Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik (FIN)

Masterarbeit

Extraktion von Feature-Modellen aus erweiterten Kconfig-Spezifikationen

Autor:

Ljubica Đorđević

20.01.2026

Betreuer:

Prof. Dr. Gunter Saake, M.Sc. Elias Kuiter
Institut für Technische und Betriebliche Informationssysteme (ITI)
Otto-von-Guericke-Universität Magdeburg

Prof. Dr. Thomas Thüm
Institut für Softwaretechnik und Fahrzeuginformatik (ISF)
TU Braunschweig

Đorđević, Ljubica:
Extraktion von Feature-Modellen aus erweiterten Kconfig-Spezifikationen
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2026.

Zusammenfassung

Das Kconfig-Konfigurationssystem des Linux-Kernels unterstützt dessen hochgradige Konfigurierbarkeit, indem es Features in Kconfig-Spezifikationen kapselt. Dadurch kann Linux an die heterogenen Anforderungen zahlreicher Einsatzbereiche angepasst werden, was wiederum zu seiner Popularität beigetragen hat. Infolgedessen inspirierte Linux weitere Projekte aus der Domäne Systemsoftware (z.B. BusyBox und Zephyr) dazu, das Linux-Konfigurationssystem zu übernehmen oder eine eigene Variante davon umzusetzen. Im Gegensatz zu anderen Systemen definiert Zephyr seine Spezifikationen mittels Kconfiglib und geht dadurch über den Sprachumfang der Kconfig-Sprache hinaus. Diese Erweiterungen von Kconfig führen dazu, dass die Spezifikationen von Werkzeugen wie KConfigReader oder Kclause nicht in aussagenlogische Formeln überführt werden können. Dies schließt Projekte wie Zephyr von weitergehenden Softwareproduktlinienanalysen aus, welche zur Qualitätssicherung essenziell sind.

In dieser Arbeit wird ein Lösungsansatz vorgeschlagen, der die Kconfiglib-basierten Spezifikationen automatisiert in eine annähernd äquivalente Darstellung der Kconfig-Syntax des Linux-Kernels transformiert. Zu diesem Zweck wird zunächst eine systematische Abbildung der Kconfiglib-Erweiterungen auf die Syntax der Linux-Kconfig-Sprache durchgeführt. Daraus werden Transformationsregeln abgeleitet und in Form eines Transformationsprototyps umgesetzt. Anschließend wird der Prototyp auf fünf ausgewählte Kconfiglib-basierte Open-Source-Projekte angewendet, die im Rahmen einer vorangegangenen Recherche identifiziert wurden. Die Evaluation umfasst die Analyse der Verbreitung von Kconfiglib-Erweiterungen, sowie Auswirkungen der Transformation auf die Semantik der erzeugten Kconfig-Spezifikationen.

Inhaltsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1. Einführung

Moderne Softwaresysteme sind zunehmend darauf ausgelegt, heterogene Kundenanforderungen und variierende Hardware-Ressourenbeschränkungen zu adressieren [1]. Variabilität in der Softwareentwicklung ermöglicht es, solche Systeme an unterschiedliche Rahmenbedingungen und individuelle Nutzerbedürfnisse gezielt anzupassen [2; 3; 4]. Insbesondere spielt sie eine zentrale Rolle im Kontext von Softwareproduktlinien [5]. Diese verfolgen die Grundidee, den Entwicklungs- und Wartungsaufwand für einzelne Systemvarianten zu minimieren, indem sie systematische Wiederverwendbarkeit von Softwareartefakten fördern [4; 6]. Dadurch wird ermöglicht, die gewünschte Systemvariante aus wiederverwendbaren Komponenten zu konstruieren, anstatt sie für unterschiedliche Anforderungen neu zu entwickeln. Hierbei erfolgt eine gezielte Auswahl von Konfigurationsoptionen (Features), die in Softwareproduktlinien Eigenschaften des Systems repräsentieren und bei der Variabilitätsmodellierung oft durch Feature-Modelle dargestellt werden [6; 7; 8; 9].

Im Bereich der Open-Source-Projekte ist der Linux-Kernel ein prominentes Beispiel für ein hochgradig konfigurierbares System [10; 11]. Er ist als Feature-orientierte Softwareproduktlinie implementiert und wird von eingebetteten Systemen bis hin zu Supercomputern eingesetzt [12; 13; 14]. Angesichts der Vielzahl seiner Anwendungsszenarien erfordert der Linux-Kernel einen Mechanismus zur Verwaltung und Definition seiner Variabilität. Zu diesem Zweck werden seine Features sowie deren Abhängigkeiten in Kconfig-Spezifikationen festgelegt, die auf der Syntax der domänenspezifischen Sprache Kconfig basieren [15; 14; 16]. Diese Spezifikationsdateien werden vom Linux-Konfigurationssystem und dessen C-basierten Werkzeugen (z.B. `menuconfig`) verarbeitet, um eine gewünschte Variante des Linux-Kernels zu erzeugen [17; 18; 14].

Die Variabilität von Softwareproduktlinien wie dem Linux-Kernel kann verschiedene Probleme verursachen, wie zum Beispiel stetig wachsende und hochkomplexe Inkonsistenzen zwischen der im Variabilitätsmodell vorgesehenen Konfigurierbarkeit und der tatsächlich im Code realisierten Variabilität [19; 20]. Um diesen Herausforderungen zu begegnen, werden in der Literatur typischerweise Produktlinienanalysen herangezogen [21]. Hierbei werden Feature-Modelle häufig in aussagenlogische For-

meln überführt und mithilfe von SAT-Solvern auf ihre Erfüllbarkeit geprüft [22; 23]. Dies ermöglicht zahlreiche weiterführende Analysen auf Basis der Feature-Modelle, die beispielsweise mögliche Produktkonfigurationen, Veränderungen in der Konfigurierbarkeit des Modells oder Gültigkeit einer bestimmten Produktkonfiguration untersuchen [24; 21; 25] oder automatisiertes Testen unterstützen [26; 27]. Eine Überführung der Kconfig-Spezifikationen ermöglicht solche Analysen auch auf dem Linux-Kernel.

Neben Linux hatten zahlreiche weitere Projekte im Bereich der Systemsoftware die Grundideen und Mechanismen zur System-Konfiguration übernommen oder eigene Erweiterungen entwickelt. Hierzu zählen unter anderem BusyBox, Freetz-NG, Toybox und Zephyr.[1] Insbesondere Zephyr wurde von der Linux Foundation als Open-Source-Echtzeitbetriebssystem (RTOS) entwickelt, welches auf den Einsatz in ressourcenbeschränkten Systemen ausgelegt ist und als „kleiner Bruder“ von Linux bekannt ist.[2] Die Grundlage des Konfigurationssystems von Zephyr ist eine vom Linux-Kernel inspirierte Variante des C-basierten Konfigurationssystems, die als Python-basierte Kconfiglib-Bibliothek realisiert ist.[3]

Problemstellung

Linux und die weiteren genannten Projekte stellen Softwareproduktlinien dar, deren Variabilität eine Analyse nahelegt. Im Fall des Linux-Kernels wird dies durch bewährte Werkzeuge wie KConfigReader [28], ConfigFix [29] und Kclause [30] unterstützt. Diese Werkzeuge eignen sich zur Überführung Kconfig-basierter Spezifikationsdateien in aussagenlogische Formeln, auf deren Basis formale Analysen mittels SAT-Solvern durchgeführt werden können. Allerdings sind diese Werkzeuge nicht auf Kconfiglib-basierte Systeme anwendbar, da Kconfiglib die klassische Kconfig-Syntax um eigene Sprachelemente wie `configdefault`- oder `osource`-Optionen erweitert. Diese Ausdrucksmöglichkeiten weichen vom Kconfig-Sprachumfang des Linux-Kernels ab und verhindern eine vollständige Verarbeitung durch bestehende Werkzeuge. Folglich können für Kconfiglib-basierte Systeme wie Zephyr vorhandene Werkzeuge nicht wiederverwendet werden, wodurch automatisierte Analysen solcher Systeme derzeit nicht durchführbar sind.

Zielstellung der Arbeit

Im Rahmen dieser Arbeit wird ein Lösungsansatz entwickelt, der Kconfiglib-basierte Spezifikationsdateien automatisiert in eine annähernd äquivalente Darstellung der Kconfig-Syntax des Linux-Kernels transformiert. Ziel dieser Transformation ist es, einen Vorverarbeitungsschritt zur Extraktion von Feature-Modellen bereitzustellen, der semantisch äquivalente Spezifikationsdateien erzeugt und als Eingabe für Werkzeuge wie KConfigReader oder Kclause dient.

Mit der Wiederverwendbarkeit dieser Werkzeuge soll erreicht werden, dass herkömmliche Analysemöglichkeiten auch für Kconfiglib-basierte Projekte wie Zephyr verfügbar

[1] https://github.com/mirror/busybox; https://freetz.github.io/wiki/; https://landley.net/toybox/; https://www.zephyrproject.org/

[2] https://www.zephyrproject.org/meet-linuxs-little-brother-zephyr-a-tiny-open-source-rtos/

[3] https://docs.zephyrproject.org/latest/build/kconfig/extensions.html

werden. Ein alternativer Ansatz wäre, ein neues Extraktionswerkzeug speziell für die Syntax von Kconfiglib zu entwickeln. Dies wäre nicht nur mit erheblichem Entwicklungsaufwand verbunden, der den Rahmen dieser Arbeit überschreiten würde, sondern steht auch im Widerspruch zur Idee, existierende Werkzeuge wiederzuverwenden. Entsprechend wurde der erste Lösungsansatz gewählt, da er effizienter und nachhaltiger erscheint.

Gliederung der Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert. Kapitel 2 führt den Linux-Kernel ein und erläutert die Funktionsweise von Kconfig. Der Schwerpunkt liegt dabei auf der detaillierten Vorstellung der Kconfig-Sprachelemente, da diese die Grundlage für die angestrebte Transformation bilden. Kapitel 3 analysiert die Funktionsweise sowie Verbreitung von Kconfiglib in verschiedenen Open-Source-Projekten und identifiziert die Erweiterungen im Vergleich zur Kconfig-Sprache. Anschließend wird ein Konzept für die Transformationsregeln ausgearbeitet. Kapitel 4 stellt die Umsetzung der konzipierten Regeln in Form eines Transformationsprototyps vor, Kapitel 5 umfasst eine Evaluierung von vier Forschungsfragen und Kapitel 6 stellt verwandte Arbeiten vor. Abgeschlossen wird die Arbeit mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeiten in Kapitel 7.

2. Grundlagen

Dieses Kapitel führt zunächst den Linux-Kernel ein und legt anschließend den Fokus auf das zugrunde liegende Kconfig-System und dessen Sprachumfang, der wiederum die Grundlage für die Transformation erweiterter Kconfig-Spezifikationen bildet.

2.1 Linux-Kernel

Der Linux-Kernel wurde im Jahr 1991 von Linus Torvalds ins Leben gerufen [31]. Er wurde ursprünglich als kostenfreies Betriebssystem für lediglich eine Architektur (Intel 80386 CPU) konzipiert und von Torvalds selbst als ein reines Hobbyprojekt betrachtet, ohne den Anspruch, eine professionelle Alternative zu etablierten Systemen zu werden [32].[4] Bereits 1993 erreichte der Kernel eine Stabilität, die mit vielen kommerziellen Unix-Systemen konkurrieren konnte [31]. Er entwickelte sich zu einem prominenten Open-Source-Projekt, das von einer globalen Entwicklergemeinschaft aus Freiwilligen und Industriepartnern kontinuierlich weiterentwickelt wird [33]. So haben an der Version 6.18 (November 2025) mehr als 2000 Entwickler mit insgesamt mehr als 13500 Commits aktiv mitgewirkt.[5] Heutzutage ist der Linux-Kernel ein hochgradig konfigurierbares System [10; 11], das in verschiedenen Bereichen (z.B. mobile Betriebssysteme [34]) eingesetzt wird. Zudem ist er als eine feature-orientierte Softwareproduktlinie umgesetzt [12], deren Features in Kconfig-Spezifikationen festgelegt sind [15; 14; 16].

Angesichts der hohen Variabilität von Softwareproduktlinien wie dem Linux-Kernel werden in der Literatur zur Qualitätssicherung typischerweise Produktlinienanalysen herangezogen [21]. Hierbei werden Feature-Modelle häufig in aussagenlogische Formeln überführt und mithilfe von SAT-Solvern auf ihre Erfüllbarkeit geprüft [22; 23]. Auf diese Weise lassen sich mögliche Produktkonfigurationen analysieren, Veränderungen in der Konfigurierbarkeit nachvollziehen oder automatisiertes Testen unterstützen [24; 21; 25; 26; 27]. Um solche Analysen auf Linux anwenden zu können, ist zunächst ein fundiertes Verständnis des zugrunde liegenden Kconfig-Systems erforderlich.

[4] https://www.cs.cmu.edu/~awb/linux.history.html
[5] https://lwn.net/Articles/1046966/

2.2 Kconfig

In der Literatur zum Linux-Kernel wird der Begriff Kconfig häufig sowohl für das Konfigurationssystems des Kernels als auch für die domänenspezifische Sprache zur Definition von Konfigurationsoptionen verwendet [35; 8; 13; 29; 16; 36; 37; 19]. Dies ist durch seine Rolle im Linux-Build-Prozess bedingt, bei der die Kconfig-Sprache und das Kconfig-Konfigurationssystem sehr eng miteinander verknüpft sind und die Konfiguration von Kernel-Varianten gewährleisten.

Im Rahmen dieser Arbeit trennen wir zwischen Linux-Kconfig-Konfigurationssystem und Linux-Kconfig-Sprache, um die Differenz zwischen dem gesamtem Werkzeug und dem Regelwerk deutlich zu machen. Zudem werden die `Kconfig`-Dateien, als Kconfig-Spezifikationen bezeichnet. Abbildung 2.1 visualisiert vereinfacht diese terminologische Einordnung, wobei die dargestellte Bestandteile in den nachfolgenden Abschnitten näher beschrieben werden.

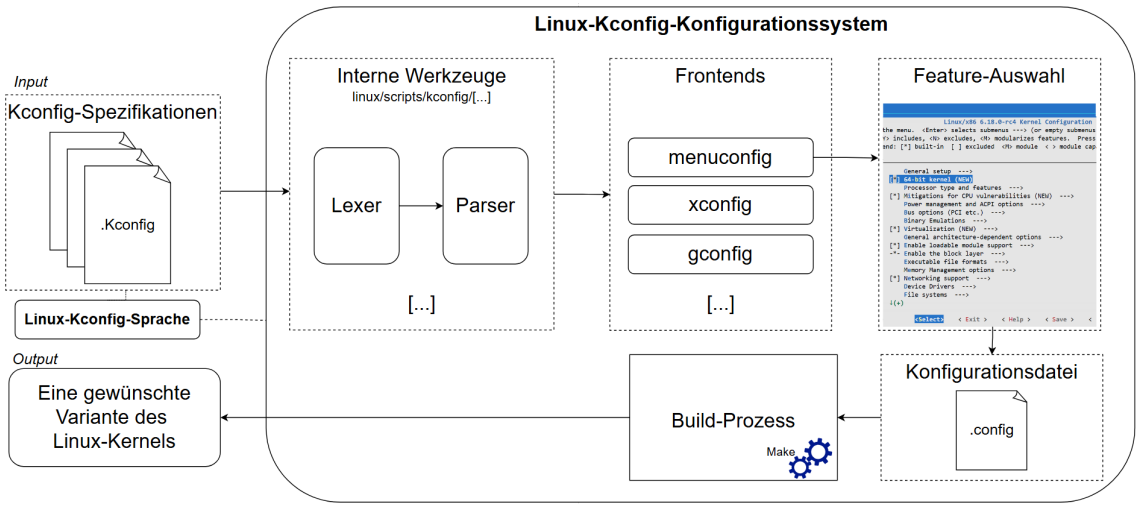

Abbildung 2.1: Terminologische Einordnung von Kconfig

2.2.1 Kconfig-Konfigurationssystem

Das Linux-Kconfig-Konfigurationssystem (auch bekannt als Linux Kernel Configurator (LKC)), wurde als Nachfolger des unübersichtlichen CML1-Konfigurationssystems (engl. *configuration menu language*) und des abgelehnten CLM2-Konfigurationssystems entwickelt [17]. Der LKC führte entscheidende Konzepte ein, die auch in der heutigen Version des Konfigurationssystems zu erkennen sind. Er definiert eine eigene Konfigurationssprache, welche die Konfigurationsoptionen, deren Attribute, Abhängigkeiten und Organisationstrukturen in Form von Kconfig-Spezifikationen abbildet. Diese Spezifikationen werden im Backend unter anderem durch einen Parser verarbeitet. Darauf aufbauend dienen verschiedene grafische Oberflächen und interaktive Kommandozeilen der Auswahl der gewünschten Konfigurationsoptionen. LKC wurde schließlich als vollständige C-Implementierung in der Linux-Kernel-Version 2.5.45 von 2002 integriert [17]. Seitdem wird er auch in vielen weiteren Projekten wie BusyBox, coreboot und Buildroot angewendet [8].[6]

[6] https://busybox.net/; https://www.coreboot.org/; https://buildroot.org/

Das Linux-Kconfig-Konfigurationssystem umfasst eine hierarchisch strukturierte Abfolge von Schritten, die darauf abzielt, eine gewünschte Variante des Linux-Kernels zu erzeugen [18; 19; 38; 39; 14]. Hierfür legen die Kconfig-Spezifikationen gemäß der Syntax der Linux-Kconfig-Sprache die verfügbaren Kernel-Features fest. Die Feature-Auswahl erfolgt über Konfigurations-Frontends wie `menuconfig` und spiegelt sich in der resultierenden `.config`-Datei wider. Diese dient als Eingabe für das Build-System und den C-Präprozessor, die bestimmen, welche Quellcode-Dateien in den Build-Prozess aufgenommen werden und wie die feingranulare Variabilität innerhalb dieser Dateien gesteuert wird. Die vorliegende Arbeit konzentriert sich auf das durch die Kconfig-Spezifikationen definierte Feature-Modell, während der Build-Prozess nicht Gegenstand der Untersuchung ist.

2.2.2 Kconfig-Sprache

Die Linux-Kconfig-Sprache ist eine domänenspezifische Sprache, die das Variabilitätsmodell des Linux-Kernels festlegt [8]. Als Bestandteil des Linux-Kconfig-Konfigurationssystems spezifiziert sie die Features des Kernels als Konfigurationsoptionen (auch Symbole genannt), denen im Konfigurationsprozess ein Wert zugewiesen wird [35; 14]. Diese Optionen sind intern in einer Baumstruktur organisiert, die sich in den Konfigurations-Frontends widerspiegelt [40].

Ein Verständnis der Syntax und Semantik der Linux-Kconfig-Sprache ist grundlegend für die erweiterten Kconfig-Spezifikationen, die im Mittelpunkt dieser Arbeit stehen. Daher werden zunächst die grundlegenden Syntaxregeln und die Definition der `config`-Option erläutert. Darauf folgen weitere Konfigurationsoptionen und deren Attribute. Abschließend werden die zuvor eingeführten Konzepte anhand eines Beispiels verdeutlicht.

Die Darstellung in den folgenden Abschnitten basiert auf der Dokumentation der Linux-Kconfig-Sprache sowie auf einer Untersuchung der Implementierung im Quellcode des Linux-Kernels (Version 6.18) [40].[7] Im Gegensatz zur Dokumentation, welche die Elemente der Kconfig-Spezifikationen als Symbole, Menüeinträge oder Blöcke bezeichnet, verwenden wir hierfür den Begriff *Konfigurationsoption* beziehungsweise *`<Schlüsselwort>`-Option*. Zu einer Konfigurationsoption gehören weitere Syntaxelemente, die deren Eigenschaften festlegen und als *Attribute* bezeichnet werden (vgl. Tabelle 2.1).

Tabelle 2.1: Elemente der Linux-Kconfig-Sprache

Elemente	**Schlüsselwörter**
Konfigurationsoptionen	`mainmenu`, `config`, `menuconfig`, `menu`, `choice`, `comment`, `source`, `if`
Attribute	`type`, `prompt`, `default`, `select`, `imply`, `range`, `depends on`, `help`, `transitional`, `modules`

[7] https://github.com/torvalds/linux/tree/master/scripts/kconfig

Syntax

Die Dokumentation der Linux-Kconfig-Sprache definiert die Basiskonzepte `symbol`, `operator` und `expr`, die in Quelltext 2.1 vereinfacht darstellt sind [40]. Mit dem ersten Konzept hebt die Dokumentation den Unterschied zwischen den Namen von Konfigurationsoptionen (`non-constant symbol`) und allen anderen Bezeichnungen hervor, die in Anführungszeichen eingeschlossenen und als konstante Symbole interpretiert werden (`constant symbol`). Der Name einer Konfigurationsoption wird mit dem Schlüsselwort `(menu)config` eingeführt und besteht aus alphanumerischen Zeichen und Unterstrichen. Ein konstantes Symbol stellt hingegen eine unveränderliche Zeichenkette beliebiger Zeichen dar.

Die Operatoren (`operator`) dienen der Verknüpfung der Symbole innerhalb der Ausdrücken und umfassen sowohl relationale (`=`, `!=`, `<`, `<=`, `>`, `>=`) als auch logische Operatoren (`&&`, `||`). Die Bezeichnung `expr` steht für einen Ausdruck, der entweder ein einzelnes Symbol oder einen Vergleich zweier Ausdrücke mittels relationaler Operatoren umfasst. Außerdem können die Ausdrücke geklammert werden, um die Auswertungsreihenfolge explizit festzulegen, oder auch durch ein Negationszeichen (`!`) verneint werden. Darüber hinaus lassen sich mehrere Ausdrücke mittels logischer Operatoren zu komplexeren Ausdrücken kombinieren.

Quelltext 2.1: Basiskonzepte der Linux-Kconfig-Sprache

```
1 symbol   ::= <non-constant symbol> | <constant symbol>
2 operator ::= <relational operator> | <logical operator>
3 expr     ::= <symbol>
4              | <expr>  <operator>  <expr>
5              | (<expr>)
6              | !<expr>
```

Es ist wesentlich zu verstehen, wie Sichtbarkeit, Wert und Abhängigkeiten von Konfigurationsoptionen bestimmt und beeinflusst werden. Abhängigkeiten legen fest, unter welchen Bedingungen eine Konfigurationsoption gültig ist. Sie können entweder einzelne Attribute einer Konfigurationsoption über eine if-Bedingung einschränken (z.B. `default 5 if CONFIG_A`), oder über das `depends on`-Attribut die Bedingungen definieren, die für sämtliche Attribute der Option gelten. Bei der Finalisierung des Menübaums werden alle festgelegte Abhängigkeiten einer Konfigurationsoption mit den Abhängigkeiten aus übergeordneten `if`- und `menu`-Optionen logisch mittels `AND` verknüpft. Dieser Vorgang wird als Propagation von Abhängigkeit bezeichnet.

