erläutert, um sichtbar zu machen, aus welchen Eigenschaften und Typen diese jeweils zusammengesetzt sind, wie sie gesamt eine Repräsentation eines Mehrkernsystems ergeben und wie sie automatisiert erzeugt werden können. Schließlich wird noch ein Vergleich von AMALTHEA mit ASAM MDX und AUTOSAR vorgenommen, um eine Einordnung von AMALTHEA in dieses Umfeld zu ermöglichen.

2 AMALTHEA und die APP4MC

Komplexe Mehrkernarchitekturen erfordern detaillierte Ablaufsbeschreibungen und Informationen, um maximale Effizienz zu erzielen und dafür entsprechende Optimierungen vornehmen zu können. Diese Herausforderung ergibt sich schlussfolgernd aus der Tatsache, dass einzelne Folgen von Abläufen nun nicht mehr nur auf einem Prozessorkern, sondern parallel auf mehreren Kernen ausgeführt werden und dies je nach Anforderung entsprechend zeitlich gesteuert werden muss, um diese bestmöglich auslasten zu können. Gleichzeitig muss jedoch eine sichere und zuverlässige Ausführung gewährleistet werden, da an die Systeme harte Echtzeitanforderungen gestellt werden. Aus dem AMALTHEA-Metamodell wird hervorgehen, dass Rahmenbedingungen wie Zeit, die für bestimmte Aufgaben höchstens zur Verfügung steht, oder auch andere Variablen, wie Bereitstellung und Verteilung von Speicher oder Rechenleistung, hauptsächlich zu diesen Beschreibungen beitragen. Mit existierenden Plattformen, wie beispielsweise AUTOSAR, war es möglich, komponentenbasierte Softwarekonstrukte und standardisierte Architekturen zu entwickeln. Eine Möglichkeit, derartige Abläufe ausreichend detailliert zu beschreiben, gab es allerdings nicht. Angetrieben durch diese Notwendigkeit entstand im Juli 2011 das ITEA 2 AMALTHEA-Projekt. Beteiligt waren fünfzehn Partner aus Finnland, Deutschland und der Türkei. Dieses basierte auf existierenden Entwicklungswerkzeugen, die angepasst wurden, und führte mit dem abstrakten AMALTHEA-Metamodell eine Möglichkeit ein, diese Abläufe auszudrücken. Da die Entwicklung der Plattform auch den angestrebten Nutzen in der praktischen Anwendung haben sollte, von dem die Industrie profitieren sollte, entstand auf Basis des Erstprojektes im September 2014 AMALTHEA4public. (vgl. [5, S. 26f]). Aus [5, S. 27] lässt sich außerdem schließen, dass dies den gewünschten Erfolg erbracht hat, da AMALTHEA in der Industrie von Automobilunternehmen wie BMW, Daimler, Volkswagen und PSA eingesetzt wird. Durch die Veröffentlichung entstand zudem eine Gemeinschaft aus Mitwirkenden und Nutzern, die die Weiterentwicklung fördert. An AMALTHEA4public waren insgesamt zwanzig Partner aus Deustchland, Spanien, Schweden und der Türkei beteiligt. 2015 wurde das AMALTHEA-Projekt in die APP4MC (Application Platform For MultiCore) auf Basis von Eclipse übernommen. (vgl. [5, S. 26]) Diese Arbeit bezieht sich auf den Stand des aktuell neuesten Releases (0.9.6). (vgl. [6])

2.1 Aufbau der Plattform

Abbildung 1 zeigt die einzelnen Komponenten der APP4MC und deren Beziehungen zueinander. Hierbei bauen obere Elemente auf den unteren auf bzw. verwalten diese. Die Entwicklungsumgebung Eclipse stellt aufgrund ihrer offenen und erweiterbaren Charakteristik eine Grundlage für die Integration unterschiedlicher Softwareentwicklungsframeworks zur Verfügung. Somit entstand auch die APP4MC auf Basis dieser. (vgl. [3, S. 517]) Die Robert Bosch GmbH stellt als einer der Hauptentwickler der Plattform zusätzlich unterstützende Werkzeuge zur Verfügung. Dabei sind interne Tools mit kommerzieller oder In-House-Verfügbarkeit (vgl. die blauen Elemente in Abbildung 1) von öffentlich zugänglichen Tools (vgl. die grünen Elemente in Abbildung 1) zu unterscheiden.

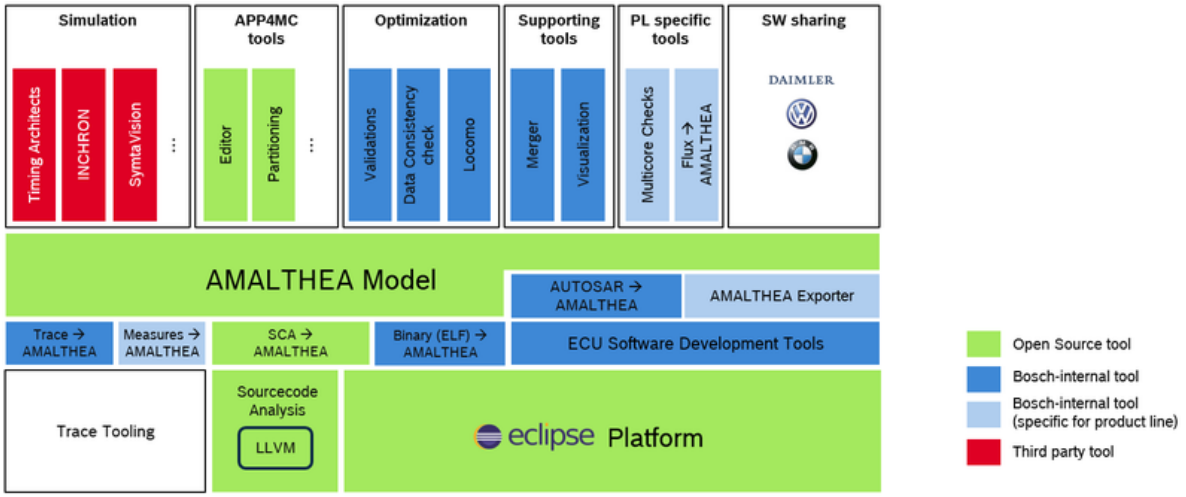


Abbildung 1: Der Aufbau der APP4MC-Plattform mit zugehörigen Werkzeugen [7]