Die Sichtbarkeit einer Konfigurationsoption bestimmt, ob sie in Konfigurations-Frontend angezeigt wird. Sie wird durch das `prompt`-Attribut gesteuert, das wiederum nur wirksam ist, wenn alle relevanten Bedingungen erfüllt sind. Das heißt, sowohl die propagierten Abhängigkeiten aus übergeordneten Optionen als auch die eigene if-Bedingung des `prompt`-Attributs müssen erfüllt sein. Der tatsächliche Wert einer Konfigurationsoption entspricht dem vom Linux-Kconfig-Konfigurationssystem berechneten Endwert, der unter Berücksichtigung von Sichtbarkeit, Abhängigkeiten, Standardwerten und Benutzereingaben bestimmt wird.

Die Linux-Kconfig-Makrosprache und die Möglichkeit der Mehrfachdefinition von Konfigurationsoptionen sind weitere Eigenschaften, die unmittelbare Auswirkungen

auf die Modellierung von Konfigurationsoptionen haben. Der interne Aufbau des Linux-Kconfig-Konfigurationssystems und seine Funktionen wie `sym_lookup` stellen sicher, dass basierend auf dem Namen einer neu definierten `config`-Option entweder eine bereits existierende Instanz zurückgegeben oder eine neue angelegt wird.[8] Dadurch wird gewährleistet, dass mehrere `config`-Definitionen derselben Konfigurationsoption stets auf dasselbe Objekt verweisen, statt dass für ein und denselben Namen mehrere unabhängige Objekte entstehen.

Die Linux-Kconfig-Makrosprache dient zur Durchführung von Textsubstitutionen innerhalb von Kconfig-Spezifikationen, bevor die eigentliche Evaluierung der Konfigurationsoptionen durchgeführt wird.[9] Sie unterstützt ausschließlich die Notation `$(VAR)` zum Referenzieren von Variablenwerten. Darüber hinaus können Variablen parametrisiert werden und somit als benutzerdefinierte Funktionen in der Form `$(VAR,arg1,arg2,...,argn)` fungieren. Zudem stellt sie eingebaute Funktionen wie `$(shell,command)` oder `$(info,text)` zur Verfügung, die es ermöglichen, beliebige Shell-Befehle ausführen beziehungsweise Informationen über `stdout` auszugeben.

`config`-Option

Quelltext 2.2 verdeutlicht die formale Syntax der `config`-Optionen. Das Schlüsselwort **`config`** und der eindeutige Name leiten die Definition der Konfigurationsoptionen ein. Darauffolgend bestimmt die Typdeklaration, ob es sich um eine Konfigurationsoption vom Typ int, string, bool, hex oder tristate handelt. Der tristate-Typ kann einen der drei Zustände y, n oder m (für yes, no, oder module) annehmen. Dadurch wird eine Konfigurationsoption entweder in den Kernel integriert, weggelassen oder als ladbares Kernel-Modul bereitgestellt.

Quelltext 2.2: Syntax-Darstellung der `config`-Option

```
number      ::= (<INT> | <HEX>)
config_name ::= <non-constant symbol>
type        ::= int | string | bool | hex | tristate
def_type    ::= def_bool | def_tristate
-----------------------------------------------------------------
config_opt ::= config <config_name> <config_attr>
-----------------------------------------------------------------
config_attr ::= (<type> | <def_type> <value> [if <expr>]
                | <type> "<STRING>" [if <expr>])
                | prompt "<STRING>"       [if <expr>]
                | default <expr>          [if <expr>]
                | depends on <expr>
                | select  <config_name>   [if <expr>]
                | imply   <config_name>   [if <expr>]
                | range <number> <number> [if <expr>]
                | help <MULTILINE STRING>
                | modules
                | transitional
```

[8] https://github.com/torvalds/linux/blob/master/scripts/kconfig/symbol.c
[9] https://docs.kernel.org/kbuild/kconfig-macro-language.html

Das `prompt`-Attribut ist für die Sichtbarkeit der Option in den Konfigurations-Frontends verantwortlich und legt hierfür zugleich den gewünschten Anzeigetext fest. Es wird durch das `prompt`-Schlüsselwort eingeführt, gefolgt von einem Text in Anführungszeichen und gegebenenfalls einer if-Bedingung (vgl. Quelltext 2.2, Zeile 10). Alternativ kann dieser Text in Anführungszeichen direkt in derselben Zeile wie die Typdeklaration angegeben werden (vgl. Quelltext 2.2, Zeile 9), anstatt das `prompt`-Attribut in einer separaten Zeile zu verwenden. Diese Schreibweise wird als `Inline-Prompt` bezeichnet und dient dazu die abweichende Syntax hervorzuheben. Es ist ausschließlich ein `prompt`-Attribut pro Option zulässig. Wenn mehrere `prompt`-Attribute definiert sind, überschreibt die letzte Definition alle vorherigen. Hierbei erscheint eine Warnung im Terminal. Ohne dieses Attribut ist eine Konfigurationsoption für Benutzer nicht sichtbar und ihr Wert kann beispielsweise über Attribute wie `default`, `imply` oder `select` festgelegt werden.

Das `default`-Attribut definiert den Standardwert für eine Konfigurationsoption, der dann übernommen wird, falls Benutzer diesen nicht durch eigene Eingabe überschreiben. Es können mehrere `default`-Attribute für eine Konfigurationsoption definiert werden. Wenn eine `config`-Option mehrfach definiert ist, können auch in all diesen verschiedenen Definitionen Standardwerte spezifiziert werden. In beiden Fällen gilt, dass alle `default`-Werte in der Reihenfolge ihrer Definition berücksichtigt werden und nur der erste Standardwert übernommen wird, dessen Abhängigkeiten erfüllt sind (vgl. Quelltext 2.8, Zeilen 42-44).

Die Abkürzungsnotationen `def_bool` und `def_tristate` ermöglichen es, den Typ und den zugehörigen Standardwert einer Konfigurationsoption in einer einzigen Anweisung zu spezifizieren. Sie setzen voraus, dass der gewünschte Anzeigetext über ein separates `prompt`-Attribut definiert wird, da ein Inline-Prompt in derselben Zeile wie die `def_*`-Anweisung nicht vorgesehen ist.

Das `depends on`-Attribut dient dazu, die Abhängigkeitslogik einer Konfigurationsoption zentral zu verwalten, statt sie bei jedem einzelnen Attribut der Option in der Form einer if-Bedingung zu definieren. Die durch `|` getrennten Seiten des Beispiels 2.3 sind somit äquivalent. Die Verwendung des `depends on`-Attributs sorgt außerdem dafür, die Obergrenze für den Wert einer Option festzulegen. Das heißt, beträgt der Wert von `OPTION_A` `m`, kann die `OPTION_B` entweder auf `n` oder `m` aber nicht `y` gesetzt werden. In dem Fall wenn `OPTION_A` auf `n` gesetzt ist, kann `OPTION_B` nur den Wert `n` annehmen.

Quelltext 2.3: Beispiel für `depends on`-Option

```
1 config OPTION_A             |  config OPTION_A
2   tristate "Option A"       |    tristate "Option A"
3                             <=>
4 config OPTION_B             |  config OPTION_B
5   tristate "Option B"       |    tristate "Option B" if OPTION_A
6   default n                 |    default n if OPTION_A
7   depends on OPTION_A       |
```

Das `select`-Attribut spezifiziert eine sogenannte umgekehrte Abhängigkeit (engl. *reverse dependency*) zwischen zwei Optionen des Typs bool oder tristate. In Quell-

text 2.4 besteht sie zwischen OPTION_C und OPTION_B. Dabei legt das select-Attribut fest, dass OPTION_B als ausgewählte Option mindestens denselben Wert wie OPTION_C annehmen muss. Wesentlich ist, dass diese Wertzuweisung automatisch erfolgt, ohne die in der ausgewählten Option definierten Abhängigkeiten zu berücksichtigen.

Angenommen, OPTION_A hat den Wert n und OPTION_C wird auf den Wert y gesetzt, dann warnt das menuconfig-Frontend vor unerfüllten Abhängigkeiten ("unmet direct dependencies detected for OPTION_B Depends on [n]:OPTION_A[=n] Selected by [y]:OPTION_C[=y]"). Speichert der Benutzer die Feature-Auswahl trotz dieser Warnung, entsteht keine gültige Konfiguration. Sind die Abhängigkeiten von OPTION_B hingegen erfüllt, bestimmt das select-Attribut die Untergrenze des Werts von CONFIG_B. Das heißt, wenn OPTION_C den Wert y hat, kann CONFIG_B keinen niedrigeren Wert als y annehmen, was in menuconfig durch -*- vor dem Prompt-Text von CONFIG_B gekennzeichnet wird.

Quelltext 2.4: Beispiel für select-Option

```
1  config OPTION_A
2    tristate "Option A"
3
4  config OPTION_B
5    tristate "B depends on A"
6    default n
7    depends on OPTION_A
8
9  config OPTION_C
10   def_bool y
11   prompt "Option C"
12   select OPTION_B
```

Das imply-Attribut definiert eine schwache umgekehrte Abhängigkeit (engl. *weak reverse dependency*) zwischen zwei Optionen des Typs bool oder tristate. Es ähnelt der Logik des select-Attributs, da es den Mindestwert der implizierten Option beeinflusst, berücksichtigt aber im Gegensatz die Abhängigkeiten dieser Option.

Um dieses Attribut zu untersuchen, nehmen wir an, in dem vorherigen Beispiel 2.4 würde OPTION_C nicht OPTION_B selektieren, sondern implizieren. Wenn OPTION_A den Wert n hat, ist die depends on-Abhängigkeit von OPTION_B nicht erfüllt und der Wert von OPTION_C hat keinen Einfluss auf den Wert von OPTION_B. Wenn OPTION_A den Wert m oder y hat, ist die Abhängigkeit erfüllt und der Einfluss von OPTION_C auf OPTION_B wird in Quelltext 2.5 veranschaulicht. Dazu wurde der Bereich in zwei Abschnitten (Zeilen 1-3 und 5-7) unterteilt, in denen OPTION_A jeweils auf m beziehungsweise * (d.h. y) gesetzt ist. Der Wert von OPTION_C wird spaltenweise variiert, um den Einfluss auf den Wert von OPTION_B aufzuzeigen.

Eine wesentliche Eigenschaft des imply-Attributs besteht darin, dass es im Gegensatz zu select keinen festen Wert erzwingt. Der durch imply gesetzte Wert kann anschließend auf einen niedrigeren oder höheren Wert angepasst werden, solange dadurch nicht gegen die depends on-Abhängigkeiten verstoßen wird. Beispielsweise

wenn `OPTION_A` den Wert `y` und `OPTION_C` den Wert `m` hat, wird `OPTION_B` zwar automatisch auf `m` gesetzt, kann aber auf `n` oder `y` geändert werden.

Quelltext 2.5: `imply`-Option in `menuconfig`

```
1 <M> Option A        | <M> Option A        | <M> Option A
2 < >  B depends on A | <M>  B depends on A | <M>  B depends on A
3 < > Option C        | <M> Option C        | <*> Option C
4 ------------------------------------------------------------
5 <*> Option A        | <*> Option A        | <*> Option A
6 < >  B depends on A | <M>  B depends on A | <*>  B depends on A
7 < > Option C        | <M> Option C        | <*> Option C
```

Das `range`-Attribut bestimmt den zulässigen Wertbereich für Konfigurationsoptionen des Typs int oder hex. Es stellt sicher, dass nur Werte zugewiesen werden, die größer oder gleich dem ersten sowie kleiner oder gleich dem zweiten in der Attributdefinition angegebenen Wert sind.

Das `help`-Attribut bietet erläuternden Text zu einer Konfigurationsoption und wird im Hilfebereich der Konfigurations-Frontends angezeigt. Dieser Bereich liefert zudem detaillierte Informationen wie den Namen, den aktuellen Wert, die Position in der Linux-Kconfig-Spezifikation sowie den Prompt der aktuell betrachteten Konfigurationsoption. Die Definition dieses Attributs beginnt mit dem Schlüsselwort `help`, gefolgt von mehreren Textzeilen (vgl. Quelltext 2.7, Zeilen 18-19).

Das `modules`-Attribut stellt ein einzigartiges Attribut dar, das im gesamten Konfigurationsraum nur einmal an eine `(menu)config`-Option vergeben werden kann. Die mit diesem Attribut gekennzeichnete Option steuert die modulare Konfigurierbarkeit des Linux-Kconfig-Systems. Sie legt fest, ob der Tristate-Wert `m` für alle anderen Konfigurationsoptionen zur Auswahl stehen soll oder nicht.

Das `transitional`-Attribut dient der Migration von Konfigurationsoptionen, die beispielsweise umbenannt oder ersetzt sein sollten. Wenn zum Beispiel `OPTION_D` durch `OPTION_D_NEW` ersetzt wird, kann `OPTION_D` mit dem Attribut `transitional` ausgezeichnet werden. Dieses Attribut schränkt die Verwendung anderer Attribute ein, was dazu führt, dass `OPTION_D` nur noch eine Typdeklaration ohne Prompt sowie das `help`-Attribut besitzen darf. Durch diese Kennzeichnung wird der Wert von `OPTION_D` aus einer bestehenden `.config`-Datei weiterhin korrekt ausgelesen und kann beispielsweise als Standardwert für `OPTION_D_NEW` verwendet werden. In der neu erzeugten `.config`-Datei wird ausschließlich `OPTION_D_NEW` geschrieben, sodass `OPTION_D` vollständig entfernt wird.

Weitere Konfigurationsoptionen

Nachdem die `config`-Option zusammen mit ihren möglichen Attributen erklärt wurde, werden im Folgenden die weiteren Elemente einer Linux-Kconfig-Spezifikation (`.Kconfig`) vorgestellt. Diese Elemente erfüllen unterschiedliche Funktionen, dienen aber überwiegend dazu, einzelne Konfigurationsoptionen entweder zu spezifizieren oder zu gruppieren [40].

Die `mainmenu`-Option wird ganz am Anfang der Linux-Kconfig-Spezifikation gesetzt, um den Text in der Titelleiste des Konfigurations-Frontends zu spezifizieren. Erst danach folgen weitere Elemente einer Spezifikation (vgl. Quelltext 2.6, Zeile 2).

Quelltext 2.6: Syntax-Darstellung von Elementen der Kconfig-Spezifikation

```
1  mainmenu ::= mainmenu "<STRING>" <elements>
2  elements ::=  config_opt | menuconfig_opt | menu_opt | choice_opt
3              | source_opt | if_opt | comment_opt
4  ---------------------------------------------------------------
5  config_opt     ::= config <config_name> <config_attr>
6  menuconfig_opt ::= menuconfig <config_name> <config_attr>
7  menu_opt       ::= menu "<STRING>"
8                      (visible if <expr> | depends on) <elements>
9                      endmenu
10 choice_opt     ::= choice <choice_att>
11                     (<config_opt> | <comment_opt> | <if_opt>)
12                     endchoice
13 source_opt     ::= source "<path>"
14 if_opt         ::= if <expr> <elements> endif
15 comment_opt    ::= comment "<STRING>" [depends on <expr>]
16 ---------------------------------------------------------------
17 choice_attr ::= prompt "<STRING>"       [if <expr>]
18                 | default <config_name> [if <expr>]
19                 | depends on <expr>
20                 | help <MULTILINE STRING>
```

Die `menuconfig`-Option baut auf die `config`-Option auf und spezifiziert nicht nur eine Konfigurationsoption, sondern zugleich eine Anweisung an das Linux-Kconfig-Konfigurationssystem. Mittels dieser Anweisung wird ein Menüblock eingeleitet, in dem alle von ihm abhängigen Unterpunkte gruppiert und in den Konfigurations-Frontends als eigene, untergeordnete Liste dargestellt werden. Der `menuconfig`-Option können dieselben Attribute wie der `config`-Option zugewiesen werden.

Die `menu`-Option dient im Gegensatz zu `menuconfig` rein der Organisation anderer Elemente der Kconfig-Spezifikation und deren Gruppierung innerhalb eines Untermenüs in den Konfigurations-Frontends. Alle Optionen innerhalb des `menu`- und `endmenu`-Schlüsselworts erben die Abhängigkeiten der übergeordneten `menu`-Option. Die `menu`-Option kann durch die Attribute `depends on` und `visible if <expr>` gekennzeichnet werden, wobei das letzte die Sichtbarkeit des Menüblocks in den Konfigurations-Frontends beeinflusst.

Die `choice`-Option gruppiert mehrere Konfigurationsoptionen und stellt sicher, dass genau eine davon aktiviert wird. Das bedeutet, dass der Benutzer zwingend eine der innerhalb von `choice` und `endchoice` definierten Optionen auswählen muss. Eine `choice`-Option kann zudem die Attribute `prompt`, `default`, `depends on` und `help` annehmen. Dabei ist eine Typdefinition der `choice`-Option nicht mehr vorgesehen,

da alle enthaltenen `config`-Optionen ausschließlich vom Typ bool sind.[10] Zudem müssen die Optionen innerhalb der `choice`-Option zwingend ein `prompt`-Attribut besitzen und dürfen kein `default`-Attribut enthalten. Falls eine dieser Optionen auch außerhalb der `choice`-Option definiert ist, darf sie dort kein `prompt`-Attribut besitzen.

Die `comment`-Option definiert keine Konfigurationsoption, sondern stellt statischen Text bereit, der als Hilfestellung oder Erklärung innerhalb der Konfigurations-Frontends dient. Dieser Text erscheint ausschließlich in der Benutzeroberfläche der Frontends und beeinflusst die aus der Feature-Auswahl resultierenden `.config`-Datei nicht.

Die `source`-Option nimmt einen Dateipfad zu der einzubindenden Spezifikation als Parameter an und dient der modularen und hierarchischen Strukturierung der Linux-Kconfig-Spezifikationen. Beim Einlesen der angegebenen Spezifikation wird ein neuer Pufferzustand erzeugt, der sowohl die aktuellen Dateiinformationen als auch den Verweis auf die übergeordnete Spezifikation enthält. Dadurch kann nach der Verarbeitung der eingebundenen Spezifikation zum vorherigen Kontext zurückgekehrt werden. Gleichzeitig wird sichergestellt, dass zyklische Einbindungen der Spezifikationen erkannt und verhindert werden.

Die `if`-Option wird für die zentrale Verwaltung der Abhängigkeitslogik mehrerer Optionen gleichzeitig verwendet. Mit den Schlüsselwörtern `if` und `endif` werden die gewünschten Optionen zu einem Block zusammengefasst, auf den die im `if`-Ausdruck definierten Abhängigkeiten angewendet werden. Dadurch ist es nicht erforderlich, für jede einzelne Option dieses Block ein eigenes `depends on`-Attribut zu definieren, um dieselbe Abhängigkeitslogik sicherzustellen. Dies trägt wiederum zu einer wartungsfreundlichen Kconfig-Spezifikation bei. Im Gegensatz zur `menu`-Option hat die `if`-Option keinen Einfluss auf die Gruppierung von Optionen zu einem Untermenü in den Konfigurations-Frontends.

Konfigurationsbeispiel

Die nachfolgende Linux-Kconfig-Spezifikation wurde erstellt, um exemplarisch die zentralen Sprachelemente zu illustrieren. Sie wird aus visuellen Gründen in die Quelltexte 2.7 und 2.8 unterteilt. Quelltext 2.7 umfasst die Optionen `source`, `comment`, `choice`, `config`, `if` und deren zugehörigen Attribute. Quelltext 2.8 betrachtet die Optionen `menu` und `menuconfig`.

In Quelltext 2.7 bewirkt die `source`-Anweisung, dass die Konfigurationsoptionen aus dem `utils`-Verzeichnis eingelesen werden. Dadurch werden die boolesche Konfigurationsoption `UTILS_SYS_BS` und die Ganzzahloption `SCREEN_SIZE` bereitgestellt. Diese sind funktional erforderlich, um die Abhängigkeitsbedingungen weiterer Optionen dieser Spezifikation erfüllbar zu machen.

Die `comment`-Option hinterlegt einen textuellen Hinweis bezüglich des thematischen Bereichs der Kconfig-Spezifikation und ist nur sichtbar, wenn `UTILS_SYS_BS` aktiv ist. Die darauffolgende `choice`-Option für die Auswahl des Anzeigemodus erzwingt

[10] https://github.com/torvalds/linux/commit/fde192511bdbff554320b31574bb8a9cb3275522; https://github.com/torvalds/linux/commit/bea2c5ef789a37bace99f2f45eeef3be4559b228

eine Entscheidung zwischen hellem oder den benutzerdefiniertem Modus. Dabei ist diese Option mit vier Attributen versehen und umfasst die zwei booleschen Auswahloptionen `LIGHT_MODE` und `USER_DEF_MODE`. Diese Auswahloptionen haben jeweils über die Attribute `imply` und `select` unmittelbare Auswirkung auf andere Optionen, indem sie deren Mindestwert setzen beziehungsweise erzwingen.

Die in den Zeilen 22-28 (Quelltext 2.7) spezifizierte `if`-Option bindet die beiden internen Konfigurationsoptionen konditional ein. Wenn der helle Modus nicht ausgewählt wurde, wird die boolesche Option `POWER_SAVING_MODE` standardmäßig deaktiviert, während der Wert von `BACKLIGHT_CONTROL` standardmäßig auf `m` gesetzt wird.

Quelltext 2.7: Beispiel einer hypothetischen Kconfig-Datei

```
1  source "drivers/utils/Kconfig"
2  comment "Auswahl des Anzeigemodus"
3    depends on UTILS_SYS_BS
4
5  choice
6    prompt "Anzeigemodus"
7    default LIGHT_MODE
8    depends on UTILS_SYS_BS
9
10   config LIGHT_MODE
11     bool "Heller Modus"
12     imply TEXT_COLOR_BLACK
13
14   config USER_DEF_MODE
15     bool "Benutzerdefinierter Modus"
16     select SCREEN_CUSTOM_COLOR
17     help
18       Wählen Sie diese Option, um individuelle
19       Anpassungen zu ermöglichen
20 endchoice
21
22 if !LIGHT_MODE
23   config POWER_SAVING_MODE
24     def_bool n
25
26   config BACKLIGHT_CONTROL
27     def_tristate m
28 endif
```

In Quelltext 2.8 bündelt die `menu`-Option mit der Überschrift `Bildschirmoptionen` die thematisch verwandten Optionen, die ihre `depends on`-Abhängigkeit erben. Die erste definierte depends on-Abhängigkeit hängt von dem Wert einer Option ab, während die zweite Abhängigkeit erfüllt ist, wenn ein `python3`-Interpreter auf dem System gefunden wurde.