Bosch möchte mit der Zeit einige Werkzeuge öffentlich machen und die Weiterentwicklung der Plattform damit unterstützen. Getan wurde das bereits bei SCA2AMALTHEA. Die Integration dieser Tools wird ebenso durch das Pluginsystem von Eclipse ermöglicht und eigene Erweiterungen können in Java entwickelt werden. Eine Reihe von Werkzeugen erlaubt eine automatische (Teil-)Modellgenerierung aus verschiedenen Quelldaten, wie beispielsweise Quellcode (C/C++) oder Binärcode (ELF). Auch eine Übersetzung von AUTOSAR-Modellen in AMALTHEA ist damit möglich. Diese Tools sind in Abbildung 1 zwischen dem Element "AMALTHEA Model" und der untersten Reihe ersichtlich. Die Analysewerkzeuge beispielsweise bauen nicht direkt auf der Eclipse-Plattform auf, sondern basieren auf externen Anwendungen. Wie auch in [5, S. 27] erwähnt, ist das Modell selbst der zentrale Teil der APP4MC und kann innerhalb dieser verwaltet werden. Hierfür steht ein integrierter Editor bereit. Auch die Partitionierung von Prozessen und Runnables (siehe Softwaremodell) für das Mapping (siehe Mapping-Modell), das heißt die Gruppierung nach Laufzeiten, ist bereits in die Plattform integriert. Erweiterungen für Validierung, Optimierung und die Simulation (Tests) sind nicht direkt eingebunden, sondern werden extern von Bosch oder anderen Entwicklern in Form von Plugins oder Drittanwendungen (rote Elemente in Abbildung 1) bereitgestellt. Zu nennen sind hier unter anderem die TA Tool Suite des Unternehmens Timing Architects, welches seit Januar 2018 zu Vector Informatik gehört, und die Software der INCHRON GmbH. Beide unterstützen den Softwareentwicklungsprozess für Mehrkernsysteme und stellen simulative Methodiken für das Testen und Verifizieren zur Verfügung. (vgl. [8], [9]) Die Plattform wird in einem dreimonatigen Entwicklungszyklus aktualisiert, um die Aktualität des AMALTHEA-Metamodells sicherzustellen. (vgl. [5, S. 27])

2.2 Aufbau und Erzeugung des AMALTHEA-Modells

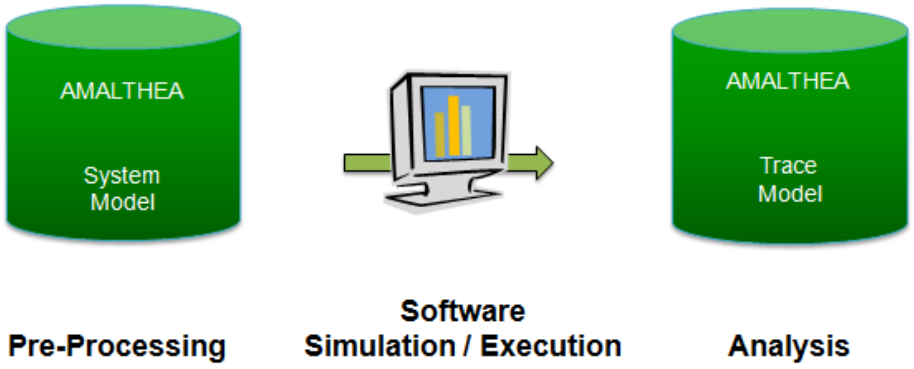


Abbildung 2: Beide Teilmodelle im Entwicklungsprozess [10]

Der Softwareentwicklungsprozess mit AMALTHEA umfasst zwei verschiedene Modelle: Das Systemmodell und das Tracing-Modell (siehe Abbildung 2). Ersteres dient zum Entwurf des Systems und letzteres dient als Grundlage für die Auswertung von Leistungsmessungen nach dessen Simulation. (vgl. [2, S. 3.1]) In dieser Arbeit soll nur das Systemmetamodell und dessen Elemente als Grundlage für die Beschreibung betrachtet werden (siehe Abbildung 3). Das Metamodell basiert auf Objektorientierung und umfasst vorgefertigte Klassen, die mit einer hierarchischen Struktur das gesamte System repräsentieren können und auf denen die spätere Auswertung und Erzeugung von Daten basiert. Instanzen innerhalb einer Struktur können mit anderen über den Eigenschaftsreiter des Modell-Editors assoziiert werden. Dadurch werden Duplikate und tiefe Schachtelungen vermieden und die mehrfache Verwendung eines Objektes ermöglicht. Für Prozessoren mit gleichen Leistungsdaten kann beispielsweise eine Definition an beide Prozessoren angehängt werden, statt diese für jeden einzeln zu deklarieren. Es wurden zusätzlich einige Datentypen und Enumerationen eingeführt, die benötigte Informationen repräsentieren. Dazu zählen beispielsweise Datengrößen und -raten, Abweichungen, Grenzen, Frequenz- und Zeiteinheiten. Jedes Element, das durch eine Modellklasse erfasst wird, umfasst eine bestimmte Anzahl von Eigenschaften, die mit jeweiligen primitiven oder diesen speziell eingeführten Typen ausreichend informativ beschrieben werden. Auch entsprechende Annotationselemente oder Zusatzklassen für die Darstellung spezieller Anforderungen bzw. Restriktionen existieren innerhalb des Metamodells. Auf diese wird in dieser Arbeit nicht detailliert eingegangen. Stattdessen werden die größten und wichtigsten Teil-Metamodelle beschrieben und wie diese im Verbund eine gesamte Repräsentation eines Systems ergeben. Darunter fallen Hardware, Betriebssystem, Software, Rahmenbedingungen, Mapping und Stimuli. Letztere werden dabei jedoch nur im Abschnitt des Softwaremodells kurz erläutert.

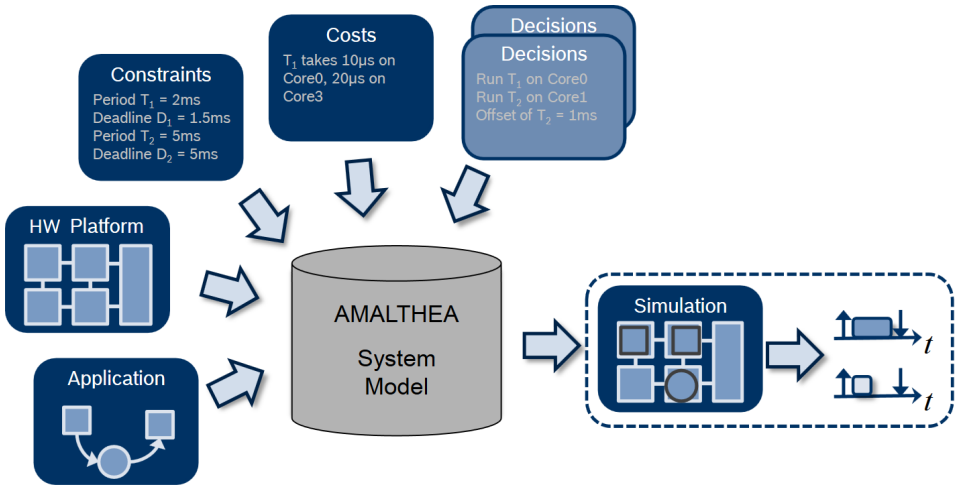


Abbildung 3: Das AMALTHEA-Systemmodell und dessen grobe Zusammensetzung [11]