Die Konfigurationsoption `APP_DEFAULT_SCREEN_SIZE` definiert mehrere Standardwerte, die von der angegebenen Bildschirmgröße (`SCREEN_SIZE`) abhängen. Wird die Bildschirmgröße auf zehn gesetzt, sind die if-Bedingungen der ersten beiden `default`-Attribute erfüllt. Der Standardwert wird aber auf acht gesetzt, da die Reihenfolge der Attribute entscheidend ist.

Die Option `SCREEN_CUSTOM_COLOR` spezifiziert eine boolesche `menuconfig`-Option, die in den Konfigurations-Frontends nur sichtbar ist, wenn der Benutzer zuvor `USER_DEF_MODE` statt `LIGHT_MODE` ausgewählt hat (vgl. Quelltext 2.7). Wenn diese `menuconfig`-Option den Wert `y` annimmt, werden die zugehörige Zeichenkettenoption `SCREEN_COLOR_PICKER` sowie die Ganzzahloption `NR_PIXEL` in den Konfigurations-Frontends sichtbar. Der zulässige Wertebereich dieser Ganzzahloption liegt zwischen 140 und 1440. Ihr Standardwert wird aus der Umgebungsvariable `ENV_PIXEL` ausgelesen.

Quelltext 2.8: Beispiel der hypothetischen Kconfig-Datei - Fortsetzung

```
29 menu "Bildschirmoptionen"
30   depends on UTILS_SYS_BS
31   depends on "$(shell, python3 -V)" != ""
32
33 config APP_DEFAULT_SCREEN_SIZE
34   int
35   default 8  if SCREEN_SIZE >= 8 && SCREEN_SIZE <= 10
36   default 10 if SCREEN_SIZE >= 10
37   default 5
38
39 menuconfig SCREEN_CUSTOM_COLOR
40   bool
41   prompt "Monitorfarbe"
42
43 config SCREEN_COLOR_PICKER
44   string "Benutzerdefinierter Farbwert in Format #RRGGBB"
45   depends on SCREEN_CUSTOM_COLOR
46   default "E5CCFF"
47
48 config NR_PIXEL
49   int "Maximale gewünschte Pixel-Anzahl"
50   depends on SCREEN_CUSTOM_COLOR
51   range 144 1440
52   default $(ENV_PIXEL)
53 endmenu
```

2.2.3 Zusammenfassung

Im diesem Kapitel wurde eine terminologische Trennung zwischen dem Linux-Kconfig-Konfigurationssystem und der Linux-Kconfig-Sprache vorgenommen. Der Fokus lag auf der Linux-Kconfig-Sprache, da ein fundiertes Verständnis ihrer Syntax und Semantik eine wesentliche Voraussetzung für die nachfolgenden Untersuchungen erweiterter Kconfig-Spezifikationen darstellt. Die Sprachelemente wurden in Konfigurationsoptionen und zugehörige Attribute gegliedert und systematisch erläutert. Ergänzend wurde ihre Syntax in Form einer informellen BNF-Notation (Backus-Naur-Form) zusammengefasst. Abschließend veranschaulichte ein Beispiel die vorgestellten Sprachelemente innerhalb einer Kconfig-Spezifikation.

3. Konzept

Im Mittelpunkt dieses Kapitels steht die Kconfiglib-Bibliothek, die im Vergleich zur Linux-Kconfig-Sprache erweiterte Sprachelemente aufweist. Diese Sprachelemente spielen eine zentrale Rolle in dieser Arbeit und werden im Folgenden als *Kconfiglib-Erweiterungen* bezeichnet. Das Kapitel gliedert sich in fünf Unterkapitel. Das erste Unterkapitel analysiert die Motivation sowie die Funktionsweise von Kconfiglib. Darauf aufbauend wird im nächsten Unterkapitel untersucht, in welchem Umfang die Bibliothek in der Praxis eingesetzt wird. Zudem wird analysiert, ob die betrachteten Projekte eigene Anpassungen oder weitere Kconfiglib-Erweiterungen einführen. Das dritte Unterkapitel stellt die identifizierten Kconfiglib-Erweiterungen systematisch vor. Im vierten Unterkapitel werden diese Erweiterungen auf die Syntax der Linux-Kconfig-Sprache abgebildet und daraus die Transformationsregeln abgeleitet. Die zentralen Ergebnisse fasst ein abschließendes fünftes Unterkapitel zusammen.

3.1 Kconfiglib

Es wird die eigenständig durchgeführte Recherche zur Kconfiglib-Bibliothek vorgestellt. Im Fokus stehen ihre Entstehung und ihre grundlegende Funktionsweise, um eine fundierte Einführung in Kconfiglib zu schaffen.

3.1.1 Entstehung

Kconfiglib ist eine Python-Bibliothek, die im Jahr 2011 im Rahmen der Abschlussarbeit von Ulf Magnusson entstand [41]. Diese Arbeit befasst sich mit der Entwicklung eines Linux- und Web-basierten Betriebssystems namens Awesom-O, das durch eine kurze Bootzeit und einen geringen Speicherbedarf charakterisiert ist. Die Analyse des Systemstarts zeigte, dass der Bootloader und der Kernel die Hauptverzögerung verursacht haben. Daher konzentrieren sich die vorgestellten Optimierungsmaßnahmen auf die Minimierung der Linux-Kernel-Konfiguration. Zu diesem Zweck wurde ein Verfahren implementiert, das Linux-Kerneloptionen systematisch deaktiviert, den Kernel anschließend erneut kompiliert und überprüft, ob das System startfähig ist und grundlegende Netzwerkfunktionen ausführen kann. Darauf basierend wurden die

einzelnen Linux-Kerneloptionen entweder als unnötig eingestuft oder als wesentlich identifiziert und wieder in der Konfiguration aktiviert. Dieses Verfahren spiegelt sich im sogenannten Kernel-Minimizer wider, der als Python-Skript umgesetzt wurde und auf Kconfiglib zurückgreift. Kconfiglib übernimmt dabei das Einlesen der Kconfig-Spezifikationen, die konsistente Neubewertung von Konfigurationsoptionen und die Verwaltung ihrer Abhängigkeiten.

Obwohl Kconfiglib ursprünglich als Python-Interpreter zur Unterstützung des Kernel-Minimizers vorgesehen war, entwickelte es sich schnell zu einer eigenständigen Open-Source-Bibliothek [41].[11] Diese Entwicklung war der Entscheidung des Autors geschuldet, einen neuen Interpreter zu implementieren, anstatt die bestehende C-Implementierung des Linux-Kconfig-Konfigurationssystem zu erweitern. Die Entscheidung wurde damit begründet, dass der gewählte Ansatz einfacher erschien. Rückblickend erwies sich die Umsetzung als deutlich aufwendiger, führte aber zur Entstehung eines eigenständigen Analysewerkzeugs für Kconfiglib-basierte Konfigurationssysteme.

3.1.2 Funktionsweise

Die gesamte Kernfunktionalität der Kconfiglib-Bibliothek ist in einer einzigen Python-Datei (`kconfiglib.py`) enthalten. Während Kconfiglib darauf abzielt, das Verhalten der C-Implementierung nachzubilden, erweitert sie ihren Funktionsumfang unter anderem durch `source`-Anweisungen mit Unterstützung für `glob`-Muster sowie durch zusätzliche Schlüsselwörter wie `def_string`.[12] Die auf Kconfiglib basierenden Dateien werden im Rahmen dieser Arbeit als Kconfiglib-Spezifikationen bezeichnet, um den Unterschied zu den Linux-Kconfig-Spezifikationen hervorzuheben.

Die Kconfiglib-Bibliothek dient der Automatisierung von Konfigurations- und Analyseprozessen. Sie ist sowohl mit Python 2.7 als auch Python 3.2 und höher kompatibel und ermöglicht einen strukturierten Zugriff auf die Informationen der Kconfiglib-Spezifikationen.[13] Darüber hinaus erlaubt sie die Erstellung von Skripten zur automatisierten Extraktion, Manipulation und Generierung dieser Spezifikationen. Dadurch entsteht ein wesentlicher Vorteil, dass eine manuelle Verwendung von Konfigurations-Frontens wie `menuconfig` entfällt.

Die vier zentralen Klassen der Bibliothek modellieren gemeinsam den vollständigen Konfigurationsraum, darunter `Kconfig`, `Symbol`, `Choice` und `MenuNode`.[12] Im Folgenden werden die Hauptaufgaben dieser Klassen sowie ihre Kernmechanismen anhand einer Untersuchung des Quellcodes der Kconfiglib-Bibliothek dargestellt. Dabei ist das Ziel, ein grundlegendes Verständnis der Funktionsweise der Bibliothek zu vermitteln.

[11] https://github.com/ulfalizer/Kconfiglib

[12] https://github.com/ulfalizer/Kconfiglib/blob/master/kconfiglib.py

[13] https://github.com/ulfalizer/Kconfiglib?tab=readme-ov-file#python-version-compatibility-2-7-3-2

Klasse `Kconfig`

Die Klasse stellt die zentrale Komponente der Kconfiglib-Bibliothek dar. Sie beschreibt den gesamten Konfigurationsraum eines Projekts zu einem bestimmten Zeitpunkt und übernimmt die Konfigurationsverwaltung, Konsistenzprüfung und Fehlerbehandlung als ihre Hauptaufgaben. Dabei erlauben ihre öffentlichen Methoden und Attribute den Zugriff auf sämtliche Aspekte der zugrunde liegenden Kconfiglib-Spezifikationen. Die Attribute erfassen die Mengen verschiedener Elemente der Kconfiglib-Spezifikationen, die eindeutigen Listen der `config`-Optionen und der `choice`-Optionen, sowie verschiedene Fehler- und Warnmechanismen. Ergänzend steuern die öffentlichen Methoden nicht nur das Einlesen und das Schreiben der aktuellen Werte einer `.config`-Datei, sondern bieten auch direkte Zugriffsmöglichkeiten auf Ausdrucksevaluierung sowie Kontrolle über Konsistenzprüfungen und Diagnoseausgaben.

Jede Instanz der Klasse `Kconfig` kapselt den vollständigen Konfigurationsraum eines Projekts und kann isoliert verwendet werden. Sie übernimmt unter anderem die grundlegende Initialisierung der Umgebungseinstellungen und die Erstellung der erforderlichen Datenstrukturen für die Informationsspeicherung. Das heißt, sie setzt Eigenschaften des Konfigurationsraums wie die Zeichencodierung oder das Wurzelverzeichnis (`srctree`) und speichert Informationen der Kconfiglib-Spezifikationen wie die Menge aller Optionen (`syms`) oder aller konstanten Optionen (`const_syms`). Darauffolgend parst sie die Kconfiglib-Spezifikationen und erstellt die interne Struktur des Konfigurationsraums in Form eines Menübaums. Dabei werden die geparsten Einträge als Menüknoten an den Wurzelknoten des Menübaums angehängt. Abschließend werden Konsistenzprüfungen durchgeführt sowie die Abhängigkeiten zwischen den Menüknoten vereinfacht und propagiert, um den zusammengestellten Menübaum zu finalisieren und validieren.

Klasse `MenuNode`

Die Klasse stellt eine grundlegende Datenstruktur bereit, um die Einheiten der Kconfiglib-Spezifikationen als Knoten des Menübaums darzustellen. Jede `MenuNode`-Instanz ermöglicht den Zugriff auf eigene Eltern-, Kind- und Nachbarknoten, wodurch ihre Position innerhalb der Baumstruktur bestimmt wird. Das `item`-Attribut eines `MenuNode` verweist auf den eigentlichen Inhalt des Knotens, wobei es sich um Optionen `config`, `menuconfig`, `choice`, `menu` oder `comment` handelt. Dies unterstützt die Mehrfachdefinition von `config`-Optionen und `named choice`-Optionen (vgl. Klasse `Choice` 3.1.2). Wird eine `config`-Option an mehreren Stellen im Konfigurationsraum definiert, so existiert eine `MenuNode`-Instanz pro Definition, die in ihrem `item`-Attribut auf dieselbe `config`-Option verweist.

Jede Instanz der Klasse `MenuNode` repräsentiert einen Knoten im Menübaum und kapselt dessen Anzeige-, Kontext- und Abhängigkeitsinformationen. Dazu gehören Eigenschaften wie der Anzeigetext (`prompt`) und der Hilfetext (`help`). Weitere Attribute einer `MenuNode`-Instanz referenzieren den Konfigurationsraum (`kconfig`), in dem sich der Knoten befindet und erfassen darüber hinaus den genauen Definitionsort (`filename, linenr`) sowie die Include-Hierarchie der `source`-Optionen (`include_path`). Die öffentlichen Methoden der Klasse ermöglichen eine strukturierte Repräsentation und Ausgabe eines `MenuNode`, entweder in einer kompakten,

informationsorientierten Form zur Analysezwecken oder als textuelle Darstellung des Knoteninhalts in Kconfiglib-Syntax.

Klasse `Symbol`

Die Klasse dient der internen Repräsentation von `config`- und `menuconfig`-Optionen innerhalb einer Kconfiglib-Spezifikation. Sie kapselt die semantischen Eigenschaften dieser Konfigurationsoptionen und modelliert dessen Konfigurationszustand, einschließlich Typ, Wert, Sichtbarkeit und Beziehungen zu anderen Optionen. Der effektive Wert einer Option wird dynamisch ermittelt, wobei die Benutzereingaben, Standardwerte und sämtliche aus übergeordneten `menu`- und `if`-Optionen propagierten Abhängigkeiten berücksichtigt werden. Darüber hinaus verwaltet diese Klasse über das Attribut `assignable`, wann und auf welchen Wert eine Option gesetzt werden darf, basierend auf ihrer Sichtbarkeit und ihren Abhängigkeiten (vgl. Syntax 2.2.2). Zudem pflegt das Attribut `nodes` dieser Klasse eine Liste von `MenuNodes`-Instanzen, die sämtliche Definitionen einer Option im Menübaum abbilden.

Klasse `Choice`

Die Klasse `Choice` dient der Modellierung von `choice`-Optionen. Sie ist mit der `Symbol`-Klasse strukturell eng verwandt und weist weitgehend ähnliche Attributnamen und vergleichbare Zugriffsmöglichkeiten auf. Dies trägt zur Einheitlichkeit der internen Struktur der Kconfiglib bei und erlaubt gleichzeitig eine klare funktionale Trennung zwischen beiden Konzepten. Eine Besonderheit der Kconfiglib besteht darin, dass `choice` zusätzlich mit einem Namen versehen werden kann. Dadurch entsteht eine sogenannte `named choice`-Option, die an mehreren Stellen definiert werden kann, wobei jede Definition die Liste der verfügbaren Auswahloptionen erweitert (vgl. `named choice`-Option 3.3.2).

3.2 Verbreitung von Kconfiglib

Die Relevanz der vorliegenden Untersuchung liegt darin, dass sie Aufschluss über die Nutzung von Kconfiglib in Open-Source-Projekten gibt. Ziel ist es, zu ermitteln, welche Projekte diese Bibliothek verwenden, welche projektspezifischen Anpassungen insbesondere an der Kconfiglib-Sprache vorgenommen wurden und aus welchen Gründen sich die Projekte für den Einsatz dieser Bibliothek entschieden haben. Die Ergebnisse zeigen Anwendungsszenarien für den angestrebten Transformationsprototyp auf und bilden zugleich eine Grundlage für das folgende Kapitel, in dem die ermittelten Erweiterungen der Kconfiglib-Sprache vertieft betrachtet werden.

3.2.1 Methodik

Als Ausgangspunkt der Untersuchung dient das GitHub-Repository von Kconfiglib.[11] Zur Ermittlung der Projektbeziehungen wurden die `Forks` und das Abhängigkeitsdiagramm des GitHub-Repository sowie die in der Repository-Beschreibung referenzierten Projekte berücksichtigt. Der `Forks`-Bereich umfasst alle öffentlichen Kopien des Repository. Das Abhängigkeitsdiagramm listet sowohl die vom Repository benötigten Abhängigkeiten als auch öffentliche Repositories und Pakete auf, die Kconfiglib

verwenden. Dabei sind letztere für die folgende Analyse insbesondere von Bedeutung. Um diesen Überblick bereitzustellen, analysiert GitHub automatisch direkte und transitive Abhängigkeiten auf Grundlage von Dateien wie `requirements.txt` oder `setup.py` und aktualisiert das Abhängigkeitsdiagramm entsprechend.[14]

In diesen GitHub-Bereichen wurde eine manuelle Analyse der Einträge durchgeführt, gefolgt von einer eigenen Kategorisierung dieser Einträge. Im `Forks`-Bereich wird zwischen Projekten privater GitHub-Nutzer und solchen von Organisationskonten unterschieden. Im Abhängigkeitsdiagramm wird jedes Repository einer der vier Kategorien `Großprojekt`, `Unterprojekt`, `akademisches Projekt` oder `privates Projekt` zugeordnet. Ein Großprojekt bezeichnet ein aktives Softwareprojekt mit hoher Anzahl an Sterne-Markierungen, vielen Mitwirkenden und in der Regel einer Verwaltung über ein Organisations- oder Unternehmenskonto. Ein Unterprojekt ist von einem größeren Projekt abgeleitet, meistens in Form eines Forks dieses Projekts. Ein akademisches Projekt bezieht sich auf Repository, dass Code zu veröffentlichten wissenschaftlichen Arbeiten enthält. Kleinere, von einzelnen GitHub-Nutzern erstellte Repositories, die meistens nur zum Testen oder experimentellen Zwecken dienen, wurden der Kategorie privates Projekt zugeordnet.

Für die finale Auswertung wurden aus dem `Forks`-Bereich die Projekte von Organisationskonten sowie aus dem Abhängigkeitsdiagramm die Kategorien `Großprojekt` und `akademisches Projekt` herangezogen. Die `Unterprojekte` sind durch die Kategorie `Großprojekt` abgedeckt und somit von detaillierten Analysen ausgeschlossen. Die privaten Projekte wurden ausgeschlossen, da sie in der Regel den aktuellen Entwicklungsstand von Kconfiglib widerspiegeln und nur wenige oder keine eigenen Commits aufweisen. Die GitHub-Repositories der final ausgewählten Projekte wurden hinsichtlich ihres Quellcodes und ihrer Release-Historie untersucht, um die jeweilige Verbindung zu Kconfiglib sowie mögliche Weiterentwicklungen festzustellen. Ergänzend wurden die offiziellen Webseiten der identifizierten Projekte analysiert, um den Anwendungskontext und mögliche Hinweise auf die Entscheidung für Kconfiglib zu ermitteln. Wenn diese Webseiten hierzu keine Informationen geliefert haben, erfolgte eine Kontaktaufnahme mit den Maintainer über projektspezifische Foren, Discord-Channels oder E-Mail.

3.2.2 Ergebnisse

Das Kconfiglib-GitHub-Repository weist über 160 Forks und mehr als 1400 von ihm abhängige Repositories und Pakete auf.[11] Laut GitHub-Informationshinweisen handelt es sich hierbei um Schätzwerte, die von der tatsächlichen Anzahl der Einträge in den Listenansichten abweichen können. Zudem verweist die Repository-Beschreibung explizit auf die Verwendung von Kconfiglib in drei Projekten Zephyr, esp-idf und ACRN.

Die eigene Analyse zeigt, dass von 167 Forks nur 40 und von 1437 abhängigen Repositories und Paketen lediglich 734 öffentlich sichtbar sind (Stand 24.11.2025).[15]

[14] https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph

[15] https://github.com/ulfalizer/Kconfiglib/network/dependents; https://github.com/ulfalizer/Kconfiglib/forks

Insgesamt wurden somit 774 öffentlich sichtbare Projekte aus beiden GitHub-Bereiche in die Analyse einbezogen. Es ist zu klären, welches Ziel die jeweiligen Projekte verfolgen, ob sie Kconfiglib aktiv verwenden oder weiterentwickeln und welche Motivation der Auswahl von Kconfiglib zugrunde liegt.

`Forks`-Bereich

Von den 40 aufgelisteten Repositories können 37 privaten GitHub-Nutzern und drei den Organisationskonten Stabl-Energy, RIOT-OS und legatoproject zugeordnet werden.[16] Das Repository von Stabl-Energy gehört dem gleichnamigen deutschen Unternehmen, das auf die Wechselrichtertechnologie für Batteriespeicher spezialisiert ist.[17] Das Repository weist lediglich wenige eigene kleine Anpassungen von Kconfiglib auf, wobei die letzte vor rund einem Jahr erfolgte. Da keine zusätzlichen Informationen öffentlich verfügbar sind und auch nicht durch Kontaktaufnahme erlangt werden konnten, wird dieses Repository als privates Projekt des Unternehmens eingestuft und nicht weiter betrachtet. Im Gegensatz dazu handelt es sich bei RIOT und legato-af jeweils um ein Open-Source-Betriebssystem beziehungsweise ein Open-Source-Framework für die Entwicklung von Anwendungen im Bereich des Internet-of-Things (IoT).[18]

RIOT

RIOT ist ein Open-Source-Betriebssystem für Mikrocontroller, das darauf ausgelegt ist, eine Software-Plattform für heterogene eingebettete Geräte bereitzustellen und neue Standards für diese zu prägen.[19] Es legt den Fokus auf eine Linux-ähnliche Open-Source-Community sowie auf Interoperabilität, Sicherheit und technische Merkmale, die unter anderem für IoT-Geräte zugeschnitten sind. Seine aktuelle Version (2025.07) zeichnet sich zudem durch Echtzeitfähigkeit, Energieeffizienz, partielle POSIX-Kompatibilität, Modularität und konsistenten API-Zugriff über alle unterstützten Architekturen hinweg aus.[20]

Die projektspezifischen Anpassungen von Kconfiglib im RIOT-Projekt manifestieren sich in der Klasse `RiotKconfig`.[21] Sie erbt von den Kconfiglib-Klassen `Kconfig` und `KconfigError`, überschreibt aber die Methoden `_parse_help` und `write_autoconf`. Konkret werden Doxygen-Markierungen aus den Hilfetexten der `help`-Attribute entfernt, während die angepasste `write_autoconf`-Methode sicherstellt, dass die erzeugten C-Präprozessor-Makros keine Bindestriche in Optionsnamen enthalten. Diese Anpassungen betreffen somit die technische Überführung in für das Build-System relevante Artefakte und nicht das durch die Kconfiglib-Spezifikationen definierte Feature-Modell. Da der Build-Prozess nicht Gegenstand der Untersuchung ist, werden diese Anpassungen nicht weiter berücksichtigt. Erweiterungen der Syntax der

[16] https://github.com/Stabl-Energy/Kconfiglib; https://github.com/RIOT-OS/RIOT; https://github.com/legatoproject/legato-af

[17] https://stabl.com/de/unternehmen/; https://stabl.com/de/technologie/

[18] https://www.riot-os.org/; https://legato.io/

[19] https://guide.riot-os.org/; https://guide.riot-os.org/general/vision/

[20] https://github.com/RIOT-OS/RIOT/releases/tag/2025.07; https://guide.riot-os.org/general/structure/

[21] https://github.com/RIOT-OS/RIOT/tree/master/dist/tools/kconfiglib

Kconfiglib-Sprache wurden nicht festgestellt. Laut Aussage der Maintainer wurde die Wahl von Kconfiglib durch einfache Integration in Python-Prozesse begründet.[22]

legato-af

Das legato-af (Legato Application Framework) unterstützt die Entwicklung modularer, skalierbarer IoT-Anwendungen .[23] Hierfür stellt es standardisierte APIs und Werkzeuge zur Verfügung, erzwingt Sicherheitsmechanismen durch isolierte Anwendungsausführung und ermöglicht einen Aufbau aus wiederverwendbaren Komponenten sowie eine Cloud- und Netzwerk-Integration. Es verfolgt das Ziel, die Komplexität der Entwicklung zu reduzieren und eine vollständige Abstimmung der Anwendungen auf die Anforderungen des Zielgeräts zu ermöglichen.

Obwohl die Dokumentation des Frameworks über die Verwendung vom Linux-Kconfig-Konfigurationssystem berichtet, vermerkt der Commit `2785295`, dass dieses System im Jahr 2019 durch Kconfiglib ersetzt wurde.[24] Die angegebene Motivation für diesen Wechsel betrifft den Zugriff auf Kconfig-Funktionen wie die `rsource`-Option, die Konfigurationsdateien relativ zur deren Verzeichnis einbindet. Zudem wird Kconfiglib in einem internen Skript `kconfig2dox` zur Erstellung von Doxygen-Dokumentationsdateien basierend auf dem Inhalt der Kconfiglib-Spezifikation verwendet.[25] Weder projektspezifische Anpassungen von Kconfiglib noch Erweiterungen der Syntax der Kconfiglib-Sprache wurden festgestellt.

Abhängigkeitsdiagramm

Die 734 Einträge im Abhängigkeitsdiagramm umfassen zwölf `Großprojekte` und 421 zugehörige `Unterprojekte`, drei `akademische Projekte` und 298 `private Projekte`. Dabei ziehen wir die Kategorien `akademisches Projekt` und `Großprojekt` in Betracht.

Zu den drei akademischen Projekten zählen PNCBF, Stargate und BYOS.[26] PNCBF legt den Fokus auf die Gewährleistung der Sicherheit dynamischer Systeme mittels Methoden der Regelungstechnik, während sich Stargate mit der Entwicklung eines Navigationssystems für Nano-Drohnen beschäftigt. Beide Projekte führen Kconfiglib als Abhängigkeit in ihrer `requirements.txt`-Datei, weisen aber keine weiteren Verwendungspunkte oder Kconfiglib-Spezifikationen auf. Das Projekt BYOS verwendet Kconfiglib zur Erstellung einer Wissensbasis (Knowledge Graph) mit dem Ziel, die Automatisierung des Linux-Kernel-Tunings mittels Large Language Models zu unterstützen. Hierbei dient Kconfiglib zur Extraktion relevanter Informationen aus den Kconfig-Spezifikationen und `.config`-Dateien. Es wurde keinen Erweiterungen der Syntax der Kconfiglib-Sprache festgestellt. Dementsprechend werden die Projekte dieser Kategorie aus den weiteren Analysen ausgeschlossen.

[22] https://forum.riot-os.org/t/usage-of-kconfiglib/4647/2

[23] https://docs.legato.io/latest/buildAppsMain.html; https://docs.legato.io/latest/getStarted.html;

[24] https://docs.legato.io/latest/toolsKconfig.html; https://github.com/legatoproject/legato-af/commit/2785295090dd32f7a520bc34b455e228445cc2de

[25] https://github.com/legatoproject/legato-af/tree/master/framework/tools/scripts

[26] https://github.com/MIT-REALM/pncbf; https://github.com/ETH-PBL/Stargate; https://github.com/LHY-24/BYOS

Tabelle 3.1 bietet einen einleitenden Überblick über die zwölf `Großprojekte`, welche im Folgenden einzeln vorgestellt werden.

Tabelle 3.1: GitHub-Repositories der Kategorie `Großprojekt`

Großprojekt	**Anzahl Unterprojekte**
PX4/PX4-Autopilot	188
esphome/esphome	125
espressif/esp-idf	43
project-chip/connectedhomeip	16
TrustedFirmware-M/trusted-firmware-m	14
RT-Thread/rt-thread	14
nrfconnect/sdk-nrf	8
zephyrproject-rtos/zephyr	5
projectacrn/acrn-hypervisor	4
SmingHub/Sming	2
NXP/nxp_tf-m	1
tuya/TyraOpen	1

PX4-Autopilot

Das Projekt PX4-Autopilot ist eine Flugsteuerungssoftware für Drohnen, die zu einem größeren von der Linux-Foundation unterstützten Open-Source-Ökosystem namens Dronecode gehört.[27] Durch seine stark modulare Architektur ermöglicht PX4-Autopilot eine flexible Integration von neuen Features in Form autonomer Module, die unabhängig vom Kernsystem angepasst werden können. Es bildet somit eine zentrale Steuerungseinheit, die auf dem Echtzeitbetriebssystem NuttX ausgeführt wird und umfangreiche Steuerungs- und Sicherheitsfunktionen für unterschiedliche Drohnenplattformen bereitstellt.[28] Seine Konfiguration an spezifische Anforderungen dieser Plattformen wird durch NuttX als es hochgradig konfigurierbares Echtzeitbetriebssystem erleichtert.[29] Obwohl das Konfigurationssystem von NuttX auf dem Linux-Kconfig-Konfigurationssystem basiert, unterstützt dieses zusätzlich Kconfiglib, um plattformunabhängige Konfiguration, striktere Syntaxprüfung und Qualitätssicherung zu ermöglichen.[30] Die Verwendung von Kconfiglib im Kontext von PX4-Autopilot ist daher indirekt motiviert und ergibt sich aus der zugrunde liegenden Nutzung von NuttX. Die Projektdokumentation sowie die durch Kconfiglib-Erweiterungen geprägten Spezifikationen im Quellcode deuten darauf hin, dass PX4-Autopilot Kconfiglib und das zugehörige `menuconfig`-Frontend zur Verarbeitung dieser Spezifikationen verwendet, aber keine projektspezifische Anpassungen vornimmt.[31]

[27] https://px4.io/software/software-overview/
[28] https://docs.px4.io/main/en/getting_started/px4_basic_concepts
[29] https://nuttx.apache.org/docs/latest/quickstart/configuring.html
[30] https://nuttx.apache.org/docs/latest/quickstart/install.html
[31] https://github.com/PX4/PX4-Autopilot; https://docs.px4.io/main/en/hardware/porting_guide_config

esphome

Das Open-Source-Framework esphome zielt darauf, die Komplexität der Entwicklung von Firmware für WLAN-fähige Mikrocontroller wie ESP32 oder RP2040 zu verringern.[32] Hierbei wird basierend auf benutzerdefinierten YAML-Spezifikationen der passende Quellcode generiert und anschließend als entsprechende Firmware mittels PlatformIO kompiliert.[33] PlatformIO übernimmt die Einrichtung des Entwicklungsframeworks wie ESP-IDF oder Arduino und aller notwendigen Komponenten für das gewünschte Zielsystem.[34] Die Analyse des esphome-GitHub-Repository zeigte, dass die Bibliothek Kconfiglib seit Release 2025.11.0 aus der `requirements.txt` entfernt wurde, da ESP-IDF auf eine eigene Kconfiglib-Implementierung umgestiegen ist und die externe Abhängigkeit damit entfällt.[35] Dieser zuvor bestehenden Eintrag erklärt die hohe Anzahl an `Unterprojekten` (siehe Tabelle 3.1), die somit transitive Verbindungen zu Kconfiglib aufgewiesen haben und in GitHub-Statistiken automatisch erfasst wurden. Da der aktuelle Stand des Projekts weder eine Verbindung zu Kconfiglib noch Kconfiglib-Spezifikationen beinhaltet, wurde es aus der weiteren Analysen ausgeschlossen.

esp-idf

Das ESP-IDF-Framework der Firma Espressif Systems stellt die erforderlichen Bibliotheken, Werkzeuge und Schnittstellen zur Entwicklung und Konfiguration von Firmware bereit.[36] Diese Firmware dient wiederum verschiedenen System-on-Chip (SoC)-Varianten, die Espressif für zahlreiche Anwendungen im Bereich des IoT und Edge AI zur Verfügung stellt.[37] Das `esp-idf-kconfig` ist die Komponente des Frameworks, die für die Analyse in der vorliegenden Arbeit von Relevanz ist. Sie übernimmt die Rolle des Konfigurationswerkzeugs zur Kompilierzeit und dient der Verarbeitung, Validierung und Generierung von esp-idf-kconfig-Spezifikationen.[38] Zudem ist sie flexibel genug, um auch eigenständig in anderen Projekten verwendet zu werden.

Einer der Hauptgründe für die Entwicklung des esp-idf-kconfig-Pakets war die mangelnde Wartung von Kconfiglib.[38] Obwohl die Implementierung eine Kompatibilität mit Kconfiglib anstrebt, wurde sie an die Anforderungen des ESP-IDF-Frameworks angepasst, was sich in der aktuellen Version des Pakets (3.3.0) widerspiegelt.[39] Einerseits umfasst diese `esp_kconfiglib` und `esp_menuconfig` als Kernkomponenten, deren Implementierung auf den entsprechenden Modulen der Kconfiglib-Bibliothek (siehe Kapitel 3.1) basiert. Andererseits bestehen mehrere funktionale Unterschiede wie eine neue Parser-Logik oder unterstützte Attribute wie `set` oder `warning`.

[32] https://esphome.io/

[33] https://esphome.io/guides/getting_started_hassio/#introduction-to-esphome

[34] https://docs.platformio.org/en/latest/what-is-platformio.html; https://esphome.io/guides/esp32_arduino_to_idf/

[35] https://esphome.io/changelog/2025.11.0/; https://github.com/esphome/esphome/pull/11210

[36] https://github.com/espressif/esp-idf; https://www.espressif.com/en/products/sdks/esp-idf; https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/api-conventions

[37] https://www.espressif.com/en/company/about-espressif

[38] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/; https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/index.html

[39] https://github.com/espressif/esp-idf-kconfig

Zu den weiteren projektspezifischen Erweiterungen des Pakets zählen die Kommandozeilenwerkzeuge `kconfcheck` und `kconfgen` sowie die Schnittstelle `kconfserver`.[38] Das Werkzeug `kconfcheck` prüft die Konsistenz der esp-idf-kconfig-Spezifikationen anhand projektspezifischer Syntaxregeln, während `kconfgen` verschiedene Ausgabedateien für das Build-System erzeugt.[40] Die `kconfserver`-Schnittstelle ermöglicht externen Anwendungen wie Entwicklungsumgebungen oder grafischen Oberflächen eine JSON-basierte Kommunikation mit dem Konfigurationssystem des Projekts.[41] Dadurch können externe Anwendungen die `.config`-Dateien (im Kontext des ESP-IDF als `sdkconfig` bezeichnet) bearbeiten, ohne die interne Logik des gesamten Konfigurationssystems implementieren zu müssen. Es ist wichtig zu betonen, dass die Speicherung von Optionswerten in diesen Dateien projektspezifisch durch eine zusätzliche Kennzeichnung (`# default:`) erweitert wurde. Diese signalisiert dem Konfigurationssystem, dass der in der folgenden Zeile angegebene Optionswert aus der zugrunde liegenden esp-idf-kconfig-Spezifikation abgeleitet wurde. Sie wird automatisch entfernt, sobald der Optionswert vom Benutzer explizit gesetzt wird.

Die folgenden Abschnitte widmen sich den projektspezifischen Merkmalen des esp-idf-kconfig-Pakets, um einen Überblick über die Unterschiede zu Kconfiglib zu geben.[42] Das Paket ist ausschließlich mit Python 3 kompatibel und entfernt die Unterstützung für den `tristate`-Typ, die `def_*`-Attribute sowie für die alternative Schreibweise des `help`-Attributs (`---help---`). Gleichzeitig wurden die `choice`-Optionen auf den booleschen Typ beschränkt und das `optional`-Attribut findet für sie keine Anwendung mehr. Die Regeln für gültige Optionsnamen und Einrückungen innerhalb der esp-idf-kconfig-Spezifikationen wurden ebenfalls strenger gefasst. Beispielsweise ist die Verwendung von Großbuchstaben vorgeschrieben und die Einträge einer Spezifikation haben einen einheitlichen Präfix zu verwenden. Die Syntaxvalidierung einer Spezifikation kann über die `kconfcheck`-Komponente erfolgen, die bei Bedarf automatisch eine korrigierte Fassung der Spezifikation erstellt.

Die Initialisierung einer Kconfig-Instanz in der `esp_kconfiglib`-Komponente umfasst im Vergleich zu Kconfiglib neue Parameter wie `info`, `print_report` und `parser_version`. Die ersten beiden Parameter steuern jeweils, ob informative Meldungen und der Konfigurationsreport ausgegeben werden. Basierend auf der gesetzten Detailtiefe erfasst dieser Report Informationen über den aktuellen Status sowie alle Meldungen, die während der Konfiguration des Projektes auftreten. Der Wert des Parameters `parser_version` bestimmt, ob der Kconfiglib-Parser oder der esp-idf-Parser aktiviert wird. Derzeit wird beispielsweise der esp-idf-Parser ausgewählt, wenn der Parameter (`KCONFIG_PARSER_VERSION`) größer als eins gesetzt ist.[43]

Die Implementierung des esp-idf-Parsers trennt die Definition der syntaktischen Struktur der Spezifikationen (`KconfigGrammar`-Klasse) von ihrer semantischen Verarbeitung (`Parser`-Klasse). Dadurch entsteht eine modulare und erweiterbare Architektur, die auf der `pyparsing`-Bibliothek basiert.[44] Darüber hinaus erkennt der esp-idf-

[40] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfcheck/index.html; https://github.com/espressif/esp-idf-kconfig/blob/master/kconfgen/core.py
[41] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfserver/index.html
[42] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/differences.html
[43] https://github.com/espressif/esp-idf-kconfig/blob/master/esp_kconfiglib/core.py
[44] https://github.com/pyparsing/pyparsing/tree/master

Parser `set` und `set default` sowie `warning` als neue Attribute für `(menu)config`-Optionen.[45] Die ersten beiden ermöglichen die Definition einer umgekehrten Abhängigkeit von einem booleschen zu einem nicht-booleschen Zielsymbol, während `warning` als zusätzlichen Sicherheitsmechanismus dient und eine Bestätigung bei der Änderung des Symbolwerts auffordert (vgl. Kapitel 3.3.2).

connectedhomeip

Das connectedhomeip-GitHub-Repository beinhaltet das Open-Source Projekt Matter (auch früher als `Project CHIP` bekannt).[46] Dieses Projekt zielt darauf, einen standardisierten Protokollstack für sichere und interoperable Kommunikation zwischen Smart-Home-Geräten verschiedener Hersteller bereitzustellen.[47] Im GitHub-Repository des Projekts sind Kconfiglib-Spezifikationen vorhanden, dienen aber nicht der Konfiguration von Matter selbst, sondern unterstützen die Integration von Matter auf bestimmten Plattformen wie Zephyr.[48] Da keine direkte Nutzung von Kconfiglib festgestellt werden konnte, wurde das Projekt von weiteren Analysen ausgeschlossen.

TrustedFirmware-M

TrustedFirmware-M (auch TF-M) ist eine Referenzimplementierung für Arm-Prozessoren.[49] Sie entspricht den Anforderungen und Spezifikationen, die erfüllt werden müssen, um einen Mindeststandard für die Sicherheit von IoT-Anwendungen zu gewährleisten. Daher dient TF-M als Basisarchitektur für Projekte, die eine Zertifizierung gemäß dieser Sicherheitsanforderungen anstreben. Zudem ist TF-M hochgradig konfigurierbar und stellt hierfür sowohl ein CMake-basiertes als auch ein Kconfiglib-basiertes Konfigurationssystem bereit.[50] Kconfiglib-basiertes Konfigurationssystem wurde ab Version 1.7.0 eingeführt und mit Version 1.8.0 vollständig integriert.[51] Die Analyse des GitHub-Repository zeigt, dass Kconfiglib als Grundlage für das interne Konfigurationsskript `tfm_kconfig.py` dient, aber keine projektspezifischen Erweiterungen der Kconfiglib-Sprache implementiert wurden.

rt-thread

Das Echtzeitbetriebssystem RT-Thread wird seit 2006 entwickelt und zielt darauf, ein leistungsfähiges, multithreadfähiges Open-Source-Betriebssystem für IoT-Anwendungen bereitzustellen.[52] Es ist modular aufgebaut und lässt sich für unterschiedliche ressourcenbeschränkte eingebettete Systeme konfigurieren. Eine aktive Nutzung von Kconfiglib sowie entsprechende Kconfiglib-Spezifikationen sind im GitHub-Repository vorhanden.[53] Es wurden keine projektspezifische Anpassungen von Kconfiglib identifiziert.

[45] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/language.html
[46] https://github.com/project-chip/connectedhomeip
[47] https://project-chip.github.io/connectedhomeip-doc/index.html
[48] https://github.com/project-chip/connectedhomeip/blob/master/config/zephyr/Kconfig
[49] https://trustedfirmware-m.readthedocs.io/en/latest/introduction/index.html
[50] https://trustedfirmware-m.readthedocs.io/en/latest/configuration/kconfig_system.html
[51] https://trustedfirmware-m.readthedocs.io/en/latest/releases/1.8.0.html; https://trustedfirmware-m.readthedocs.io/en/latest/releases/1.7.0.html
[52] https://rt-thread.github.io/rt-thread/page_introduction.html
[53] https://github.com/RT-Thread/rt-thread

sdk-nrf

Das sdk-nrf Repository steht für das nRF Connect SDK der Firma Nordic Semiconductor.[54] Diese Software-Suite umfasst alle notwendigen Komponenten, um die Entwicklung drahtloser Anwendungen mit geringem Stromverbrauch für die von Nordic Semiconductor hergestellten nRF-Mikrocontroller zu unterstützen. Das SDK integriert das Echtzeitbetriebssystem Zephyr als Basis. Da Zephyr seinerseits Kconfiglib einsetzt, ergibt sich für sdk-nrf eine transitive Motivation zur Verwendung von Kconfiglib. Das Repository selbst enthält Kconfiglib-Spezifikationen, definiert aber keine projektspezifischen Erweiterungen.

Zephyr

Zephyr (auch Zephyr Project, Zephyr RTOS) wurde von der Linux Foundation als Open-Source-Echtzeitbetriebssystem entwickelt und ist für den Einsatz in ressourcenbeschränkten Systemen ausgelegt.[55] Der Name wurde vom lateinischen Wort `Zephyrus` abgeleitet, das den altgriechischen Gott des Westwinds bezeichnet.[56] Zudem trägt das zentrale Kommandozeilenwerkzeug von Zephyr den Namen `west` und dient als Schnittstelle für alle entwicklungsbezogenen Aufgaben wie Kompilieren, Flashen und Debuggen.[56] Außerdem übernimmt `west` die Verwaltung aller Repositories im Zephyr-Ökosystem, da dieses unter anderem auch Repositories externer Module umfasst.

Zephyr unterstützt über 900 Boards und mehr als 200 Hardware-Erweiterungsplatinen (Stand November 2025) und findet Einsatz in zahlreichen Anwendungsbereichen wie Wearables, Gesundheitswesen, Tracking, Monitoring und Industrielles IoT. [57] Dazu zählen Produkte wie AI-Smart-Glasses, Hörgeräte, IoT-Halsbänder zur Datenerfassung bei Rindern in der landwirtschaftlichen Forschung sowie verschiedene Arten von Geräten zur Verfolgung von Industriegütern und Tieren oder zur Überwachung des Stromnetzes und des Haushalts.

Die Grundlage des Zephyr-Konfigurationssystems bildet die Kconfiglib-Bibliothek, deren Weiterentwicklung im Januar 2023 offiziell von Zephyr übernommen wurde.[58] Betrachtet man das Kconfiglib-Package auf PyPI, befindet sich die bisher analysierte Kconfiglib-Bibliothek von Ulf Magnusson unter der Version 14.1.0, während Zephyr derzeit eine Vorabveröffentlichung der Version 14.1.1a4 zur Verfügung bereitstellt.[59] Es ist wichtig zu betonen, dass die Weiterentwicklung von Kconfiglib außerhalb des Zephyr-Projekts im Repository `zephyrproject-rtos/Kconfiglib` erfolgt und nicht der Kconfiglib-Version im Hauptrepository `zephyr/scripts/kconfig` entspricht.[60]

[54] https://github.com/nrfconnect/sdk-nrf; https://docs.nordicsemi.com/bundle/ncs-latest/page/nrf/index.html

[55] https://www.zephyrproject.org/meet-linuxs-little-brother-zephyr-a-tiny-open-source-rtos/; https://github.com/zephyrproject-rtos/zephyr

[56] https://docs.zephyrproject.org/latest/develop/west/index.html

[57] https://docs.zephyrproject.org/latest/boards/index.html; https://www.zephyrproject.org/products-running-zephyr/; https://docs.zephyrproject.org/latest/hardware/porting/shields.html; https://www.zephyrproject.org/wp-content/uploads/2025/06/Zephyr-Overview-20250626.pdf

[58] https://github.com/zephyrproject-rtos/zephyr/issues/53894

[59] https://pypi.org/project/kconfiglib/#history

[60] https://github.com/zephyrproject-rtos/Kconfiglib

Derzeit führt Zephyr ein Cherry-Picking aus dem `zephyrproject-rtos/Kconfiglib` durch und ergänzt dieses durch projektspezifische Erweiterungen wie `configdefault`. Durch direkte Anfrage berichten die Maintainer des Projekts, dass Zephyr zunächst die C-basierte Linux-Kconfig-Implementierung benutzt hat. Der Wechsel zu Kconfiglib erfolgte, um plattformübergreifende Nutzung zu vereinfachen, insbesondere aufgrund von Kompilierungs- und Abhängigkeitsproblemen von Kconfig auf den Windows-Betriebssystemen.