2.2.1 Hardwaremodell

Dieser Teil entspricht dem Element "HW-Platform" in Abbildung 3. Das Hardwaremodell (Klasse `HWModel` im Metamodell) beschreibt die Zusammensetzung der Hardwarekomponenten innerhalb des Systems. Alle weiteren zur Spezifikation notwendigen Instanzen von Klassen werden von einem Objekt dieser Klasse assoziiert. Eine Übersichtsgrafik für den genauen Aufbau ist im Anhang bei Abbildung 4 zu finden. Die Darstellung von Hardwarestrukturen (hierzu zählen beispielsweise Steuergeräte, Mikrokontroller oder Cluster) erfolgt über die Klasse `HwStructure`. Dabei kann eine Hardwarestruktur rekursiv weitere Hardwarestrukturen referenzieren. Es ist also zum Beispiel möglich, innerhalb eines Steuergerätes mehrere Mikrokontroller zu beschreiben. Jede Struktur enthält dabei selbst eine Vielzahl von Hardwaremodulen, repräsentiert durch die abstrakte Klasse `HwModule`. Dazu zählen Prozessoren, Cache- und Arbeitsspeicher sowie deren Verbindungen zueinander, beispielsweise über Bussysteme oder einen Cluster Interconnect. Mehrere Kerne oder Speichermodule werden im Modell als eigenständige Prozessoren und Speicher angegeben. Die benötigten Daten für jeden Modultyp, wie die Art eines Caches (Daten oder Befehlscache), werden dabei in spezielle `HwDefinition`-Unterklassen (hier dann beispielsweise eine `CacheDefinition`) ausgelagert, die spezifische Attribute für die Beschreibung enthalten und für die Modulinstanzen sichtbar sind. Für die beiden Schnittstellen jeder Kommunikationsleitung (`HwConnection`) innerhalb einer Struktur müssen entsprechende Hardware-Ports (`HwPort`) spezifiziert werden, die Informationen zu Datenbreite, Leitungstyp (dazu zählen CAN, I2C, Ethernet und mehrere weitere Standards), Kommunikationsrolle (Sender oder Empfänger) und Priorität enthalten. Details zu Leistung (Strom) oder Frequenz von Komponenten werden über die `PowerDomain` und `FrequencyDomain` (beides Unterklassen von `HwDomain`) in der Wurzel des Hardwaremodells verfügbar gemacht. (vgl. [2, Sektion 3.8])

2.2.2 Betriebssystemmodell

Das Betriebssystem ist die Basis für die eingesetzte Software und interagiert direkt mit der Hardwareschicht, die vorher beschrieben wurde. Es ist somit ein Modell notwendig, um das jeweilige Scheduling-Verhalten sowie die Verwaltung von Ressourcen zu beschreiben. Entsprechend lassen sich Konzepte wie Semaphoren, Puffer oder Aufgabenplanung in diesem Metamodell finden. Das Wurzelelement ist ein Objekt der `OSModel`-Klasse. Dieses kann mehrere Betriebssystemdefinitionen (`OperatingSystem`) enthalten, welche eine Liste von `TaskSchedulern` und `InterruptControllern` verfügbar machen (siehe Abbildung 5 im Anhang). Diese beiden sind Spezifizierungen der abstrakten `Scheduler`-Klasse und definieren über die `SchedulingAlgorithm`-Klasse einen entsprechenden Planungsalgorithmus für die Priorisierung und Ausführung von Aufgaben sowie den Einsatz von Unterbrechungsroutinen (ISR). Eine Verteilung auf die entsprechenden Prozessorkerne wird jedoch erst im Mapping-Modell beschrieben. Dabei steht eine Reihe von vordefinierten Algorithmen zur Verfügung. Beispielsweise gibt es für AUTOSAR-basierte Systeme die Klasse `FixedPriorityPreemptive`. Zusätzlich assoziiert mit dem Wurzelelement können Semaphoren (`Semaphore`) sein, die die Anzahl der Prozesse beschränken, die auf eine gemeinsame Ressource gleichzeitig zugreifen dürfen. Des Weiteren kann der Overhead (hier: zusätzliche Zeit, die für die Abarbeitung einer bestimmten Aufgabe noch eingeplant werden muss) beim Aufruf von jeweiligen Systemfunktionen (Syscall) mit der benötigten Zahl an Ticks spezifiziert werden. Dies ist notwendig, um die Zeitberechnungen in der späteren Analyse mit der erforderlichen Präzision durchzuführen. Hierfür steht die `OsOverhead`-Containerklasse mit Unterteilungen in `OsAPIOverhead` und `OsISROverhead` (Kategorie 1 und 2) zur Verfügung. Objekte dieser Containerklasse können zu jeweils einem Betriebssystem zugeordnet werden. Dies ist in den Abbildungen 5 und 6 im Anhang ersichtlich. Abschließend ist die Beschreibung der Datenkonsistenz mittels `OSDataConsistency` zu nennen. Diese wird pro `OperatingSystem`-Instanz angegeben und soll für externe Werkzeuge Informationen und Berechungsresultate zur Datenintegrität und -kohärenz bereitstellen. Dazu kann ein Absicherungsmodus, ein `DataStability`-Objekt mit Ausführungseinheiten (siehe Softwaremodell), für die Datenkonsistenz gewährleistet sein muss, sowie ein `NonAtomicDataCoherency`-Objekt, für dessen Ausführungseinheiten Kohärenz vorliegen muss, angegeben werden. (vgl. [2, Sektion 3.11]) Ein entsprechendes Diagramm für diese Klassen ist ebenfalls im Anhang unter Abbildung 7 vorhanden.

2.2.3 Softwaremodell

Um Teile der Softwarestruktur adäquat auf den verfügbaren Speicher der Hardware verteilen zu können, muss jedes beteiligte Element Informationen zur verwendeten Datengröße bereitstellen. Für die Repräsentation dieser Elemente gibt es eine abstrakte Klasse `AbstractMemoryElement`. Es existieren insgesamt sechs Unterklassen von dieser, die ein veerbtes `size`-Attribut vom Typ `DataSize` mit Wert und Einheit enthalten. Die Darstellung von einzelnen Datenelementen in einem Kontext, wie etwa Parameter, Variablen oder Konstanten, wird durch die Klasse `Label` übernommen. Ein `Channel`

verwaltet eine durchlaufende Menge von Elementen eines Typs im Speicher, stellt also einen Puffer dar, der beispielsweise für Transfers von größeren Datenemengen verwendet werden kann. Auch ausführbare Einheiten wie Funktionen werden durch eine Unterklasse `Runnable` repräsentiert. Gleichzeitig existieren `Task` und `ISR`-Klassen, die sich über eine abstrakte Oberklasse `Process` in diese Liste eingliedern. Der Zusammenhang dieser Klassen sowie die beschriebene Vererbungshierarchie sind in Abbildung 8, welche sich im Anhang befindet, ersichtlich. An alle drei können direkt Stimuli angehängt werden, die sie aktivieren. Es gibt einige verschiedene Arten von ihnen. Darunter fallen unter anderem der `SingleStimulus` für einmalige, zeitliche Aktivierung, periodische Stimuli für wiederkehrende Aktivierungen, ereignisbasierte, stochastisch ausgelöste Stimuli und variable Stimuli, die von selbst gewählten Größen (beispielsweise einer bestimmten Geschwindigkeit) abhängig sind.