ACRN

Das Projekt ACRN wurde im Jahr 2018 von der Linux Foundation und Intel als Open-Source-Lösung für die Virtualisierung im IoT-Bereich gestartet.[61] Es ermöglicht eine sichere und echtzeitfähige Virtualisierung verschiedener IoT-Workloads auf einer Plattform. Mit Version 2.4 erfolgte die Umstellung auf eine XML-basierte Konfiguration, wodurch Kconfiglib ersetzt wurde.[62] Aufgrund der fehlenden aktiven Verwendung von Kconfiglib wird ACRN nicht weiter betrachtet.

Sming

Das C++-basierte Open-Source-Framework Sming ist eine modulare Entwicklungsumgebung für IoT-Mikrocontrollerfamillien der Firma Espressif und der Raspberry Pi Foundation.[63] Ursprünglich wurde es im Jahr 2015 für den ESP8266 Mikrocontroller entwickelt und ist inzwischen unter anderem um eine Host-Emulationsumgebung ergänzt worden, so dass die Qualitätssicherung des erstellten Codes durchgeführt werden kann.[64] Die Verbindung zu Kconfiglib wurde über den Eintrag in der `requirements.txt`-Datei identifiziert und es wurden keine projektspezifische Anpassungen von Kconfiglib festgestellt. Allerdings deutet die Struktur der vorhandenen Spezifikationen, insbesondere die Einrückung ihrer Elemente, darauf hin, dass es sich um esp-idf-kconfig-Spezifikationen handelt.[65] Dies entspricht wiederum dem ursprünglichen Zweck des Frameworks und seiner Unterstützung der Espressif Mikrocontroller.

nxp_t-fm

Das Repository nxp_t-fm ist ein Teil der übergeordneten MCUXpresso-SDK von NXP und enthält von NXP bereitgestellte Softwarekomponenten. Zudem weist die Beschreibung des Repository darauf hin, dass es der Bereitstellung von Zephyr-Unterstützung für die RW61x-Plattform dient.[66] Obwohl die Projektdokumentation projektspezifische Änderungen von Kconfiglib beschreibt, konnte im GitHub-Repository die entsprechende Implementierung solcher Anpassungen nicht identifiziert werden.[67]

[61] https://www.linuxfoundation.org/press/press-release/the-linux-foundation-announces-an-open-source-reference-hypervisor-project-designed-for-iot-device-development

[62] https://projectacrn.github.io/2.7/release_notes/release_notes_2.4.html

[63] https://sming.readthedocs.io/en/latest/about.html

[64] https://sming.readthedocs.io/en/latest/_inc/Sming/Arch/Host/README.html

[65] https://github.com/SmingHub/Sming

[66] https://github.com/NXP/nxp_tf-m

[67] https://mcuxpresso.nxp.com/mcuxsdk/latest/html/develop/build_system/Configuration_System.html; https://github.com/NXP/nxp_tf-m/blob/mcux_main/tools/kconfig/tfm_kconfig.py

TuyaOpen

Dieses Entwicklungsframework ist darauf ausgelegt, die Entwicklung KI-gestützter IoT-Anwendungen zu erleichtern.[68] Es stellt hierfür unter anderem Funktionen für die Audio-, Bild- und Sensorverarbeitung bereit und bietet technische Mechanismen zur Integration von Large Language Models oder KI-Plattformen. TuyaOpen verwendet zum Konfigurieren sowohl Kconfiglib als auch das interne Skript `tos.py` und verfügt über Kconfiglib-Spezifikationen.[69] Durch direkte Anfrage berichten die Maintainer des Projekts, dass die leichte Integration und die Wartungsfreundlichkeit von Kconfiglib die Hauptgründe für den Einsatz von Kconfiglib waren.

3.2.3 Diskussion

Anhand der Analyse des GitHub-Repository wurden insgesamt 1604 Projekte identifiziert, die aufgrund einer direkten oder transitiven Beziehung zum Kconfiglib-Repository in den GitHub-Statistiken erfasst wurden. Davon sind 774 Repositories öffentlich zugänglich und konnten selbst definierten Kategorien zugeordnet werden, mit dem Ziel, relevante Projekte zu identifizieren. Unter relevanten Projekten werden solche verstanden, die eine aktive Verwendung von Kconfiglib oder eine eigene Variante beziehungsweise projektspezifische Anpassungen dieser Bibliothek aufweisen. Darüber hinaus war es von Interesse zu untersuchen, worum es sich bei diesen Projekten handelt, welche Ziele sie verfolgen und aus welchen Gründen sie sich für den Einsatz von Kconfiglib entschieden haben. Insgesamt wurden 14 Projekte detailiert betrachtet, von denen elf eine aktive Verwendung von Kconfiglib aufweisen. Dadurch wurden die Projekte esphome, connencthomeip und ACRN ausgeschlossen.

Die Tabelle 3.2 gibt für jedes der elf Projekte einen Überblick über die GitHub-Statisken (Stand November 2025), listet stichpunktartig projektspezifische Anpassungen sowie die jeweilige Motivation für die Verwendung von Kconfiglib. Die Projekte sind überwiegend auf IoT-Anwendungen zugeschnitten und stellen hierfür entweder Betriebssysteme oder Entwicklungsframeworks bereit. Bei PX4-Autopilot, TF-M und sdk-nrf handelt es sich hingegen um eine Flugsteuerungssoftware für Drohnen, eine Referenzimplementierung für Arm-Prozessoren beziehungsweise ein Software Development Kit.

Die Mehrheit der Projekte verwendet Kconfiglib ohne projektspezifische Anpassungen und begründen die Wahl dieser Bibliothek vor allem mit der einfachen Integration, der plattformübergreifenden Einsetzbarkeit, sowie die durch Kconfiglib bereitgestellten Funktionen. RIOT nimmt kleinere Modifikationen vor, führt aber keine neuen Sprachelemente ein. Im Gegensatz dazu hat sich ESP-IDF aufgrund der damals fehlenden Wartung von Kconfiglib dazu entschieden, eine eigene Variante (esp-idf-kconfig) zu entwickeln. Diese führt eine neue Parser-Struktur, zusätzliche Syntaxregeln und weitere Sprachelemente ein, während sie gleichzeitig die Unterstützung für bestimmte bislang vorhandene Elemente entfernt. Zephyr hat die Weiterentwicklung von Kconfiglib übernommen und verwendet diese Version als Grundlage für seine projektspezifische Version. Hierführ wendet Zephyr Cherry-Picking an und ergänzt die Bibliothek um eigene Erweiterungen wie `configdefault`.

[68] https://tuyaopen.ai/docs/about-tuyaopen

[69] https://github.com/tuya/TuyaOpen/tree/master/tools/kconfiglib: https://tuyaopen.ai/docs/tos-tools/tos-guide

Tabelle 3.2: Projekte mit Kconfiglib

Projekte	Merkmale	Anpassungen	Motivation
RIOT-OS /RIOT	Betriebssystem Version: **2025.07** (Aug 2025) 1. Version: 2013.08 (Aug 2013) > 380 Mitwirkende > 5600 Sterne-Markierungen > 2000 Forks	Methodenanpassungen für das Build-Prozess, aber keine Erweiterungen der Kconfiglib-Sprache	einfache Integration in Python-basierte Prozesse
legatoproject /legato-af	Framework Version: **19.11.6** (Dez 2021) 1. Version: 14.04.0 (Mai 2014) > 50 Mitwirkende > 150 Sterne-Markierungen > 120 Forks	keine	Zugriff auf Kconfiglib-Funktionen wie `rsource`
PX4 /PX-Autopilot	Flugsteuerungssoftware Version: **1.16.0** (Aug 2025) 1. Version: 1.0.0 (Dez 2014) > 760 Mitwirkende > 10500 Sterne-Markierungen > 470 Forks	keine	Transitive Beziehung aufgrund des zugrunde liegenden Betriebssystems NuttX, das Kconfiglib unterstützt
espressif /esp-idf	Framework Version: **5.5.1** (Sep 2025) 1. Version: 0.9 (Sep 2016) > 960 Mitwirkende > 16700 Sterne-Markierungen > 7980 Forks	Zusätzliche Syntaxregeln für gültige Optionsnamen und Einrückungen, sowie neue Attribute (`set`, `set default`, `warning`) und einen neuen Parser Nicht unterstützt werden `def_*`-Attribute, die Schreibweise `---help---`, die Tristate-Logik sowie das `optional`-Attribut für `choice`-Optionen.	Aufgrund der damals fehlenden Wartung von Kconfiglib wurde eine eigene Version implementiert
TrustedFirmware-M /trusted-firmware-m	Referenzimplementierung für Arm-Prozessoren Version: **2.2.2** (Nov 2025) 1. Version: 1.0 (Feb 2019) > 200 Mitwirkende > 30 Sterne-Markierungen > 20 Forks	keine	N/A
RT-Thread /rt-thread	Betriebssystem Version: **5.2.2** (Okt 2025) 1. Version: 1.2.0 (Jan 2014) > 780 Mitwirkende > 11570 Sterne-Markierungen > 5300 Forks	keine	N/A

Tabelle 3.3: Projekte mit Kconfiglib - Fortsetzung

Projekte	Merkmale	Anpassungen	Motivation
nrfconnect /sdk-nrf	Software Delopment Kit Version: **3.1.1** (Sep 2025) 1. Version: 1.0.0 (Apr 2020) > 440 Mitwirkende > 1200 Sterne-Markierungen > 1400 Forks	keine	Transitive Beziehung aufgrund des zugrunde liegenden Betriebssystems Zephyr
zephyrproject-rtos /zephyr	Betriebssystem Version: **4.3.0** (Nov 2025) 1. Version: 1.0.0 (Feb 2016) > 2990 Mitwirkende > 13800 Sterne-Markierungen > 8300 Forks	Neue `configdefault`-Option	Vereinfachung der plattformübergreifenden Nutzung, insbesondere auf Windows-Systemen
SmingHub /Sming	Framework Version: **6.1.0** (Jul 2025) 1. Version: 1.0.5 (Apr 2015) > 90 Mitwirkende > 1540 Sterne-Markierungen > 340 Forks	keine	N/A
NXP /nxp_tf-m	Software Delopment Kit Version: **4.2.0** (Nov 2025) 1. Version: 4.0.0 (Nov 2024) > 130 Mitwirkende > 1 Sterne-Markierungen > 1 Forks	keine	Transitive Beziehung aufgrund des zugrunde liegenden Betriebssystems Zephyr
tuya /TuyaOpen	Framework Version: **1.5.1** (Nov 2025) 1. Version: 1.0.0 (Aug 2024) > 15 Mitwirkende > 1100 Sterne-Markierungen > 190 Forks	keine	Leichte Integration in Python-basierte Prozesse und Wartungsfreundlichkeit

3.3 Kconfiglib-Erweiterungen

Die Relevanz der erarbeiteten Grundlage besteht darin, dass sie die Voraussetzung für die Ableitung von Transformationsstrategien für Kconfiglib-Erweiterungen in die Linux-Kconfig-Sprache schafft. Sie ermöglicht die notwendige Transparenz darüber, welche Abweichungen zur Linux-Kconfig-Sprache existieren und welche Aspekte bei der Entwicklung von Transformationsregeln zu berücksichtigen sind.

3.3.1 Methodik

Es soll eine fundierte Wissensbasis über die Kconfiglib-Erweiterungen bereitgestellt werden. Die Vorgehensweise umfasst die Identifikation der Erweiterungen sowie deren Erläuterung im Hinblick auf ihre Funktionsweise. Die Voraussetzung hierfür bildet die im vorherigen Kapitel durchgeführte Analyse (vgl. Kapitel 3.2), da sie zunächst einen Überblick über die Verbreitung von Kconfiglib-Bibliothek und ihren Varianten vermittelt (vgl. Tabelle 3.2).

Dieser Analyse nach wird die bisher betrachtete Kconfiglib-Bibliothek von Ulf Magnusson (GitHub-Nutzer `ulfalizer`) seit der Version 14.1.0 nicht mehr aktiv gewartet. Ihre Weiterentwicklung wurde von Zephyr übernommen und wird unter dem gleichen Namen im Repository `zephyrproject-rtos/Kconfiglib` fortgeführt. Daher trennen wir im Folgenden zwischen den Bezeichnungen *Kconfiglib* und *Zephyr-Kconfiglib*. Da Zephyr zudem im eigenen Repository (`zephyrproject-rtos/zephyr`) eine projektspezifische Version von Zephyr-Kconfiglib bereitstellt, wird diese als *ZRTOS-Kconfiglib* bezeichnet. Darüber hinaus stellt das ESP-IDF-Framework mit dem Paket esp-idf-kconfig eine weitere Variante von Kconfiglib bereit.

Die identifizierten Varianten von Kconfiglib dienen als Ausgangspunkt für die Ermittlung der Kconfiglib-Erweiterungen. Die Dokumentationen von Kconfiglib und ZRTOS-Kconfiglib wurden zunächst herangezogen, da sie die vorhandenen Kconfiglib-Erweiterungen adressieren und deren Funktionsweise anhand von Beispielen erläutern.[70] Anschließend wurde der Quellcode der jeweiligen Variante analysiert, um weitere Erweiterungen zu ermitteln. Die Varianten Zephyr-Kconfiglib und ZRTOS-Kconfiglib übernehmen die Struktur des Kconfiglib-Quellcodes nahezu unverändert, wodurch es möglich ist, die jeweilige Version der Datei `kconfiglib.py` mit der Originalversion aus Kconfiglib zu vergleichen und projektspezifische Anpassungen zu identifizieren. Im Gegensatz dazu wurde bei esp-idf-kconfig aufgrund der abweichenden Quellcode-Struktur die Dokumentation der esp-idf-kconfig-Sprache herangezogen.[71]

Abschließend wurden die ermittelten Kconfiglib-Erweiterungen aller Varianten hinsichtlich ihrer Syntax und Semantik systematisch beschrieben. Hierfür wurde der entsprechende Quellcode untersucht und die Funktionsweise anhand von beispielhaften Spezifikationen über das `menuconfig`-Frontend analysiert. Insbesondere konnten im Quellcode von Kconfiglib, Zephyr-Kconfig und ZRTOS-Kconfig die unterstützten Tokens, wie `_T_OPTION` oder `_T_DEF_STRING`, berücksichtigt werden, um jene Schlüsselwörter zu identifizieren, die vom Lexer und Parser der Linux-Kconfig-Sprache nicht erkannt oder laut der Commmit-Historie nicht mehr unterstützt sind.

[70] https://github.com/ulfalizer/Kconfiglib?tab=readme-ov-file#kconfig-extensions; https://docs.zephyrproject.org/latest/build/kconfig/extensions.html

[71] https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/language.html; https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/differences.html

3.3.2 Ergebnisse

Tabelle 3.4 bietet einen Überblick über die identifizierten Kconfiglib-Erweiterungen der jeweils betrachteten Variante von Kconfiglib. Für jede Variante werden das zugehörige Repository, ihre terminologische Bezeichnung im Rahmen dieser Arbeit sowie eine stichpunktartige Auflistung der jeweiligen Kconfiglib-Erweiterungen erfasst. Im Anschluss an die kurze Erläuterung der Kconfiglib-Varianten werden die aufgeführten Erweiterungen einzeln vorgestellt. Diese Vorstellungen basieren auf der Analyse des Quellcodes sowie der verfügbaren Dokumentation der jeweiligen Kconfiglib-Varianten.[72]

Tabelle 3.4: Kconfiglib-Varianten

Repository	**ulfalizer /Kconfiglib**	**zephyrproject-rtos /Kconfiglib**	**zephyrproject-rtos /zephyr/.../kconfig**	**espressif /esp-idf-kconfig**
Bezeichnung	Kconfiglib	Zephyr-Kconfiglib	ZRTOS-Kconfiglib	esp-idf-kconfig
Wartung	keine	aktiv	aktiv	aktiv

Erweiterungen der Linux-Kconfig-Sprache	`source`-Option, die Glob-Muster in Pfaden unterstützt und ihre Alternativen, d.h. `osource`, `rsource` und `orsource`
	
	`def_*`-Attribute, d.h. `def_string`, `def_hex`, `def_int`
		Dieselben wie in Kconfiglib	Dieselben wie in Kconfiglib
			
			`configdefault`-Option
				`source`-Option, die Glob-Muster in Pfaden unterstützt und ihre Alternativen, d.h. `osource`, `rsource` und `orsource`
				
				`warning`-Attribut
				
				Attribute `set` und `set default`

Von Kconfiglib-Varianten unterstützte, in der Linux-Kconfig-Sprache nicht mehr gültige Elemente	`option`-Attribute, d.h. `option (env| allnoconfig_y | defconfig_list | modules)` `named choice`-Option Attribute und Elemente von `choice`-Optionen Schreibweise des `help`-Attributs: `---help---` Schreibweise des booleschen Typs: `boolean` Optionen `grsource` und `gsource`	Dieselben wie in Kconfiglib	Dieselben wie in Kconfiglib	`named choice`-Option `option env`-Attribut (als veraltet eingestuft, aber kann eingelesen werden)

Im Gegensatz zur Linux-Kconfig-Sprache (Kernel-Version 6.18, Stand: November 2025) weist Kconfiglib zusätzliche Sprachelemente auf. Dazu gehört die `source`-

[72] https://github.com/ulfalizer/Kconfiglib; https://docs.zephyrproject.org/latest/build/kconfig/extensions.html; https://github.com/zephyrproject-rtos/Kconfiglib; https://docs.espressif.com/projects/esp-idf-kconfig/en/latest/kconfiglib/language.html

Option mit Unterstützung für Glob-Muster sowie neue Alternativen dieser Option wie `rsource` oder `osource`. Zudem erweitert Kconfiglib die Menge der verfügbaren `def_*`-Attribute für Optionen der Typen int, hex und string.

Kconfiglib unterstützt das Einlesen bestimmter, historisch bedingter Schreibweisen zur Sicherung der Abwärtskompatibilität. Dazu zählen mehrere Varianten des `option`-Attributs, die `named choice`-Option und das `optional`-Attribut für `choice`-Optionen. Darüber hinaus werden alternative Schlüsselwörter `grsource` und `gsource`, sowie Schreibweisen `---help---` und `boolean` akzeptiert. Diese werden intern auf die bestehende Tokens abgebildet und jeweils als Schlüsselworter `rsource`, `source`, `help` beziehungsweise `bool` verarbeitet.

Zephyr-Kconfiglib und ZRTOS-Kconfiglib basieren auf der Struktur von Kconfiglib und übernehmen somit die in den vorherigen Abschnitten aufgelisteten Anpassungen. Der Vergleich der jeweiligen Datei `kconfiglib.py` dieser beiden Varianten mit der Referenzversion von Kconfiglib zeigt, dass lediglich kleinere strukturelle Modifikationen des Quellcodes vorgenommen werden, wie die Einführung von `finally`-Klauseln innerhalb von `try`-Anweisungen oder das Hinzufügen zusätzlicher Variablen. Für die vorliegende Arbeit sind zwei Sprachelemente relevant, die beide Zephyr-Varianten im Vergleich zu Kconfiglib aufweisen. ZRTOS-Kconfiglib führt die `configdefault`-Option ein, während Zephyr-Kconfiglib das `modules`-Attribut ergänzt. Da das `modules`-Attribut mit der Syntax der Linux-Kconfig-Sprache übereinstimmt, wurde es nicht als Erweiterung in Tabelle 3.4 aufgenommen.

Die Variante esp-idf-kconfig umfasst nur teilweise dieselben Erweiterungen wie Kconfiglib. Dies liegt daran, dass sie die Unterstützung für einige Optionen und Attribute entfernt hat (vgl. Tabelle 3.2). Zusätzlich erlaubt esp-idf-kconfig die projektspezifische Attribute `set`, `set default` und `warning`.

`source`-Option

Die `source`-Option dient der dynamischen Einbeziehung weiterer Kconfiglib-Spezifikationen, die über den angegebenen Dateipfad geladen werden. Im Gegensatz zur Linux-Kconfig-Sprache unterstützt die `source`-Option in allen identifizierten Kconfiglib-Varianten (vgl. Tabelle 3.4) die Verwendung von Glob-Mustern als Dateipfade. Dabei sind Glob-Muster als Zeichenketten zu verstehen, die spezielle Platzhalter wie `*` oder `?` enthalten und zum Abgleich mit Pfadnamen verwendet werden.[73] Dadurch können mit einer einzigen `source`-Option mehrere Kconfiglib-Spezifikationen adressiert und zugleich alle dem Glob-Muster entsprechenden Spezifikationen flexibel eingebunden werden, ohne deren einzelne Namen angeben zu müssen.

Alle Kconfiglib-Varianten verwenden die Python-Bibliothek `glob`, um eine Liste aller mit dem angegebenen Glob-Muster übereinstimmenden Pfade bereitzustellen.[74] Angenommen, in dem Verzeichnis `test` befinden sich die Dateien `{KconfigA, b.doc, KconfigB}` und eine `source`-Option enthält als Dateipfad das Glob-Muster `"test/K*"`. In diesem Fall werden alle Kconfig-Spezifikationen eingebunden, deren Name mit dem Buchstaben K beginnt und keine oder mehrere weitere Zeichen

[73] https://pubs.opengroup.org/onlinepubs/9799919799/functions/glob.html
[74] https://docs.python.org/3/library/glob.html

enthält. Die Regeln für den Musterabgleich dieser Bibliothek ähneln denen der Unix-Shell, erlauben aber zum Beispiel keine Tilde-Ersetzung (`~`) durch den Pfad zum Home-Verzeichnis.[75] Außerdem liefert die `glob`-Bibliothek die Ergebnisse in einer undefinierten Reihenfolge.

Die Python-Bibliothek `glob` unterstützt die Platzhalter `?, *, [], [!], **`.[76] Diese stehen jeweils für genau ein Zeichen (`?`), kein oder mehrere beliebige Zeichen (`*`) sowie für Zeichenbereiche (`[]`) oder deren Negation (`[!]`). Angenommen, in einem Verzeichnis befinden sich die Dateien `{a.txt, b.txt, c.txt, d.txt}` und es wird das Glob-Muster `"[!cd].*"` verwendet. Dieses Muster spezifiziert alle Dateien, deren Name nicht mit den Buchstaben c oder d beginnt, gefolgt von einem Punkt und gegebenenfalls mehreren beliebigen Zeichen. Die Methoden der `glob`-Bibliothek liefern für dieses Muster die Dateien `a.txt, b.txt` zurück. Hingegen wurden für das Glob-Muster `"[cd].*"` die Dateien `c.txt, d.txt` zurückgegeben. Darüber hinaus interpretiert die Bibliothek den Platzhalter `**` als Anweisung zur rekursiven Durchsuchung von Unterverzeichnissen, wobei die Verzeichnisse selbst in die Ergebnismenge einbezogen werden. Voraussetzung hierfür ist, dass der interne Parameter `recursive` der entsprechenden Methoden dieser Bibliothek auf `True` gesetzt ist. s Die Alternativen der `source`-Option sind `osource`, `rsource` und `orsource`. Ihre grundlgende Funktiosweise hinsichtlich der Unterstützung von Glob-Muster entspricht derjenigen von `source`. Im Gegensatz zur `source`-Option, die verlangt, dass mindestens ein Dateipfad gefunden wird, erlaubt `osource` eine leere Ergebnissmenge, ohne eine Fehlermeldung auszulösen. Dies ist nützlich wenn optionale Dateien generiert werden. Während die `source`-Option Kconfiglib-Spezifikationen relativ zum Projektwurzelverzeichnis einbindet, ermöglicht `rsource` eine Einbindung dieser relativ zu dem Verzeichnis, in dem sich die Spezifikation mit dieser Option befindet. Darauf aufbauend erlaubt die `orsource`-Option ebenfalls eine leere Ergebnismenge.