Der Unterschied zwischen einem `Task` und einem `Runnable` liegt darin, dass ersterer direkt dem Betriebssystem und damit auch dessen Ausführungsplanung unterliegt und nur von diesem aktiviert werden kann. Ein `Runnable` kann hingegen nur innerhalb von Tasks (insofern das Runnable nicht als Service deklariert wurde) und anderen Runnables aktiviert werden.

Zu jedem dieser Elemente kann ein Call Graph gehören. Dieser modelliert den Kontrollfluss eines Softwarekonstruktes. Da eine Beschreibung aller verschiedenen Graphelemente den Rahmen dieser Arbeit überschreiten würde, sollen nur grobe Konzepte und ein paar der wichtigsten Elemente erläutert werden. Die Elemente können innerhalb von Gruppen, die zur Strukturierung dienen, der Reihe nach von oben nach unten oder zufällig ausgeführt werden. Die verschiedenen Call Graph Items umfassen Zugriffe auf Labels, Channel, Semaphoren, Aufrufe von Runnables, Mode- und Probability Switches für alternative Ausführungspfade mit Bedingungen (if-else bzw. switch-case), verschiedene Events und Trigger, auf die gewartet werden kann sowie Annotationselemente für Informationen zu Ausführungszeiten (Ticks und Execution Needs). (vgl. [2, Sektion 3.14]) Ein beispielhafter Aufbau eines Call Graphs zur groben Veranschaulichung sowie dessen verschiedene Items werden in den Abbildungen 9 und 10 aufgezeigt.

SCA2AMALTHEA SCA2AMALTHEA ist ein von Bosch veröffentlichtes Werkzeug. Es kann unter dem zugehörigen Git-Repository[1] abgerufen werden und nutzt die LLVM-Compilerinfrastruktur mit clang als Frontend, um existierenden C- oder C++-Code zu analysieren (Static Code Analyzer) und in ein AMALTHEA-Softwaremodell zu übersetzen. Die oben angesprochenen Elemente werden dabei automatisch generiert, da ein manuelles Erstellen bei großen Codedateien entsprechend aufwendig wäre. Die Realisierung erfolgt dabei über ein Plugin, das in APP4MC installiert werden kann. (vgl. [12, S. 22])

[1] https://git.eclipse.org/r/plugins/gitiles/app4mc/org.eclipse.app4mc.tools/+/13baf17cfe40fef2b3a58f27ee044e3afa255cfa/eclipse-tools/sca2amalthea

2.2.4 Rahmenbedingungen

Im Rahmenbedingungsmodell (vgl. “Constraints“ in Abbildung 3) können Anforderungen und Laufzeitdetails für verschiedene Objekte der anderen Teilmodelle definiert werden. Dies ist wichtig, da Aufgaben in Echtzeitsystemen zuverlässig erledigt werden müssen und gleichzeitig eine bestimmte, maximale Ausführungszeit benötigen dürfen. Die Hardwarerechenleistung ist begrenzt und bei jeder Operation müssen viele zusätzliche Parameter berücksichtigt werden (beispielsweise Overhead oder Zugriffslatenzen). Anforderungen werden durch die abstrakte `Requirement`-Klasse beschrieben, für die Spezifizierungen zur gesamten Architektur, zu Runnables, Prozessen und Prozessketten zur Verfügung stehen. Dabei kann jede Anforderung ein `RequirementLimit` definieren, das wiederrum selbst von unterschiedlichen Begrenzungstypen (Zeitlimits, Frequenzlimits sowie prozentuale und absolute Zahlenvorgaben) abstrahiert. Jeder dieser Begrenzungstypen definiert dabei einen Wert und eine jeweilige Metrik (über eine Enumeration), damit klar ist, welche Grenze die angegebene Zahl beschreiben soll. Metriken sind beispielsweise Speicherzugriffs- oder Prozessausführungszeiten bei Zeitbegrenzungen, Cache-Hit-Frequenzen bei Frequenzlimits oder Cache-Hit-Raten bei den prozentualen Angaben. Bei den `CountRequirementLimit`s für maximal zulässige Anzahlen von Ereignissen kann etwa die Begrenzung von Prozessaktivierungen genannt werden. Das zugehörige Klassendiagramm wird in Abbildung 11, welche im Anhang zu finden ist, dargestellt. Bei der Anordnung mehrerer Runnables innerhalb eines Call Graphs ist keine chronologische Ausführung sichergestellt, da diese zufällig passiert. Entsprechend kann mit dem `RunnableSequencingConstraint` eine Aufrufsreihenfolge definiert werden. Um die Aktualität von Daten zu gewährleisten, kann außerdem ein `DataAgeConstraint` verwendet werden, der Grenzen für das Alter eines Speicherelements definiert. Für Datenkohärenz und -konsistenz für Gruppen von Labels in ihren jeweiligen Prozessen und Runnables existiert die `DataCoherencyGroup`- und `DataStabilityGroup`-Klasse. Innerhalb der Zeitlimits sind zusätzlich Angaben zu Wiederholungen, Verzögerungen und Synchronisation von Ereignissen möglich. Die jeweiligen Rahmenbedingungen sind direkt im `ConstraintsModel`, das als Wurzelklasse dient, enthalten. (vgl. [2, Sektion 3.6])

2.2.5 Mapping-Modell

Dieses Metamodell entspricht den “Decisions“ in Abbildung 3. Für das im Betriebssystemmodell bereits angesprochene Mapping werden die entsprechenden Elemente des Hardware- und Softwaremodells mit den Schedulern des Betriebssystems assoziiert. Hierfür stehen Allocations bereit. Die `SchedulerAllocation`-Klasse ist dabei für die Verteilung der Tasks auf die einzelnen Prozessorkerne zuständig. Folglich muss mindestens eine `ProcessingUnit`, für die diese verantwortlich ist, zugewiesen werden. Da das Scheduling selbst gewissermaßen einen Task darstellt und dieser auch von einem Prozessorkern durchgeführt werden muss, muss dem `SchedulerAllocation`-Objekt die verantwortliche `ProcessingUnit` zugeordnet werden, um den dabei entstehenden Overhead korrekt berechnen zu können. Ein einzelner `Scheduler` kann mehrere dieser `SchedulerAllocation`s enthalten. Für die Tasks und Unterbrechungsroutinen, die auf den Kernen ausgeführt

werden sollen, haben die jeweiligen Scheduler (Controller) eine Liste von `TaskAllocation` bzw. `ISRAllocation`-Instanzen, die sie verwalten. Jede von diesen beiden Klassen hält zudem eine Liste von `RunnableAllocations`. Die `RunnableAllocations` enthalten jeweils genau ein `Runnable`. Analog assoziieren Task- und ISR-Scheduler jeweils genau ein `Task`- und `ISR`-Element. Abbildung 12 im Anhang veranschaulicht diesen Aufbau nochmals grafisch.

Da die `AbstractMemoryElement`-Objekte nicht direkt innerhalb der Prozessorregister zur Verfügung gestellt werden können, muss bis zum Laden für deren Verteilung im Haupt- oder Cachespeicher auch ein entsprechendes Memory Mapping vorgenommen werden. Hierfür gibt es die Klasse `MemoryMapping`. In ihr können die jeweiligen Elemente im Softwaremodell mit den `Memory`-Instanzen aus dem Hardwaremodell verbunden werden, auf die sie verteilt werden sollen. Dabei kann noch eine entsprechende Speicheradresse für die genaue Position angegeben werden. Die `MemoryAddressMappingType`-Enumeration dient dabei für die Angaben zur Adressierungsart (absolute Position, relativ zum ersten Speicherelement oder gar keine Information). (vgl. [2, Sektion 3.9])

3 Vergleich

Neben AMALTHEA sind in der automobilen Domäne weitere, anerkannte Standards vorhanden. Dazu zählen unter anderem ASAM MDX und AUTOSAR, welche auch modellbasiert sind. Im Folgenden sollen der grundsätzliche Aufbau der Metamodelle und die jeweiligen Einsatzgebiete sowie Ziele der Plattformen verglichen werden, um eine Einordnung von AMALTHEA in dieses Umfeld vorzunehmen. Alle drei Modelle basieren auf dem XML-Format. (vgl. [4, S. 1f])

3.1 Vergleich zu ASAM MDX

MDX steht für Model Data Exchange Format und stellt ein Metamodell zur Verfügung, das zur Darstellung von Softwarekomponenten eines Steuergerätes verwendet werden kann. Es deckt im direkten Vergleich mit AMALTHEA gewissermaßen den Teil des Softwaremodells ab und bietet keine Möglichkeiten, Informationen zu Hardwareaufbau oder Betriebssystemen abzubilden. Genaue Ablaufsbeschreibungen durch z.B. einen Call Graph gibt es nicht. Ein Scheduling für die Festlegung der Ausführungsreihenfolge von Funktionen und Angaben zu deren erlaubter Ausführungszeit sind jedoch möglich. Ebenso können wie in AMALTHEA auch Rahmenbedingungen für Datenalter, Gruppen für Datenkohärenz und Spezifikationen für das Mapping der Softwarestruktur auf den Speicher angegeben werden. Der Fokus von MDX liegt damit nicht auf Analyse, Verwaltung oder Optimierung von Prozessen auf eingebetteten Systemen, sondern hauptsächlich der Dokumentation über ein nicht-proprietäres Modellformat. Verfügbar wird es über einen Kauf oder eine Mitgliedschaft bei der ASAM. Es handelt sich im Gegensatz zur APP4MC mit AMALTHEA folglich nicht um eine öffentlich zugängliche Plattform. Wie AMALTHEA auch, hat MDX Relevanz für die Kommunikation zwischen den an der

Entwicklung des Systems beteiligten Unternehmen, da es Schnittstellen für externe Applikationen anbietet und diese entsprechend integriert werden können. (vgl. [4, S. 2ff])

3.2 Vergleich zu AUTOSAR

AUTOSAR steht für Automotive Open System Architecture und definiert einen Architekturstandard für die Beschreibung und Repräsentation von verteilten Systemen. Die entsprechende Spezifikation ist öffentlich zugänglich. Die Plattform für den konkreten Entwicklungsprozess steht dabei nur Mitgliedern und Partnern zur Verfügung. Sailer et. al erwähnen in [4, S. 3f], dass das AMALTHEA-Modell den Fokus auf die Beschreibung eines einzelnen Steuergerätes läge, wohingegen AUTOSAR die Möglichkeit bietet, einen ganzen Verbund von Steuergeräten und damit das gesamte System als solches zu repräsentieren. Aus dem AMALTHEA-Metamodell geht jedoch gegenteilige Information hervor. Da Sailer et. al Versionen aus 2015 als Referenz nutzen, ist es möglich, dass sich das Modell dahingehend geändert hat. Schließlich können mit AMALTHEA auch derartige Verbunde von Hardwarestrukturen beschrieben werden, die dann z.B. eine Mehrzahl von Steuergeräten und deren Verbindungen über Ports und ConnectionHandler enthalten. Das AUTOSAR-Modell ist wie AMALTHEA auch in der Lage, Hard- und Software zu beschreiben. Dazu besteht es aus einer System Configuration Description und ECU Configuration Description. Erstere, welche stark an das Modell von ASAM MDX angelehnt ist, bildet dabei die komplette Softwarestruktur ab und beschreibt zudem auch die Kommunikation zwischen den Steuergeräten. Die ECU Configuration Description bildet entsprechend die Hardware ab. Komplexe Strukturen und deren relevante Komponenten (Prozessoren, Speicher, Sensoren, ...) können beschrieben werden. Zusätzlich besteht die Aufgabe dieser Konfiguration darin, die nötigen Parameter für die Ausführung der Software bereitzustellen. Dazu zählen dann Informationen zur Laufzeitumgebung und zum Betriebssystem. Auch werden die im System Configuration Model angegebenen Funktionen auf dieser Ebene in Tasks repräsentiert und deren Ausführung geplant. Die Verteilung der Software selbst auf die einzelnen Steuergeräte funktioniert dabei wie in AMALTHEA auch über ein Mapping. Für Rahmenbedingungen stehen lediglich Zeitbegrenzungen für Tasks und Anforderungen an Events bereit. Andere Metriken sind nicht definierbar. Detaillierte Abläufe können zudem in AUTOSAR nicht ausreichend ausgedrückt werden. Dies ist jedoch für die Beschreibung von Mehrkernsystemen notwendig. Ursächlich dafür sind fehlende Call Graphs und die mangelnde Unterstützung von stochastischen, variablen Daten, beispielsweise bei den Stimuli oder Ausführungszeiten. Zusammenhängende Ausführungseinheiten sind ebenso nicht ausdrückbar, da es keine Affinity Constraints gibt und auch Gruppen für Datenkohärenz fehlen. AMALTHEA bietet dadurch, dass es diese Aspekte beschreiben kann, somit einen entsprechenden Vorteil in diesem Bereich. (vgl. [4, S. 2ff])

4 Zusammenfassung und Ausblick

Abschließend ist festzuhalten, dass AMALTHEA für die Beschreibung von Mehrkernsystemen ein umfangreiches Metamodell zur Verfügung stellt und mit der APP4MC eine erweiterbare Basis geboten wird, die die Entwickler im kompletten Entwicklungsprozess unterstützt. Dabei hat sich diese in einem bestehenden Umfeld mit anerkannten Standards wie AUTOSAR etabliert und hat Relevanz für Softwareprojekte in der heutigen Automobilindustrie. Die aktuelle Version 0.9.6 wird voraussichtlich auf Grundlage des dreimonatigen Entwicklungszyklus' im Februar 2020 durch eine neue Version abgelöst. Zum jetzigen Zeitpunkt sind keine Planungen für umfangreiche Veränderungen am Projekt oder Metamodell selbst bekannt. Es ist also davon auszugehen, dass der jetzige Stand auch weiterhin die notwendigen Anforderungen erfüllt und entsprechende Verwendung findet.