Alle identifizierten Kconfiglib-Varianten rufen intern die Methode `iglob` der `glob`-Bibliothek auf, um die `source`-Option sowie deren Alternativen zu verarbeiten. Dabei wird als Argument eine zusammengesetzte Zeichenkette übergeben, die aus dem Projektwurzelverzeichnis und dem in Anführungszeichen angegebenen Dateipfad gebildet wird. Im Fall einer `rsource`- oder `orsource`-Option setzt sich dieses Argument hingegen aus dem Projektwurzelverzeichnis, dem Verzeichnis der Spezifikation, in der die Option verwendet wird, sowie dem angegebenen Dateipfad zusammen (siehe Quelltext 3.7). Darüber hinaus wird die Methode ohne Setzen des Arguments `recursive` auf `true` aufgerufen, sodass beim Auftreten des Glob-Musters `**` keine rekursive Durchsuchung von Unterverzeichnissen erfolgt.

`def_*`-Attribute

Die `def_*`-Attribute erlauben es, den Typ und den Standardwert einer `(menu)config`-Option gleichzeitig festzulegen. Kconfiglib, Zephyr-Kconfiglib und ZRTOS-Kconfiglib akzeptieren `def_bool`, `def_tristate`, `def_int`, `def_hex` und `def_string` als mögliche Attribute, während esp-idf-kconfig keines dieser Attribute unterstützt. Bei der Verwendung eines `def_*`-Attributs ist es wichtig, dass kein Inline-Prompt erlaubt ist, sondern das `prompt`-Attribut muss in einer separaten Zeile angegeben werden.

[75] POSIX.1-2024 - 2.14.3 Patterns Used for Filename Expansion
[76] https://github.com/python/cpython/blob/3.14/Lib/glob.py

`option`-Attribute

Zu dieser Gruppe zählen die Attribute, die die Syntax `option <keyword>[=<value>]` verfolgen und in Version 4.1 des Linux-Kernels eingeführt wurden, aber in der aktuellen Version (6.18) nicht mehr in ihrer ursprünglichen Form vorhanden sind.[77] Sie wurden entweder komplett aus der Syntax entfernt, durch Umgebungsvariablen ersetzt oder umbenannt (vgl. Mapping 3.4.2). Im Folgenden wird deren Funktionsweise in den Kconfiglib-Varianten näher erläutert.

Das `option env`-Attribut ordnet den angegebenen Wert als Standardwert zu einer `(menu)config`-Option zu, die mit diesem Attribut gekenzeichnet wurde. Dies ist insbesondere dann nützlich, wenn Benutzer bestimmte systemspezifische Informationen übergeben möchten. Angenommen, es existiert eine Umgebungsvariable namens `ENV_A`, die den Wert `i7-1260P` hat, sowie eine `config`-Option mit dem Namen `OPTION_A`. Die Option `OPTION_A` hat den Typ string und das Attribut `option env="ENV_A"`. Diese Definition führt dazu, dass der Standardwert von `OPTION_A` auf `i7-1260P` festgelegt wird. Zudem weist Kconfiglib ebenso wie die beiden Zephyr-Varianten darauf hin, dass eine manuelle Anpassung des Optionsnamens von `OPTION_A` in `ENV_A` zur Kompatibilität mit der C-Implementierung erforderlich ist. Darüber hinaus ist es zu beachten, dass eine mit dem `option env`-Attribut gekennzeichnete Option nicht in die `.config`-Datei geschrieben wird.

Das `option allnoconfig_y`-Attribut dient als Flag für externe Skripte und kennzeichnet eine Konfigurationsoption, deren Wert unabhängig von der Ausführung der Skripte wie `allnoconfig.py` auf `y` gesetzt bleibt. Das `option defconfig_list`-Attribut kennzeichnet eine Option des Typs string, deren `default`-Attribute eine priorisierte Liste von Konfigurationsdateien spezifizieren (vgl. Quelltext 3.1). Diese Dateien werden im Build-Prozess zum Laden der grundlegenden Konfiguration herangezogen, sofern keine `.config`-Datei vorhanden ist. Intern wird das `option defconfig_list`-Attribut als Flag erkannt, das zunächst eine sequenzielle Überprüfung der durch die `default`-Attribute angegebenen Konfigurationsdateien auslöst. Dabei wird unter Beachtung der Attributreihenfolge der Wert des ersten `default`-Attributs übernommen, wenn die if-Bedingung dieses Attributs erfüllt ist und die angegebene Konfigurationsdatei geöffnet werden kann. Darüber hinaus wird eine mit diesem Attribut gekennzeichnete Option nicht in die `.config`-Datei geschrieben.

Quelltext 3.1: Beispiel für `defconfig_list`-Option

```
1 config DEFCONF
2   string
3   option defconfig_list
4   default "/lib/ARCH32/.config" if ARCH32
5   default "/lib/ARCH64/.config" if ARCH64
```

Das `option modules`-Attribut dient der Kennzeichnung einer Option, die die modulare Konfigurierbarkeit des gesamten Konfigurationsraums steuert. Das heißt, wenn

[77] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/kbuild/kconfig-language.txt?h=v4.1; https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/kconfig/zconf.y?h=v4.1

diese Option den Wert `y` hat, dann steht der Tristate-Wert `m` für alle anderen Konfigurationsoptionen zur Auswahl. Die esp-idf-kconfig-Variante erlaubt die Verwendung dieses Attributs nicht, da sie den Typ tristate nicht unterstützt. Alle anderen betrachteten Kconfiglib-Varianten folgen den aus der Linux-Kconfig-Sprache bekannten Regeln, dass dieses Attribut nur einmal im gesamten Konfigurationsraum an eine `(menu)config`-Option vergeben werden kann.

`named choice`-Option

Quelltext 3.2 veranschaulicht eine `named choice`-Option anhand der Ausschnitte zweier eigener Kconfiglib-Spezifikationen, die in den Verzeichnissen `boo` und `bar` definiert sind. Er bildet zwei Definitionen einer `choice`-Option ab, die jeweils mit dem Schlüsselwort `choice` gefolgt von einem Namen eingeführt werden. Der Name einer `choice`-Option besteht wiederum, wie der Name einer `config`-Option, aus alphanumerischen Zeichen und Unterstrichen (vgl. Syntax 2.2.2). Eine `named choice`-Option kann dieselbe Attribute wie eine `choice`-Option annehmen.

Intern führt das Einlesen der ersten Definition zur Erstellung einer neuen Instanz, die den Namen `GRAFIK_TYPE` und die Optionen `G_22` und `G_23` in ihrer internen Datenstruktur speichert. Beim Einlesen der zweiten Definition wird die bestehende Instanz wiederverwendet und ihre Datenstruktur um die Option `G_24` ergänzt. Dies führt dazu, dass alle gespeicherten `config`-Optionen als Auswahloptionen zusammengeführt werden und in den Konfigurations-Frontends unter einer `named choice`-Option als Auswahloptionen erscheinen.

Quelltext 3.2: Beispiel für `named choice`-Option

```
1  # erste Definition in foo/Kconfig
2  choice GRAFIK_TYPE
3    prompt "wähle Typ der Grafik aus"
4    config G_22
5      bool "Typ 22"
6    config G_23
7      bool "Typ 23"
8
9  # zweite Definition in bar/Kconfig
10 choice GRAFIK_TYPE
11   config G_24
12     bool "Typ 24"
```

Attribute und Elemente von `choice`-Optionen

Einige Kconfiglib-Varianten erlauben im Vergleich zur Linux-Kconfig-Sprache zusätzliche Attribute und Elemente einer `choice`-Option. Dazu zählen die explizite Typdeklaration (bool oder tristate) dieser Optionen, das `optional`-Attribut sowie die Unterstützung von `menuconfig`-Optionen als Elemente innerhalb einer `choice`-Option. Insbesondere kann das `optional`-Attribut ausschließlich `choice`-Optionen zugewiesen werden. Es erlaubt dem Benutzer, die gesamte `choice`-Option auf den

Wert `n` zu setzen und somit die sonst verpflichtende Auswahl einer in `choice` enthaltenen Option vollständig auszuschließen.

Angenommen, eine `choice`-Option umfasst drei `config`-Optionen (A, B und C) und besitzt einen Prompt-Text (`"wähle Option"`). In diesem Fall ist diese `choice`-Option im Konfigurations-Frontend standardmäßig sichtbar und der Benutzer muss eine der drei `config`-Optionen auswählen. Sobald ein `optional`-Attribut definiert ist, erscheint vor dem Prompt-Text die Notation `[*]` beziehungsweise `[]`, die darauf hinweist, ob diese `choice`-Option ausgewählt oder abgewählt ist.

`warning`-Attribut

Das `warning`-Attribut ist eine projektspezifische Kconfiglib-Erweiterung von esp-idf-kconfig-Variante für `(menu)config`-Optionen. Es kennzeichnet Optionen, bei denen eine manuelle Änderung ihres Werts unerwünschte kontextabhängige Nebenwirkungen verursachen kann. Seine Syntax besteht aus dem Schlüsselwort `warning`, gefolgt vom Text der Warnmeldung in Anführungszeichen. Wird eine mit diesem Attribut versehene Option in Konfigurations-Frontends geändert, erscheint ein Pop-Up-Fenster, das dem Benutzer die Warnmeldung anzeigt und ihn zur Bestätigung der gewüschten Wertänderung auffordert.

`set`-Attribute

Die Attribute `set` und `set default` sind projektspezifische Kconfiglib-Erweiterungen von esp-idf-kconfig-Variante für boolesche `(menu)config`-Optionen. Sie ermöglichen es einer booleschen Option, den Standardwert einer nicht-booleschen Zieloption festzulegen. Während `set`-Attribut interne Abhängigkeiten und Standardwerte der Zieloption überschreibt, berücksichtigt `set default` sie und setzt einen neuen Standardwert, der vom Benutzer weiterhin geändert werden kann. Hingegen schließt `set`-Attribut eine Änderung durch den Benutzer aus.

Die Funktionsweise von `set`- und `set default`-Attributen wird anhand einer eigenen esp-idf-kconfig-Spezifikation 3.3 veranschaulicht. Diese spezifiziert vier `config`-Optionen jeweils vom Typ `string`, `int` beziehungsweise `bool`. Die nicht-booleschen Optionen `UNI_NAME` und `NUMBER_OF_STUDENTS` speichern jeweils den Namen einer Universität als Zeichenkette beziehungsweise die Anzahl der Studierenden als numerischen Wert. Die beiden booleschen Optionen `UNI_DEFAULT_VALUES` und `FIN_FACULTY` beeinflussen die Werte der nicht-booleschen Optionen jeweils mittels des `set default`- beziehungsweise `set`-Attributs.

Der Quelltext 3.4 veranschaulicht den Einfluss der booleschen Optionen der Spezifikation 3.3 auf die übrigen Optionen. Er ist in vier Bereiche unterteilt, in denen die Ausgaben des Konfigurations-Frontends `esp_menuconfig` dargestellt sind. Die Zeilen 1 bis 4 zeigen den Zustand von `esp_menuconfig`, nachdem die Spezifikation erstmals geladen wurde und weder `UNI_DEFAULT_VALUES` noch `FIN_FACULTY` ausgewählt sind. Wird ausschließlich `UNI_DEFAULT_VALUES` ausgewählt, werden den Zieloptionen die Standardwerte `OvGU` und `12606` zugewiesen (vgl. Zeilen 6-9). Die Auswahl der Option `FIN_FACULTY` legt die spezifische Werte (`FIN`, `1464`) für eine Fakultät fest (vgl. Zeilen 11-14). Werden beide booleschen Optionen gewählt, überschreibt die `FIN_FACULTY` die durch `UNI_DEFAULT_VALUES` gesetzten Werte (vgl. Zeilen 16-19).

Quelltext 3.3: Beispiel einer esp-idf-kconfig-Spezifikation

```
mainmenu "Test set and set default"
    menu "My Test"
        config UNI_NAME
            string "Name der Uni"
            default "Name"

        config NUMBER_OF_STUDENTS
            int "Anzahl der Studierenden"
            default 0

        config UNI_DEFAULT_VALUES
            bool "set default"
            default n
            set default UNI_NAME="OvGU"
            set default NUMBER_OF_STUDENTS=12606

        config FIN_FACULTY
            bool "set"
            default n
            set UNI_NAME="FIN"
            set NUMBER_OF_STUDENTS=1464
endmenu
```

Quelltext 3.4: set und set default in esp_menuconfig

```
(Name) Name der Uni (default value)
(0) Anzahl der Studierenden (default value)
[ ] set default (default value)
[ ] set (default value)
----------------------------------------------------------------
(OvGU) Name der Uni (default value)
(12606) Anzahl der Studierenden (default value)
[*] set default
[ ] set (default value)
----------------------------------------------------------------
(FIN) Name der Uni (default value) (force-set by FIN_FACULTY)
(1464) Anzahl der Studierenden (default value) (force-set by
   FIN_FACULTY)
[ ] set default
[*] set
----------------------------------------------------------------
(FIN) Name der Uni (default value) (force-set by FIN_FACULTY)
(1464) Anzahl der Studierenden (default value) (force-set by
   FIN_FACULTY)
[*] set default
[*] set
```

`configdefault`-Option

Eine `configdefault`-Option stellt eine projektspezifische Option von ZRTOS-Kconfiglib dar. Sie erlaubt ausschließlich die Verwendung von `default`-Attributen und dient dazu, die gewünschte Standardwerte für die gleichnamige `(menu)config`-Option zu spezifizieren. Intern werden die durch `configdefault` definierten Standardwerte zur Liste aller Standardwerte einer Konfigurationsoption hinzugefügt, wobei die Reihenfolge der Definitionen berücksichtigt wird. Das bedeutet, dass die durch das Symbol | visuell getrennten Seiten des Beispiels 3.5 äquivalent sind. Die rechte Seite verdeutlicht, wie die Reihenfolge der `default`-Attribute beibehalten wird und wie die `if`-Option `ARCH32` in die entsprechenden if-Bedingungen der `default`-Attribute propagiert wird. Die `OPTION_C` übernimmt gemäß der Reihenfolge den ersten Standardwert, dessen if-Bedingung erfüllt ist.

Quelltext 3.5: Beispiel für `configdefault`-Option

```
config OPTION_C                 | config OPTION_C
  int "C"                       |   int "C"
  default 5 if OPTION_E         |   default 5 if OPTION_E
if ARCH32                      <=>  default 6 if OPTION_F && ARCH32
configdefault OPTION_C          |
  default 6 if OPTION_F         |
endif                           |
```

3.3.3 Diskussion

Die fehlende kontinuierliche Wartung der Kconfiglib-Bibliothek führte zu projektspezifischen Weiterentwicklungen und eigenständigen Abspaltungen. Dies spiegelt sich in den vier identifizierten Kconfiglib-Varianten, deren Einfluss auf die Linux-Kconfig-Sprache sich in den jeweils eingeführten Kconfiglib-Erweiterungen manifestiert. Hierbei lassen sich diese Erweiterungen auf hoher Ebene in zwei Gruppen einteilen. Zum einen sind das die für die Linux-Kconfig-Sprache unbekannte Sprachelemente wie Glob-Muster oder `configdefault`-Optionen. Zum anderen umfassen die Kconfiglib-Varianten Erweiterungen wie `named choice`-Optionen, die im Linux-Kconfig-Sprachumfang nicht mehr vorgesehen sind. Dies deutet darauf hin, dass diese Elemente weiterhin in anderen Projekten benötigt werden können. Quelltext 3.6 bietet einen Überblick über die durch Kconfiglib-Erweiterungen verursachten Abweichungen vom Sprachumfang der Linux-Kconfig-Sprache. Ergänzend zu Tabelle 3.4 werden darin die Unterschiede auf der Syntaxebene visualisiert.

Infolge der projektspezifischen Erweiterungen ist keiner der Parser der vier identifizierten Kconfiglib-Varianten in der Lage, alle zugehörigen Spezifikationen vollständig zu verarbeiten. Der Kconfiglib-Parser kann keine Zephyr-Kconfiglib-Spezifikationen verarbeiten, da diese das zusätzliche Attribut `modules` einführen. Umgekehrt verarbeitet Zephyr-Kconfiglib zwar die Kconfiglib-Spezifikationen, scheitert aber an ZRTOS-Kconfiglib-Spezifikationen aufgrund des dort eingeführten `configdefault`-Attributs. Die esp-idf-kconfig-Spezifikationen können ausschließlich vom ESP-IDF-Parser verarbeitet werden, da sie weitere, von den übrigen Varianten unbekannte Attribute `set`, `set default` und `warning` verwenden.

Quelltext 3.6: Syntax der Linux-Kconfig-Sprache mit Kconfiglib-Erweiterungen

```
mainmenu ::= mainmenu "<STRING>" <elements>
elements ::= (menu)config_opt | menu_opt | [named]choice_opt
           | source_opt | if_opt | comment_opt | configdefault
-----------------------------------------------------------------
(menu)config_opt ::=(menu)config <config_name> <config_attr>
 menu_opt        ::= menu "<STRING>"
                     (visible if <expr> | depends on) <elements>
                     endmenu
[named]choice_opt::= choice [<choice_name>] <choice_att>
                     (<(menu)config_opt> | <comment_opt> | <if_opt>)
                     endchoice
 source_opt      ::= (source|osource|rsource|orsource) "<path>"
 if_opt          ::= if <expr> <elements> endif
 comment_opt     ::= comment "<STRING>" [depends on <expr>]
 configdefault   ::= configdefault <config_name>
                     default <expr> [if <expr>]
-----------------------------------------------------------------
number      ::= (<INT> | <HEX>)
config_name ::= choice_name ::= <non-constant symbol>
type        ::= int | string | bool | hex | tristate
def_type    ::= def_bool|def_tristate|def_string | def_int | def_hex
choice_type ::= bool | tristate
set_value   ::= (<value> | <config_name>)
-----------------------------------------------------------------
config_attr ::= (<type> | <def_type> <value> [if <expr>]
              | <type> "<STRING>" [if <expr>])
              | prompt "<STRING>"       [if <expr>]
              | default <expr>          [if <expr>]
              | depends on <expr>
              | select  <config_name>   [if <expr>]
              | imply   <config_name>   [if <expr>]
              | range <number> <number> [if <expr>]
              | help <MULTILINE STRING>
              | (modules | option modules)
              | transitional
              | option (env=<value> | allnoconfig_y | defconfig_list)
              | set <config_name> = <set_value>
              | set default <config_name> = <set_value>
              | warning "<STRING>"      [if <expr>]
-----------------------------------------------------------------
choice_attr ::= <choice_type>
              | optional
              | prompt "<STRING>"       [if <expr>]
              | default <config_name>   [if <expr>]
              | depends on <expr>
              | help <help text>
```

3.4 Abbildung von Kconfiglib auf Kconfig

Aufbauend auf dem vorherigen Kapitel werden Transformationsregeln für Kconfiglib-Erweiterungen konzipiert. Diese Regeln bilden die Grundlage für die Implementierung eines Transformationsprototyps, der durch Kconfig-Erweiterungen geprägte Spezifikationen in semantisch äquivalente Spezifikationen der Linux-Kconfig-Sprache überführt.

3.4.1 Methodik

Der Analyse des vorherigen Kapitels nach existieren vier Varianten von Kconfiglib, die jeweils Erweiterungen gegenüber der Linux-Kconfig-Sprache aufweisen (vgl. Tabelle 3.4 und Quelltext 3.6). Die Konzeption der Transformationsregeln basiert auf einem systematischen Vergleich der Syntax und Semantik der Kconfiglib-Erweiterungen mit der Linux-Kconfig-Sprache. Es wird insbesondere die technische Implementierung dieser Erweiterungen auf Quellcodeebene betrachtet, um daraus geeignete Transformationsregeln abzuleiten.

3.4.2 Ergebnisse

In den folgenden Abschnitten werden die Transformationsregeln für einzelne Kconfiglib-Erweiterungen vorgestellt. Der Fokus liegt dabei auf der Herausarbeitung der Unterschiede zur Linux-Kconfig-Sprache, die die Ableitung dieser Regeln begründen.

`source`-Option

Die Linux-Kconfig-Sprache unterstützt ausschließlich die `source`-Option und kennt keine zusätzlichen Alternativen. Für die Transformation ist es daher zunächst erforderlich, die Optionen `rsource`, `orsource` und `osource` in äquivalente `source`-Optionen zu überführen. Der Quelltext 3.7 veranschaulicht, wie der in einer Spezifikation angegebe Pfad durch die Optionen `source` und `rsource` intern verarbeitet wird. Hierzu ist ein Verzeichnisbaum eines Projekts dargestellt, wobei die Optionen `source` und `rsource` in der Kconfiglib-Spezifikation im Unterverzeichnis `U02` definiert sind. Als Parameter erhalten beide Optionen den Pfad zu einer Kconfig-Spezifikation, die sich im Unterverzeichnis `U01` befindet. Die von diesen Optionen verarbeiteten Pfade unterscheiden sich durch das zusätzliche Präfix `/U02/` (vgl. Zeilen 8-9 im Quelltext 3.7). Dies ist darauf zurückzuführen, dass die `rsource`-Option den Pfad relativ zu dem Verzeichnis interpretiert, in dem sie definiert ist. Damit lässt sich eine `rsource`-Option durch eine äquivalente `source`-Option ersetzen, sofern der entsprechende Pfad explizit um das Verzeichnispräfix erweitert wird.

Die Linux-Kconfig-Sprache unterstützt keine Glob-Muster als angegebenen Dateipfad einer Spezifikation. Deswegen ist es für die Transformation erforderlich, Platzhalter `*`, `?`, `[]` oder `[!]` innerhalb eines Dateipfads aufzulösen. Da die Auflösung zu mehreren Treffern führen kann, ist sicherzustellen, dass jeder ermittelte Pfad als eigene source-Option angegeben wird. Der Quelltext 3.8 fasst die Transformationsregeln für die `source`-Option sowie deren Alternativen zusammen. Dabei steht die Bezeichnung `PFAD_MIT_GM` für einen Dateipfad, der die Platzhalter (`*`, `?`, `[]` oder `[!]`) enthält und als Glob-Muster aufgelöst wird, während `PFAD_OHNE_GM` einen Dateipfad ohne die Platzhalter darstellt. Zudem wird das Verzeichnis der Spezifikation, in der die `(o)rsource`-Optionen definiert sind, als `SPEC_ORDNER` bezeichnet.

Quelltext 3.7: `source` und `rsource`

```
project
   |- Kconfig
   |- UO1
   |   |- Kconfig
   |- UO2
   |   |- Kconfig
            |_____________________
            |source  "UO1/Kconfig"| <=> "project/UO1/Kconfig"
            |rsource "UO1/Kconfig"| <=> "project/UO2/UO1/Kconfig"
            |        [...]        |
            |_____________________|
```

Quelltext 3.8: Transformationsregeln für `source`-Optionen

```
RSOURCE_PFAD ::= <PROJECT_ROOT>/<SPEC_ORDNER>/<PFAD_OHNE_GM>
-----------------------------------------------------------------
(o)source "<PFAD_OHNE_GM>" -> source "<PFAD_OHNE_GM>" (1:1)
(o)source "<PFAD_MIT_GM>"  -> source "<PFAD_OHNE_GM>" (1:n)
                              source "<PFAD_OHNE_GM>"
                              [...]
-----------------------------------------------------------------
(o)rsource "<PFAD_OHNE_GM>"-> source "<RSOURCE_PFAD>" (1:1)
(o)rsource "<PFAD_MIT_GM>" -> source "<RSOURCE_PFAD>" (1:n)
                              source "<RSOURCE_PFAD>"
                              [...]
-----------------------------------------------------------------
* Falls osource oder orsource keine Ergebnisse zurückliefern:
-> Die Zeile entfernen
```

Einige Kconfiglib-Varianten (vgl. Tabelle 3.4) können die Schlüsselwörter `grsource` und `gsource` zum Zweck der Abwärtskompatibilität einlesen und behandeln diese intern als die Optionen `orsource` beziehungsweise `osource`. Diese Schlüsselwörter wurden in Version 4.1.0 von Kconfiglib eingeführt, um die Glob-Muster und optionale Inklusion von Kconfiglib-Spezifikationen zu ermöglichen.[78] In Version 9.0.0 von Kconfiglib wurden sie zugunsten einer klaren Trennung zwischen verpflichtenden und optionalen Inklusionen entfernt.[79] Seitdem unterstützen `source`-Option sowie ihre Alternativen standardmäßig Glob-Mustern, wobei `source`-Option im Gegensatz zu `osource`- und `orsource`-Optionen mindestens einen Treffer verlangt. Als Transformationsregel wird eine Matching-Regel definiert. Das heißt, eingelesene `grsource`- und `gsource`-Optionen werden im Transformationsergebnis als `orsource` beziehungsweise `osource` ausgegeben und auch als solche behandelt (vgl. Regeln im Quelltext 3.8).

[78] https://github.com/ulfalizer/Kconfiglib/commit/daac69dc05217d024fbe021447629fa2b3d95b70

[79] https://github.com/ulfalizer/Kconfiglib/commit/7a428aa415606820a44291f475248b08e3952c4b

`def_*`-Attribute

Die Linux-Kconfig-Sprache erkennt ausschließlich `def_bool`- und `def_tristate`-Attribute. Die Kconfiglib-Varianten erweitern diese Menge durch `def_int`, `def_hex` und `def_string`. Diese Erweiterungen ermöglichen die gleichzeitige Festlegung des Typs und des Standarwerts einer `(menu)config`-Option. Die Transformationsregel für diese Attribute wird im Quelltext 3.9 veranschaulicht. Dabei werden der nach dem Schlüsselwort `def_*` angegebene Wert sowie gegebenenfalls die zugehörige if-Bedingung dem Wert des `default`-Attributs zugewiesen. In der Typdeklaration verbleibt nur der jeweilige Typ string, int oder hex.

Quelltext 3.9: Transformationsregel für `def_*`-Attribute

```
def_type ::= def_int | def_hex | def_string
type     ::= int | hex | string
-----------------------------------------------------------------
(menu)config <config_name>      ->  (menu)config <config_name>
  <def_type> <value>[if <expr>]       <type>
                                      default <value> [if <expr>]
```

`option`-Attribute

Die Linux-Kconfig-Sprache unterstützt derzeit keine `option`-Attribute. Da die betrachteten Kconfiglib-Varianten diese Erweiterungen unterstützen, werden im Folgenden die entsprechenden Transformationsregeln erläutert. Das `option env`-Attribut wurde ab Version 4.18 des Linux-Kernels entfernt.[80] Es diente dazu, den Wert einer Umgebungsvariable als Standardwert einer `(menu)config`-Option festzulegen. Die ursprüngliche Syntax `option env="<ENV>"` wurde durch eine direkte Referenzierung mittels `$(<ENV>)` ersetzt. Diese Änderung bildet die Grundlage der im Quelltext 3.10 dargestellten Transformationsregel. Dabei ist hervorzuheben, dass diese Transformation eine syntaktische Modifikation der Spezifikation bewirkt, da die betreffende Option nicht mehr vom Schreiben aus der `.config`-Datei ausgeschlossen wird, wie bei der Verwendung eines `option env`-Attributs.

Quelltext 3.10: Transformationsregel für `option env`-Attribut

```
(menu)config <config_name>     ->   (menu)config <config_name>
  option env="<ENV>"                    default $(<ENV>)
```

Die Attribute `option allnoconfig_y` und `option defconfig_list` werden seit Version 5.13 des Linux-Kernels lediglich in Form von Umgebungsvariablen mit den Namen `KCONFIG_ALLCONFIG` und `KCONFIG_DEFCONFIG_LIST` unterstützt.[81] Diese

[80] https://github.com/torvalds/linux/commit/104daea149c45cc84842ce77a9bd6436d19f3dd8; https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/kbuild/kconfig-language.txt?h=v4.18

[81] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/kbuild/kconfig.rst?h=v5.13; https://github.com/torvalds/linux/commit/ab838577aaaeda12242b7f1e2da3f25c9b4cec3a; https://github.com/torvalds/linux/commit/b75b0a819af9f78fc395b189cddd40f590194d20

Umgebungsvariablen sind eng mit dem Build-Prozess des Linux-Kernels verknüpft, der nicht im Fokus dieser Arbeit steht. Aus diesem Grund wurde auf die Definition der Transformationsregeln verzichtet, da sie für die Variabilitätsanalyse nicht wesentlich sind.

Das `option modules`-Attribut dient der Kennzeichnung derjenigen Option, die den Tristate-Wert `m` für alle anderen Konfigurationsoptionen zur Auswahl bereitstellt. Seit Version 5.13 des Linux-Kernels wurde die Syntax dieses Attributs auf das Schlüsselwort `modules` reduziert.[82] Daraus ergibt sich die Transformationsregel, nach der jedes Vorkommen von `option modules` durch `modules` ersetzt wird (vgl. Quelltext 3.11).

Quelltext 3.11: Transformationsregel für `option modules`-Attribut

```
1 (menu)config <config_name>     ->   (menu)config <config_name>
2   option modules                       modules
```

`named choice`-Option

Die Linux-Kconfig-Sprache hat die Unterstützung für `named choice`-Optionen ab Version 6.9 entfernt, da diese im Kernel keine Anwendung fanden.[83] Die Formulierung der Transformationsregel beansprucht mehrere Transformationsschritte. Als Voraussetzung müssen alle Defintionen einer solchen Option eingelesen und als die Wissensgrundlage vorbereitet werden. Diese umfasst den extrahierten Auskunft darüber, wie viele Definitionen der betrachteten `named choice`-Option gibt es, welche Attribute und Elemente umfassen diese, sowie welche Abhängigkeitsbedingungen aus den eigenen `depends on`-Abhängigkeiten sowie propagierten Abhängigkeiten sollen berücksichtigt werden.

Das Ziel der Transformation besteht darin, alle zugehörigen `config`-Optionen unter einer gemeinsamen `choice`-Option zusammenzufassen, indem die erste `named choice`-Definition entsprechend erweitert wird. Die hierführ vorgesehene Reihenfolge folgt der Struktur einer `choice`-Option. Das bedeutet, dass zunächst die Attribute aller Defintionen zusammengeführt werden, gefolgt von den Elementen sämtlicher Definitionen. Abschließend werden alle weiteren Defintionen, die nicht der ersten entsprechen, aus der Ausgabe entfernt. Der Optionsname entfällt ebenfalls.

Bei der Verarbeitung von Attributen dieser Optionen kommt Transformationsregel 3.13 zur Anwendung, da eine `named choice`-Option dieselben Attribute und Elemente wie eine `choice`-Option besitzen kann. Alle `depends on`-Attribute, die nicht aus der ersten Definition stammen, werden nicht direkt zur zusammengeführten Definition hinzugefügt, sondern entsprechend auf die in der Definition enthaltenen Elemente propagiert. Dadurch wird das `depends on`-Attribut aus der zweiten Definition im Tranformationsbeispiel 3.12 ausschließlich auf die `config`-Optionen `G_24` und `G_25` propagiert (vgl. Quelltext 3.12, Zeilen 13 und 15, rechte Seite).

Ein Tranformationsbeispiel ist in Quelltext 3.12 dargestellt, wobei Eingabe und Ausgabe visuell durch das Symbol `|` getrennt sind. Aus den zwei Defintionen der `named`

[82] https://github.com/torvalds/linux/commit/6dd85ff178cd76851e2184b13e545f5a88d1be30
[83] https://github.com/torvalds/linux/commit/c83f020973bc72d9eec65474d8c47495191aef20

`choice`-Option mit dem Namen `GRAFIK_TYPE` wird zunächst das `prompt`-Attribut übernommen, gefolgt von drei Attributen der zweiten Definition. Anschließend werden die Elemente der ersten Definition verarbeitet, bevor die Elemente der zweiten Defintion gemeinsam mit den propagierten Abhängigkeiten zum Tranformationsergebnis hinzugefügt werden.

Quelltext 3.12: Transformationsbeispiel für `named choice`-Option

```
1  # in foo/Kconfig          ->| # in foo/Kconfig
2  choice GRAFIK_TYPE          | choice
3    prompt "wähle Typ aus"    |   prompt "wähle Typ aus"
4    config G_22               |   default G_24 if ARCH32 && OPTION_A
5      bool "Typ 22"           |   help
6    config G_23               |     Hilfetext in
7      bool "Typ 23"           |     mehreren Zeilen
8  [...]                       |  config G_22
9                              |     bool "Typ 22"
10 # in bar/Kconfig            |  config G_23
11 if OPTION_A                 |     bool "Typ 23"
12 choice GRAFIK_TYPE          |  config G_24
13   default G_24              |     bool "Typ 24" if ARCH32 && OPTION_A
14   depends on ARCH32         |  config G_25
15   help                      |     bool "Typ 25" if ARCH32 && OPTION_A
16     Hilfetext in            |
17     mehreren Zeilen         | [...]
18   config G_24               |
19     bool "Typ 24"           | # in bar/Kconfig
20   config G_25               | if OPTION_A
21     bool "Typ 25"           |
22 [...]                       | [...]
23 endif                       | endif
```

Attribute und Elemente von `choice`-Optionen

Das `optional`-Attribut für `choice`-Optionen wurde ab Version 6.10 des Linux-Kernels entfernt, da es zuvor keine funktionale Verwendung hatte.[84] Darüber hinaus wurde explizite Typdeklaration im Kontext von `choice`-Optionen entfernt.[85] Dies ist darauf zurückzuführen, dass die Linux-Kconfig-Sprache inzwischen ausschließlich `choice`-Optionen vom Typ bool unterstützt, sodass eine zusätzliche Typangabe als redundant eingestuft wurde. Kconfiglib-Varianten erlauben `menuconfig`-Optionen als Elemente einer `choice`-Option, während die Linux-Kconfig-Sprache lediglich Optionen `config`, `comment` und `if` unterstützt.

Die Transformationsregeln berücksichtigen diese Einschränkungen und führen somit zu einer bewussten semantischen Anpassung der Kconfiglib-Spezifikationen (vgl. Quelltext 3.13). Konkret werden sowohl die explizite Typdeklaration als auch das

[84] https://github.com/torvalds/linux/commit/6a1215888e23aa9fbc514086402f04708c84f454
[85] https://github.com/torvalds/linux/commit/bea2c5ef789a37bace99f2f45eeef3be4559b228

`optional`-Attribut entfernt. Die Auswahloptionen werden auf `config`-Optionen vom Typ bool beschränkt, während `menuconfig`-Optionen außerhalb der `choice`-Optionen positioniert werden. Diese `mencuonfig`-Optionen übernehmen dabei die Abhängigkeiten der `depends on`-Attribute der `choice`-Option, in deren Kontext sie zuvor definiert waren. Dadurch wird die Propagierung von Abhängigkeiten berücksichtigt. Darüber hinaus ersetzt der Prototyp eine mit dem `Inline-Prompt` verbundene Typdeklaration durch das Schlüsselwort `prompt`, gefolgt von dem in Anführungszeichen gesetzten Prompt-Text (s. Quelltext 3.13, Zeile 2, rechte Seite).

Quelltext 3.13: Transformationsregel für `choice`-Option

```
1  choice <config_name>            ->|choice
2   (bool | tristate) ["<STRING>"]  | prompt "<STRING>" if [<expr>]
3   optional                        | default <config_name>if[<expr>]
4   prompt "<STRING>" if [<expr>]   | depends on <expr>
5   default <config_name> if[<expr>]| help
6   depends on <expr>               |   <MULTILINE STRING>
7   help                            | config <config_name>
8     <MULTILINE STRING>            |    bool "<STRING>"
9   config <config_name>            |    depends on <exp>
10    (bool | tristate) "<STRING>"  | if <expr>
11    depends on <expr>             |   config_opt
12  menuconfig <config_name>        | endif
13    (bool | tristate) "<STRING>"  |endchoice
14    depends on <expr>             |
15  if <expr>                       |menuconfig <config_name>
16    (config_opt | menuconfig_opt) |  bool  "<STRING>"
17  endif                           |  depends on <expr>
18 endchoice                        |  depends on <expr>
19                                  |  [...]
```

Quelltext 3.14: Transformationsregel für `configdefault`-Option

```
1  # 1. Defintion                | # 1. Defintion
2  (menu)config <config_name>   | (menu)config <config_name>
3    default <value> if <expr>  |   default <value> if <expr>
4                               |   default <value> if <expr>
5  # 2. Defintion               |   default <value> if <expr>
6  if <expr>                    |
7  [...]                        | # 2. Defintion
8  configdefault <config_name>  | if <expr>
9    default <value> if <expr>  |   [...]
10 endif                        | endif
11                              |
12 # 3. Defintion               | # 3. Defintion
13 [...]                        | [...]
14 configdefault                |
15   default <value> if <expr>  |
```

`configdefault`-Option

Die `configdefault`-Option ist eine projektspezifische Erweiterung von ZRTOS-Kconfiglib und dient der Definition zusätzlicher Standardwerte für eine gleichnamige Option (vgl. `configdefault`-Option 3.3.2). Die Transformationsregel (vgl. Quelltext 3.14) setzt voraus, dass die erste `(menu)config`-Option mit eigenen `default`-Attributen als Zielort der Transformation identifiziert wurde. Zusätzlich werden die `default`-Attribute aller zugehörigen `configdefault`-Definitionen extrahiertet und um den propagierte Abhängigkeiten aus umschließenden Elementen ergänzt. Die so vorbereiteten Zeilen können unter der ermittelten Defintion der `(menu)config`-Option eingefügt werden, während die ursprünglichen `configdefault`-Optionen aus der Ausgabe entfernt werden.

`warning`-Attribut und `set`-Attribute

Die Attribute `warning`, `set` und `set default` sind spezifische Erweiterungen der Kconfiglib-Variante esp-idf-kconfig und in der Linux-Kconfig-Sprache nicht vorgesehen. Das `warning`-Attribut stellt eine rein benutzerinteraktive Erweiterung dar, deren semantisch äquivalente Transformation eine funktionale Erweiterung der Konfigurations-Frontends des Linux-Kconfig-Konfigurationssystems erfordern würde. Da keine Sprachelemente der Linux-Kconfig-Sprache eine Abfrage von Benutzerbestätigungen bei Wertänderungen auslösen, werden `warning`-Attribute aus dem weiteren Transformationsumfang ausgeschlossen.

Die Attribute `set` und `set default` ermöglichen es booleschen `(menu)config`-Optionen, aktiv auf die Standardwerte nicht-boolescher Optionen einzuwirken. Ein solcher Einfluss ist im Sprachumfang der Linux-Kconfig-Sprache nicht vorgesehen. Aus diesem Grund werden auch diese Attribute nicht in den Transformationsumfang des angestrebten Transformationsprototyps aufgenommen.

3.4.3 Diskussion

Die Ausarbeitung der Transformationsregeln hat gezeigt, dass eine semantisch korrekte Umsetzung dieser Regeln nur unter Berücksichtigung des gesamten Konfigurationsraums möglich ist. Insbesondere für die Transformation der Optionen `configdefault` und `named choice` ist es erforderlich, zunächst sämtliche zugehörigen Definitionen vollständig zu erfassen und als Wissensbasis zu extrahieren.

Darüber hinaus wurden die direkten semantischen Auswirkungen einzelner Transformationsregeln deutlich. Die Transformation des `option env`-Attributs führt zu einer Erweiterung der Menge von `(menu)config`-Optionen, die in der `.config`-Datei aufgenommen werden. Dies bedeutet, dass mehrere Optionen bei der Konfiguration der gewünschten Systemvariante berücksichtigt werden. Hingegen führt die Transformation von `(named) choice`-Optionen zu einer Einschränkung der erlaubten Attribute und Elemente dieser Optionen, wodurch beispielsweise keine optionalen `choice`-Optionen existieren und eine verpflichtende Auswahl eines Elements dieser Option notwendig ist.

Zugleich wurde erkennbar, dass sich die zunächst als einfach erscheinenden Transformationsschritte vor allem aufgrund der notwendigen Berücksichtigung propagierter

Abhängigkeiten als deutlich komplexer erweisen. Diese Komplexität sowie die erforderliche Vorverarbeitung aller relevanten Informationen führen dazu, dass der Implementierungsaufwands der Transformationsregeln insgesamt höher einzuschätzen ist als ursprünglich angenommen.

3.5 Zusammenfassung

In diesem Kapitel wurde zunächt die qualitative Analyse der Motivation und Funktionsweise der Kconfiglib-Bibliothek dargestellt. Darauf aufbauen wurde ihre Verbreitung in Open-Source-Projekten untersucht, wobei mehr als 1600 Repositories (davon 734 öffentlich sichtbaren) auf GitHub identifiziert wurden. Die öffentlich sichtbaren Repositories wurden anschließend eigenständig nach deren Relevanz und Hauptzweck kategorisiert und gefiltert. Von insgesamt 14 final ausgewerteten Projekten weisen elf eine aktive Verwendung von Kconfiglib auf. Kconfiglib wird in den meisten Projekten unverändert eingesetzt, wobei die Entscheidung für diese Bibliothek auf der einfachen Integrationsmöglichkeit, der plattformübergreifenden Einsetzbarkeit und dem bereitgestellten Funktionsumfang beruht.

Insgesamt wurden vier Kconfiglib-Varianten identifiziert, die in unterschiedlichem Ausmaß die Linux-Kconfig-Sprache erweitern. Die Funktionsweise der identifizierten Kconfiglib-Erweiterungen wurde analysiert, und die Transformationsregeln für die einzelne Kconfiglib-Erweiterungen ausgearbeitet. Die Ergebnisse verdeutlichen die Komplexität der Transformationsregeln sowie deren unmittelbare semantische Auswirkungen auf die resultierenden Ausgabespezifikationen.

4. Implementierung

In diesem Kapitel stellen wir die prototypische Umsetzung der in Kapitel 3 ausgearbeiteten Transformationsregeln vor. Dabei erläutern wir die getroffenen Entscheidungen hinsichtlich der gewählten Architektur, der implementierten Funktionalitäten und des verwendeten Technologiestack. Zudem beschreiben wir die Herausforderungen, denen wir während der Implementierung begegnet sind.

Der Transformationsprototyp verfolgt das Ziel, Kconfiglib-basierte Spezifikationen in äquivalente Linux-Kconfig-basierte Spezifikationen zu überführen. Zu diesem Zweck wurden mehrere Implementierungsentscheidungen (IE) getroffen, die in den folgenden Abschnitten näher erläutert werden:

- IE_1: Wie ist der Transformationsprototyp strukturiert?
- IE_2: Welche Eingabe benötigt der Transformationsprototyp?
- IE_3: Wie ist die Ausgabe des Transformationsprototyps strukturiert?
- IE_4: In welcher Reihenfolge erfolgt die Transformation?
- IE_5: Welcher Technologiestack wurde bei der Implementierung verwendet?
- IE_6: Welche Herausforderungen sind bei der Implementierung aufgetreten?

4.1 IE_1 Architektur

Der entwickelte Prototyp ist als Python-basiertes Projekt umgesetzt, dessen Bestandteile sich klar in der Struktur des Quellcodes widerspiegeln (vgl. Quelltext 4.1). Der Ordner `core` enthält die zentralen Python-Skripte, welche die Hauptlogik der Transformation implementieren (vgl. IE_2-IE_4), während im Ordner `external` die Git-Submodule der einzelnen Kconfiglib-Varianten eingebunden sind. Ergänzend umfasst das Projekt sowohl die Dokumentation der durchgeführten Experimente als auch technische Hinweise zur Ausführung des Prototyps in Python-virtuellen Umgebungen. Diese Inhalte sind in den entsprechenden Verzeichnissen `test_dir_` und `transform_projects` abgelegt. Die Python-virtuellen Umgebungen wurden bereits zu Beginn der Implementierung als zusätzliche Absicherungsmaßnahme eingeführt, um potenzielle Konflikte zwischen den unterschiedlichen Projektvoraussetzungen der betrachteten Systeme, wie Zephyr oder ESP-IDF, zu vermeiden.[86]

Die Integration der Kconfiglib-Varianten als Git-Submodule in die Projektstruktur ist eine direkte Konsequenz der Analyse in Kapitel 3. Diese Analyse hat gezeigt, dass aufgrund projektspezifischer Erweiterungen kein einzelner Parser in der Lage ist, alle identifizierten Varianten von Kconfiglib-Spezifikationen vollständig zu verarbeiten (vgl. Diskussion 3.3.3). Vor diesem Hintergrund wurde bewusst darauf verzichtet, einen neuen universellen Parser zu entwickeln. Stattdessen verfolgt der Transformationsprototyp den Ansatz, jeweils den für die zu transformierende Spezifikation geeigneten Parser einzubinden. Auf diese Weise entfällt nicht nur der erhebliche Implementierungsaufwand, sondern auch die Notwendigkeit, zukünftige Änderungen der verschiedenen Kconfiglib-Varianten manuell in einem neu entwickelten Parser nachzubilden.