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] *APP4MC-Dokumentation (Version 0.9.6)*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/index.html (besucht am 14. 12. 2019).

[3] Carsten Wolff u. a. „AMALTHEA — Tailoring tools to projects in automotive software development“. In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Bd. 2. Sep. 2015, S. 515–520. DOI: 10.1109/IDAACS.2015.7341359.

[4] Andreas Sailer u. a. „Distributed Multi-Core Development in the Automotive Domain - A Practical Comparison of ASAM MDX vs. AUTOSAR vs. AMALTHEA“. In: *ARCS 2016; 29th International Conference on Architecture of Computing Systems*. Apr. 2016, S. 1–8.

[5] Johan van der Heide. *AMALTHEA and AMALTHEA4public Success story*. URL: https://itea3.org/project/success-story/amalthea-and-amalthea4public-success-story.html (besucht am 29. 11. 2019).

[6] *Neuester APP4MC-Release*. 2019. URL: https://www.eclipse.org/app4mc/news/2019-11-04-release-0-9-6/ (besucht am 11. 12. 2019).

[7] *File:app4mc_platform.png*. 2019. URL: https://itea3.org/img/i/5353-1551885267.png (besucht am 11. 12. 2019).

[8] *Timing Architects*. 2019. URL: https://www.timing-architects.com (besucht am 22. 12. 2019).

[9] *INCHRON GmbH*. 2019. URL: https://www.inchron.com (besucht am 22. 12. 2019).

[10] *File:amalthea_models.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/amalthea_models.png (besucht am 22. 12. 2019).

[11] *File:amalthea_model.png*. 2019. URL: https://www.eclipse.org/app4mc/images/system-model.png (besucht am 22. 12. 2019).

[12] *AMALTHEA - D 1.3 Design Handbook*. 2017. URL: https://itea3.org/project/workpackage/document/download/3964/D1.3.%20AMALTHEA4public%20-%20Design%20Handbook.pdf (besucht am 26. 12. 2019).

[13] *File:model_hw_main.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_hw_main.png (besucht am 25. 01. 2020).

[14] *File:model_os_operatingsystem.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_os_operatingsystem.png (besucht am 25. 01. 2020).

[15] *File:model_os_osoverhead.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_os_osoverhead.png (besucht am 25. 01. 2020).

[16] *File:model_os_data$_c$onsistency.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_os_data_consistency.png (besucht am 25. 01. 2020).

[17] *File:model_swmemory_inf.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_sw_memory_inf.png (besucht am 25. 01. 2020).

[18] *File:model_sw_callgraph_structure.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_sw_callgraph_structure.png (besucht am 25. 01. 2020).

[19] *File:model_sw_callgraph_items.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_sw_callgraph_items.png (besucht am 25. 01. 2020).

[20] *File:model_constraints_requirements.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_constraints_requirements.png (besucht am 25. 01. 2020).

[21] *File:model_mapping_allocation.png*. 2019. URL: https://www.eclipse.org/app4mc/help/app4mc-0.9.6/images/model_mapping_allocation.png (besucht am 25. 01. 2020).

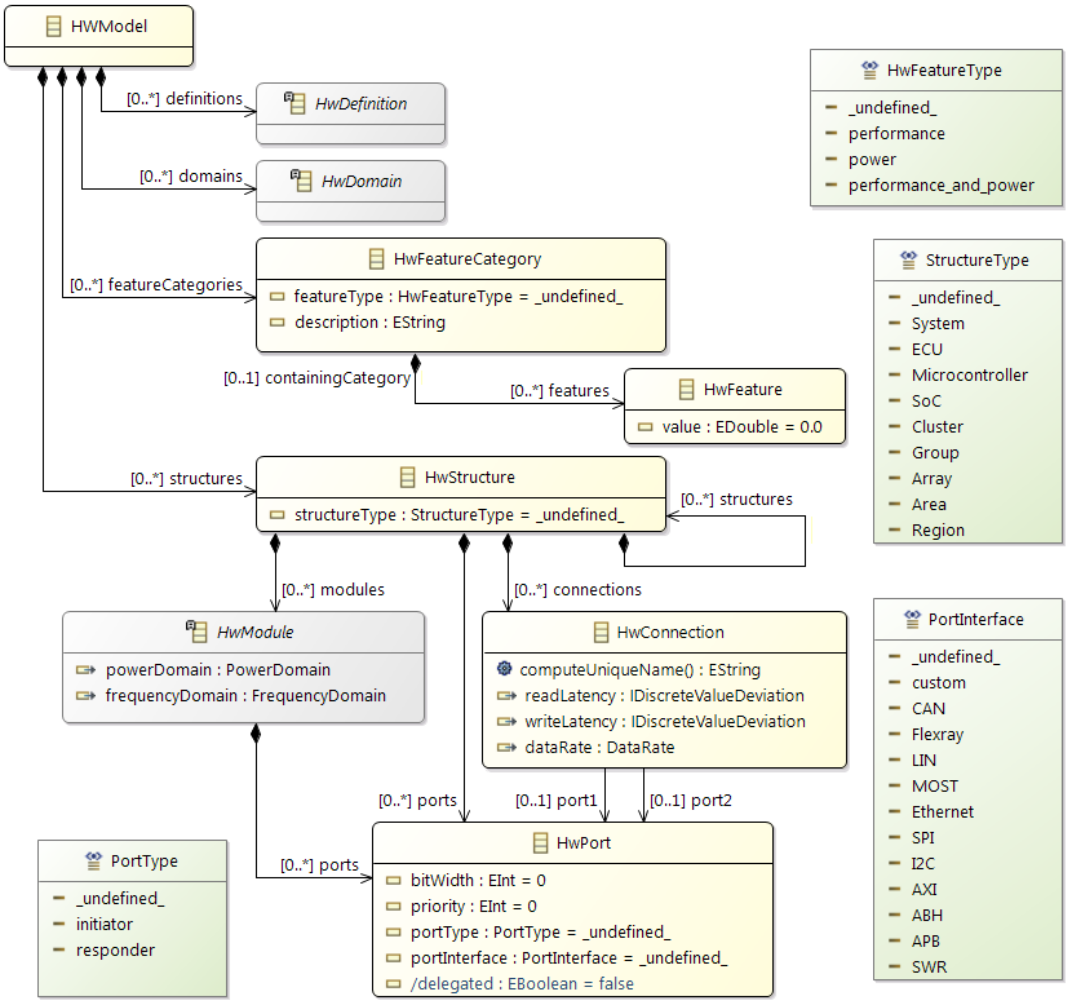


Abbildung 4: Der Aufbau des Hardwaremodells [13]

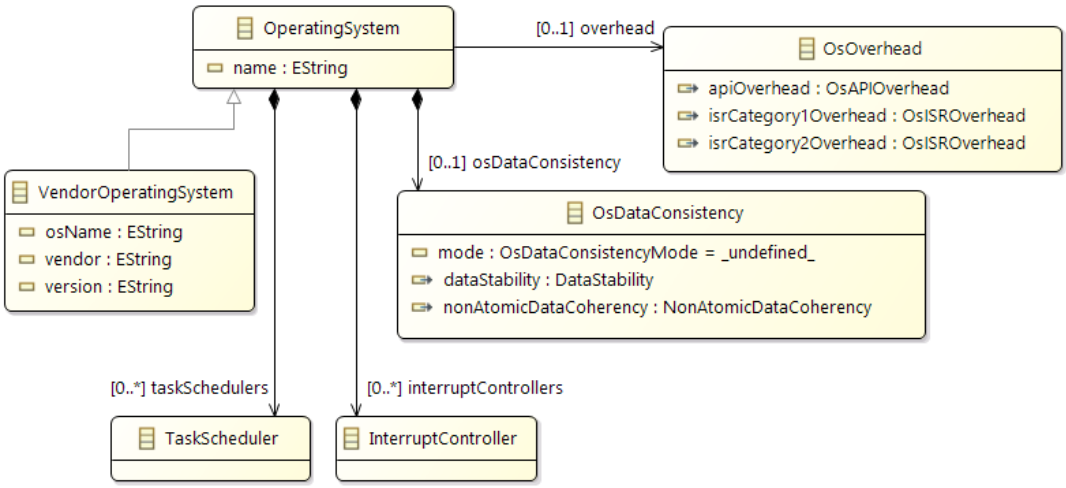


Abbildung 5: Der Aufbau des Betriebssystemmodells [14]

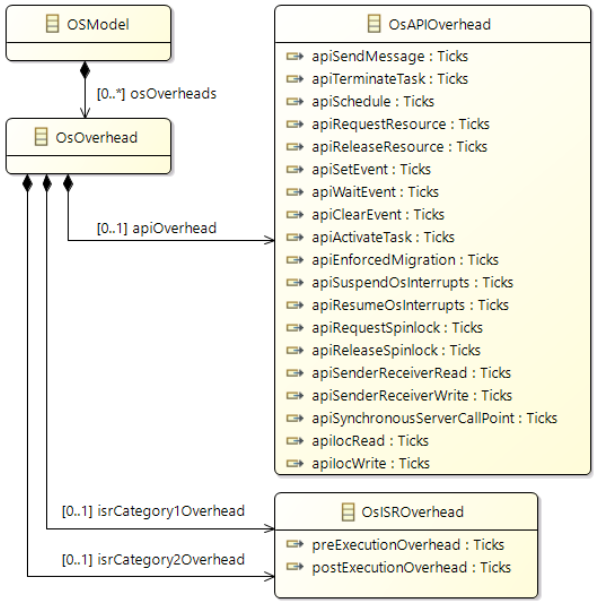


Abbildung 6: Der Aufbau des Modells zur Beschreibung von Overhead [15]

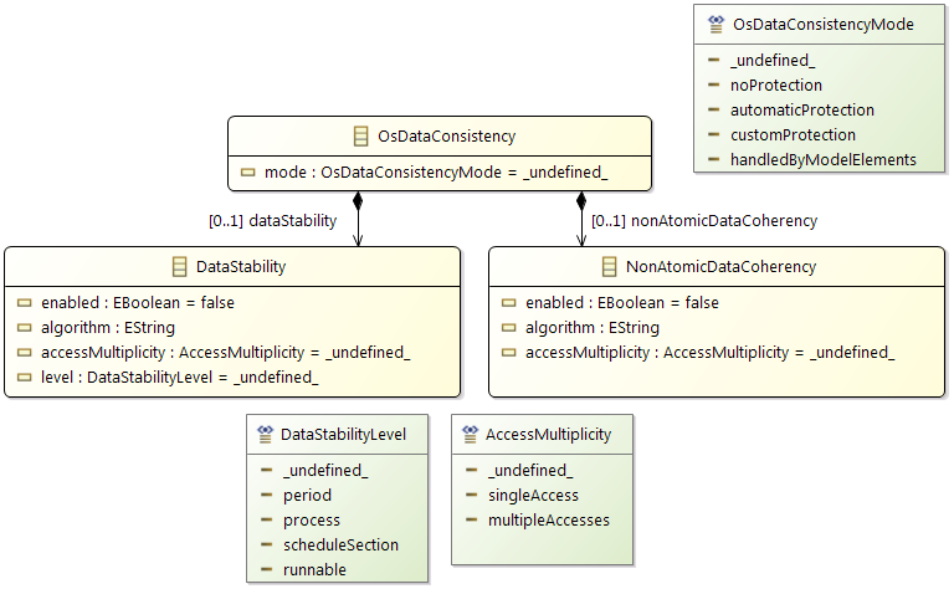


Abbildung 7: Der Aufbau des Modells zur Beschreibung von Datenkonsistenz [16]

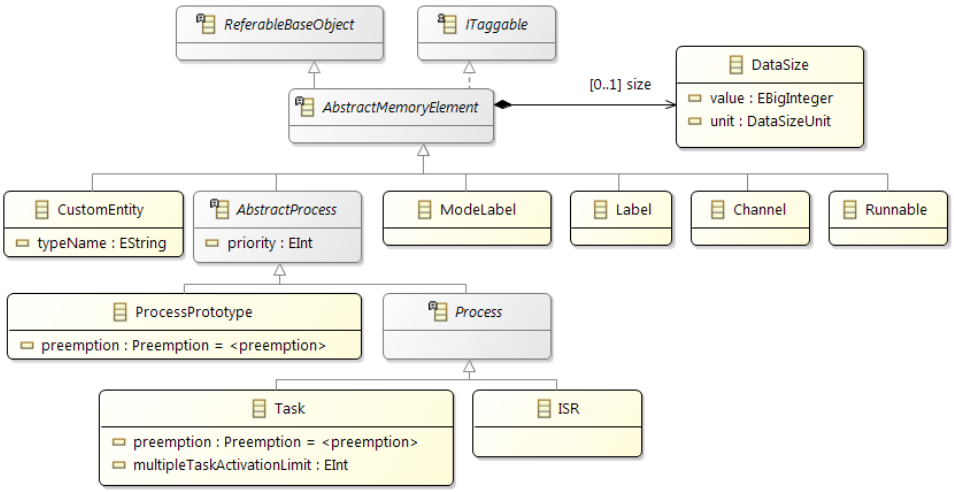


Abbildung 8: Der Aufbau des Softwaremodells mit den Speicherelementen [17]

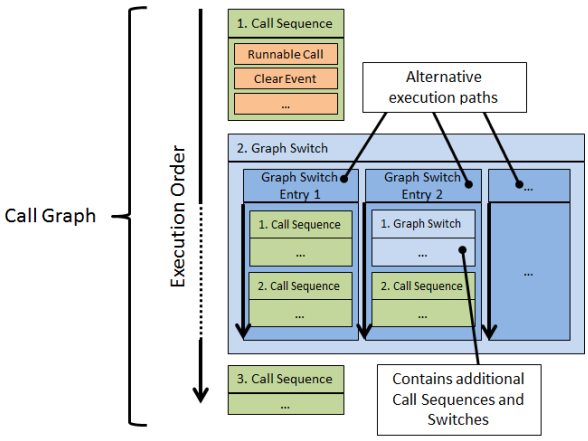


Abbildung 9: Struktur eines Call Graphs [18]

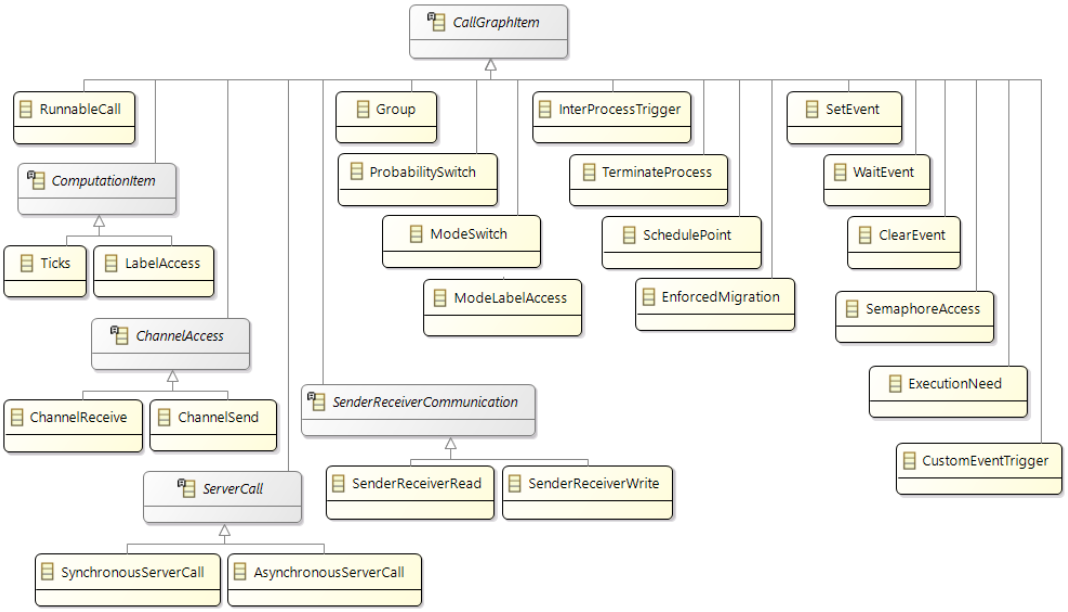


Abbildung 10: Die verschiedenen Elemente eines Call Graphs [19]

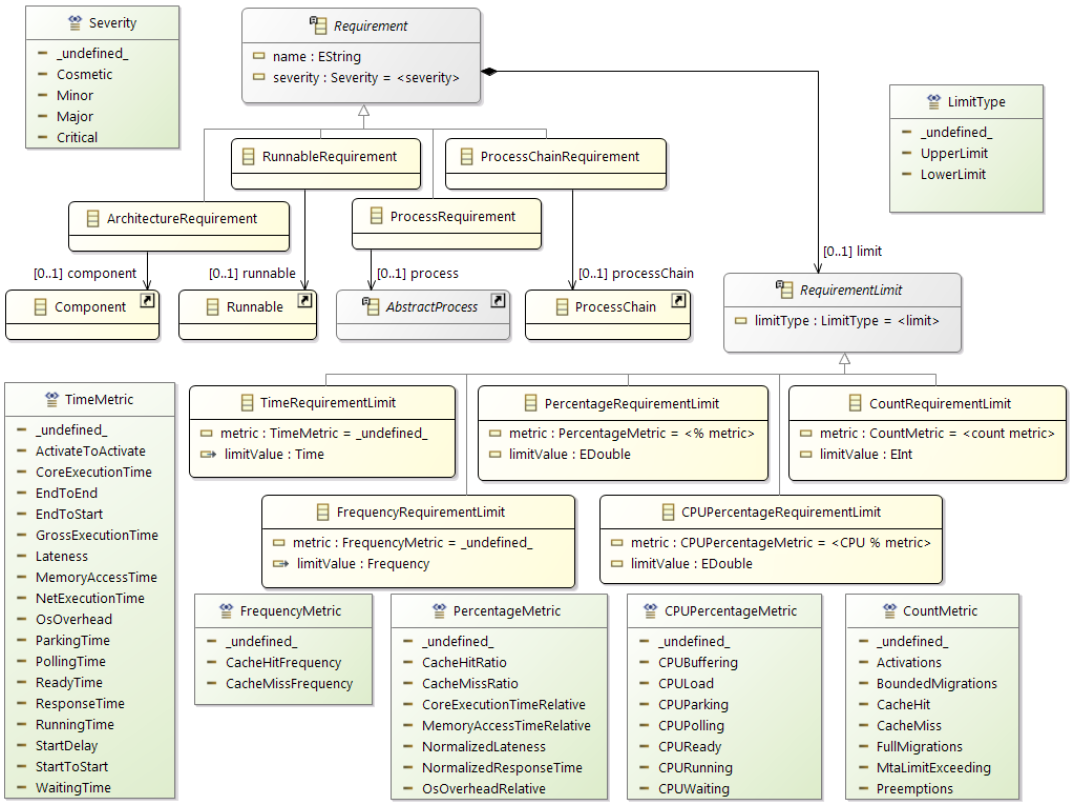


Abbildung 11: Der Aufbau von Anforderungen für Rahmenbedingungen [20]

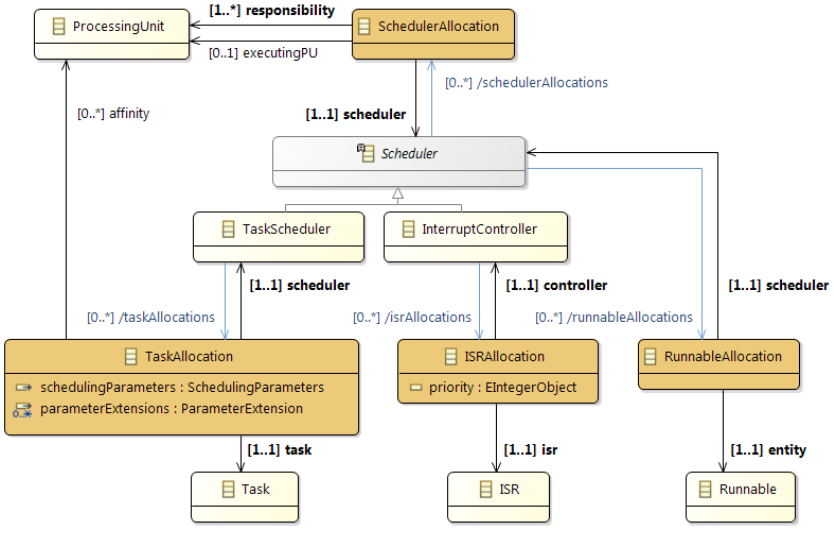


Abbildung 12: Der Aufbau der Allocations für das Mapping [21]

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, 29. Januar 2020

(Dominic Beger)