Quelltext 4.1: Struktur des Transformationsprototyps

```
prototyp
   |- core ____________________________________________
   |   |- pick_parser.py          |- zrtos_parser.py
   |   |- kconfig_writer.py       |- z_kconfiglib_parser.py
   |   |- excel_writer.py         |- espidf_parser.py
   |   |- time_writer.py
   |   |- transform_prototyp.py   [Hauptlogik]
   |   |- workflow_test.py
   |   |- target_workflow.py
   |   |- [...] ___________________________________________
   |- external
   |   |- ZephyrRTOS
   |   |- ZephyrKconfiglib
   |   |- ESPIDFKconfig
   |- test_dir_
   |- transform_projects
```

[86] https://docs.python.org/3/tutorial/venv.html

4.2 IE_2 Transformationseingabe

Es wird davon ausgegangen, dass der Dateipfad zum gesamten Projektverzeichnis dem Transformationsprototyp als manuelle Eingabe vorliegt. Der Transformationsprototyp setzt die Umgebungsvariable `SRCTREE` auf das angegebene Projektverzeichnis und verwendet die Haupt-Kconfiglib-Spezifikation des betrachteten Projekts als Startpunkt der Transformation. Diese Haupt-Kconfiglib-Spezifikation befindet sich in der Regel im Wurzelverzeichnis des Projekts, definiert den Text in der Titelleiste des Konfigurations-Frontends und bindet weitere, für das Projekt notwendige Spezifikationen ein.

Für das Einlesen des Konfigurationsraums eines Projekts ist ein Parser der jeweils passenden Kconfiglib-Variante erforderlich. Die Auswahl des gewünschten Parsers der Kconfiglib-Variante wird durch das Skript **`pick_parser.py`** gekapselt. Dieses Skript wird über Parameter gesteuert, die festlegen, welche Kconfiglib-Variante (vgl. `external` in Quelltext 4.1) für die Transformation verwendet werden soll.

Der Aufruf der ausgewählten Parser erfolgt über Skripte wie **`zrtos_parser.py`**, welche die `Kconfig`-Klasse der Kconfiglib-Variante aufrufen und eine Instanz dieser Klasse erstellen (vgl. `Kconfig`-Klasse 3.1.2). Dabei werden die internen Verarbeitungsschritte der Kconfig-Klasse gestartet, die untere anderem mithilfe des entsprechenden Parsers den gesamten Konfigurationsraum des Projekts einlesen und finalisieren. Anschließend speichert der Transformationsprototyp die gewonnene Parser-Ausgabe strukturiert ab. Dadurch werden Informationen wie hierarchische Strukturen, Definitionen und Abhängigkeiten einzelner Konfigurationsoptionen für die weiteren Transformationsschritte zugänglich gemacht.

Die Ausgabe des Parsers liegt in Form interner Strukturen vor, die nicht unmittelbar als Ausgabespezifikation verwendet werden können, ohne zuvor in eine textuelle Darstellung überführt zu werden. Aus diesem Grund extrahiert der Transformationsprototyp zunächst die für die Transformation relevanten Parser-Informationen und speichert sie in der internen Datenstruktur **`ExtParserContext`**. Erst auf dieser Grundlage können die einzelnen Transformationsregeln angewendet werden. Hierbei werden unter andem propagierte Abhängigkeiten aus den übergeordneten Konfigurationsoptionen sowie Elemente und Attribute aus verschiedenen Definitionen einer `named choice`-Optionen extrahiert.

4.3 IE_3 Transformationsausgabe

Die gewünschte Transformationsausgabe kann sowohl durch eine in-place- oder out-of-place-Transformation erzielt werden. Bei der in-place-Transformation werden die Kconfiglib-Spezifikationen des Projekts nach dem Einlesen intern transformiert und anschließend durch die Transformationsausgabe überschrieben. Die out-of-place-Transformation hingegen liest die Spezifikationen des Projekts zunächst ein, legt die transformierte Ausgabe aber in einem neuen Ausgabeverzeichnis ab, das die Struktur des Eingabeprojekts widerspiegelt. Aufgrund der angestrebten Variabilitätsanalyse (FF_2-FF_3) wurde die out-of-place-Transformation bevorzugt, da sie eine klare Trennung von Ein- und Ausgabe ermöglicht, das betrachtete Projekt nicht modifiziert und die Ausgabe ausschließlich auf dessen Kconfig-Spezifikationen reduziert.

Zur nachvollziehbaren Dokumentation der Transformationsschritte werden die Standardausgabe (`stdout`) und die Standardfehlerausgabe (`stderr`) während der Transformation in eine Log-Datei umgeleitet. Dabei wird mittels des Skripts **`time_writer.py`** auch die Ausführungszeit gemessen und in der Log-Datei festgehalten. Die erhobenen statistischen Informationen zur Anzahl der verwendeten Kconfiglib-Erweiterungen werden mittels des Skripts **`excel_writer.py`** in einer Excel-Datei gespeichert.

4.4 IE$_4$ Transformation

Der Transformationsprototyp erzeugt während der Transformation eine Log-Datei, die die vorgenommenen Transformationsschritte dokumentiert. Zunächst hält die Log-Datei fest, welches Projekt in welches Zielverzeichnis transformiert wird und welcher Parser zum Einlesen des Projekts verwendet wird. Abhängig vom gewählten Detailgrad listet die Log-Datei zudem die gewonnenen Parser-Informationen sowie die einzelnen Schritte bei der Extraktion interner Datenstrukturen auf. Der Quelltext 4.2 zeigt exemplarisch einen vereinfachten Ausschnitt einer solchen Log-Datei.[87]

Quelltext 4.2: Ausschnitt einer Log-Datei

```
1  TRANSFORMATION PROTOTYP LOG
2  Root: [...]/wsMA_prototyp/test_dir_/transform_source
3  Main file: Kconfig
4  Output dir: [...]/wsMA_prototyp/test_dir_/transform_source_output
5  ==================================================
6  START using parser:
7    spec_version: ZRTOS
8    kconfig_folder: [...]/external/ZephyrRTOS/scripts/kconfig
9    kconfiglib_version: <module 'kconfiglib' from '[...]'>
10 1. Parser Output:          [...]
11 2. Bild ExtParserContext   [...]
12 3. Filter ExtParserContext [...]
```

Als Grundlage für die Transformation werden neben den Parser-Informationen und den daraus abgeleiteten internen Strukturen die drei Hilfsklassen **`KconfigLine`**, **`KconfigWriter`** und **`KconfigReader`** aus dem Skripts **`kconfig_writer.py`** verwendet. Die Klasse `KconfigLine` bestimmt anhand des einleitenden Schlüsselworts den Typ einer Zeile und extrahiert den zugehörigen Kontext, beispielsweise den Namen der Konfigurationsoption. `KconfigReader` übernimmt den Dateipfad einer Spezifikation als Eingabe und gibt deren Zeilen als Liste von `KconfigLine`-Instanzen zurück. `KconfigWriter` nimmt hingegen eine Liste von `KconfigLine`-Objekten entgegen und schreibt diese zeilenweise in die durch einen Dateipfad angegebene Ausgabespezifikation. Beide Klassen nutzen dabei Python-eigene Standardbibliotheken zum Lesen und Schreiben von Dateien.

Das Skript **`transformation_prototyp.py`** beziehungsweise die zugehörige Klasse **`KconfigTransformer`** orchestriert die Transformation des gesamten Projekts und

[87] https://github.com/LjuDordevic/wsMA_prototyp/blob/main/test_dir_/transform_source_output/transform.log

greift dazu auf die Methoden der Hilfsklassen zurück. Es extrahiert zunächst alle Spezifikationen des betrachteten Projekts aus den Parser-Informationen, sodass ihre Verarbeitung in der gleichen Reihenfolge erfolgt wie die tatsächliche Einbindung im Konfigurationsraum. Anschließend werden die Spezifikationen sequentiell eingelesen und über `KconfigReader` in `KconfigLine`-Objekte überführt, wodurch jede Zeile typisiert und kontextualisiert vorliegt.

Im nächsten Schritt verarbeitet der Transformationsprototyp diese Zeilen sequenziell und wendet, abhängig vom jeweiligen Zeilentyp, die entsprechenden Transformationsregeln (vgl. Kapitel 3.4) an. Dadurch werden Zeilen entweder unverändert übernommen, entfernt oder durch zusätzliche Angaben beziehungsweise weitere Zeilen ergänzt. Das Ergebnis der Transformation einer Spezifikation wird schließlich an `KconfigWriter` übergeben und zeilenweise in die jeweilige Ausgabespezifikation geschrieben. Parallel dazu wird für jede transformierte Spezifikation die im Quelltext 4.3 dargestellte Log-Struktur erstellt.

Quelltext 4.3: Ausschnitt einer Log-Datei - Fortsetzung

```
4. Transform all files [...]
Reader found: 3 lines in [...]/test_dir_/transform_source/Kconfig
start transforming 3 input lines
    FILE LOG ------------------------------------------
    Reader input                3 lines
    ----------------------------------------------------
    All source without glob:    1
    All source using glob:      0
    All osource_keywords:       0
    All rsource_keywords:       0
    [...]
    ----------------------------------------------------
    Transformer Output:         3 lines
    ----------------------------------------------------
        Added new bc of def_*:          0
        Added new bc of glob:           0
        [...]
        Removed bc of config_default:   0
        Removed bc of named choice:     0
        Removed no match for o(r)source: 0
        [...]
        Done writting 3 lines in
        [...]/test_dir_/transform_source_output/Kconfig [...]
```

Abbildung 4.1 visualisiert die zuvor vorgestellten Transformationsschritte des Prototyps in zusammengefasster Form. Dieser Transformationsablauf spiegelt sich in den Skripten **`workflow_test.py`** und **`target_workflow.py`** wieder, die zum Testen einzelner Transformationsregeln (FF4) beziehungsweise zur Verarbeitung betrachteter Projekte (FF1) verwendet werden. Der Hauptunterschied zwischen diesen Skripten liegt im Detaillierungsgrad der Protokollierung.

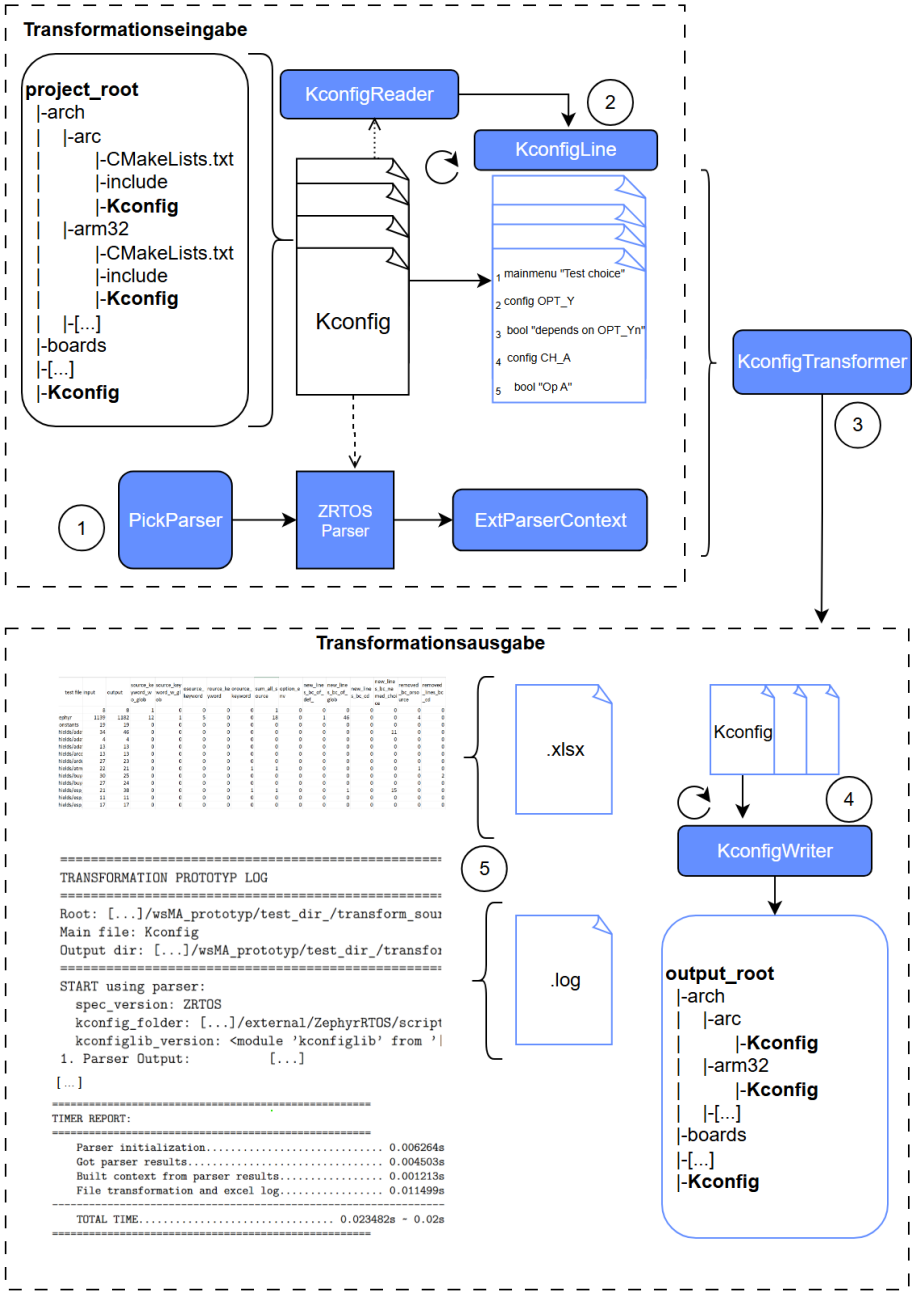

Abbildung 4.1: Transformationsschritte

4.5 IE$_5$: Technologiestack

Neben den externen Kconfiglib-Varianten, die als Submodule in den Transformationsprototyp eingebunden sind, verwendet der Prototyp Python-eigene Standardbibliotheken zum Einlesen der Spezifikationen und zum Schreiben der Ausgabespezifikationen. Für die strukturierte Ablage der während der Transformation erhobenen Statistiken wird zusätzlich die Bibliothek `openpyxl` eingesetzt, um diese in die Excel-Datei zu exportieren. Der eingesetzte Technologiestack wurde bewusst schlank gehalten, da der Fokus auf der Implementierung von Transformationsregeln liegt und die zentrale Funktionalität des Prototyps auf der Extraktion und Verarbeitung von Parser-Informationen basiert. Hierfür ist ein detailliertes Verständnis der internen

Struktur der jeweiligen Kconfiglib-Varianten notwendig. Als Programmiersprache wurde Python gewählt, da auch die eingesetzten Kconfiglib-Varianten in Python implementiert sind und somit eine direkte Nutzung ihrer Datenstrukturen möglich ist.

4.6 IE$_6$ Herausforderungen

Die in Kapitel 3 abgeleiteten Transformationsregeln bilden die Grundlage der Logik des Transformationsprototyps. Eine zentrale Herausforderung bestand in der Verarbeitung von `source`-Optionen und ihren Alternativen, insbesondere in der korrekten Auflösung relativer Pfade sowie von Dateipfaden, die durch Glob-Muster spezifiziert sind. Fehler in diesem Schritt hätten unmittelbare Auswirkungen auf die Vollständigkeit, Korrektheit und die Struktur des transformierten Konfigurationsraums.

Weitere Herausforderungen haben sich bei der Umsetzung der Transformationsregeln für die Optionen `configdefault` und `named choice` ergeben. Dies ist der Komplexität der Transformationsregeln geschuldet, da diese eine umfangreiche Vorbereitung und Auswertung von Parser-Informationen erfordern. Insbesondere musste sichergestellt werden, dass Konfigurationsoptionen aus allen Definitionen einer `named choice`-Optionen beziehungsweise `default`-Attribute aus allen Definitionen einer `configdefault`-Option korrekt zusammengeführt werden. Dabei mussten die propagierten Abhängigkeiten übergeordneten Elemente extrahiert und konsistent in die Ausgabe übernommen werden. Die Entwicklung dieser Transformationslogik erfolgte iterativ und machte wiederholte Tests sowie umfangreiche Debugging-Schritten erforderlich.

4.7 Zusammenfassung

Im Folgenden werden die getroffenen Implementierungsentscheidungen zusammengefasst.

IE$_1$ Wie ist der Transformationsprototyp strukturiert?

Als Hauptbestandteile umfasst er eigene Python-Skripte sowie externe als Git-Submodule eingebundenen Kconfiglib-Varianten. Dabei dienen die Python-Skripte der Umsetzung von Transformationsregeln und die Git-Submodule ermöglichen die Einbindung des Parsers der gewünschten Kconfiglib-Varianten.

IE$_2$ Wie können alle notwendigen Informationen gewonnen werden und können diese direkt als Transformationsausgabe verwendet werden?

Der Transformationsprototyp greift auf die von der Klasse `Kconfig` bereitstellten Parser-Informationen zurück. Zu diesem Zweck bindet er eine geeignete Kconfiglib-Variante ein und erzeugt eine neue Instanz der zugehörigen `Kconfig`-Klasse. Diese Instanz dient als zentrale Zugriffsschnittstelle auf den vollständigen Konfigurationsraum der zu transformierenden Spezifikation.

Die auf diese Weise gewonnenen Parser-Informationen können nicht direkt als Transformationsausgabe verwendet werden. Sie liegen in der Form eines Graphs vor, der den hierarchischen Menübaum und zahlreichen Datenstrukturen zur Modellierung der Kconfiglib-Spezifikation umfasst. Der Transformationsprototyp filtert, kombiniert und extrahiert diese bereitgestellten Informationen, um die notwendige Grundlage für die Transformation bereitzustellen.

IE$_3$ Wie ist die Ausgabe des Transformationsprototyps strukturiert?

Es erfolgt eine out-of-place-Transformation, bei der die eingelesenen Spezifikationen transformiert in einem neuen Verzeichnis abgespeichert werden. Dabei spiegelt die Verzeichnisstruktur der Ausgabe die Verzeichnisstruktur der Eingabe wider. Die während der Transformation gewonnenen Statistiken und die vorgenommenen Schritte werden in einer gewünschten Excel-Datei beziehungsweise einer Log-Datei abgelegt.

IE$_4$ In welcher Reihenfolge findet die Transformation statt?

Zunächst ermöglicht der gewählte Parser den Zugriff auf die Informationen des Konfigurationsraums eines Projekts. Darauffolgend stellt der Transformationsprototyp die interne Datenstruktur `ExpParserOutput` bereit und die Klasse `KconfigReader` überführt die eingelesenen Spezifikationen zeilenweise in Instanzen der Klasse `KconfigLine`. Die internen Methoden der Klasse `KconfigTransformer` wenden die entsprechenden Transformationsregeln an und verarbeiten dabei sowohl die vorbereiteten Spezifikationen als auch die extrahierten Parser-Informationen. Abschließend werden die transformierten Spezifikationen sequentiell durch die Klasse `KconfigWriter` in die Ausgabestruktur geschrieben. Zusätzlich entstehen als Endergebnis eine Log-Datei und eine Excel-Datei.

IE$_6$ Welcher Technologiestack wurde bei der Implementierung verwendet?

Der Transformationsprototyp wurde vollständig in Python implementiert und ist im zugehörigen GitHub-Repository `wsMA_prototyp` verfügbar.[88] Die Ausführung erfolgt innerhalb einer Python-virtuellen Umgebung unter Ubuntu, das über WSL2 (Windows-Subsystem für Linux) betrieben wird.

IE$_7$ Welche Herausforderungen sind bei der Implementierung aufgetreten?

Den größten Implementierungsaufwand verursachte die Umsetzung der Transformationsregeln für die Optionen `configdefault` und `choice` beziehungsweise `named choice`. Diese Elemente haben eine deutlich komplexere Verarbeitung als andere Elemente, da sie umfangreiche Extraktion der notwendigen Parser-Informationen erfordern.

5. Evaluierung

In diesem Kapitel wird der Transformationsprototyp anhand von vier Forschungsfragen evaluiert.

5.1 Forschungsfragen

Um die Anwendbarkeit und Auswirkungen des Transformationsprototyps auf unterschiedliche Projekte, sowie die Relevanz und Korrektheit der implementierten Transformationsregeln im Hinblick auf die Verbreitung von Kconfiglib-Erweiterungen sicherzustellen, werden die folgenden Forschungsfragen beantwortet:

- FF_1: Auf welche Projekte lässt sich der Transformationsprototyp anwenden?
- FF_2: Wie häufig verwenden bestehende Projekte Kconfiglib-Erweiterungen?
- FF_3: Wie wirkt sich die Transformation auf die Größe der resultierenden Spezifikationen aus? Wie lange dauert die Transformation?
- FF_4: Werden Features und deren Abhängigkeiten durch die Transformation korrekt behandelt?

5.2 Methodik

Der entwickelte Transformationsprototyp dient als zentrales Werkzeug zur Evaluierung der Forschungsfragen. Der Quellcode ist in dem Repository `wsMA_prototyp` verfügbar, dessen Verzeichnisse im Folgenden referenziert werden.[88] Er wird in einer Python-virtuellen Umgebung unter Ubuntu ausgeführt, das über WSL2 betrieben wird. Diese Umgebung erleichtert eine reproduzierbare Ausführung der Transformation und läuft auf einem HP ENVY x360 Laptop mit einem 12th Gen Intel Core i7-1260P (2.10 GHz) Prozessor, Intel Iris Xe Graphics und 16 GB Arbeitsspeicher.

[88] https://github.com/LjuDordevic/wsMA_prototyp

5.3 FF_1 Anwendbarkeit

Dieses Kapitel untersucht die Anwendung des Transformationsprototyps auf Projekte, die auf Kconfiglib basieren und im Kapitel 3.2 identifiziert wurden.

Forschungsfrage 1

Auf welche Projekte lässt sich der Transformationsprototyp anwenden?

5.3.1 Methodik

Eine Untersuchung aller in Kapitel 3.2 elf identifizierten Projekte würde den Rahmen dieser Arbeit sprengen. Daher wurde zunächst eine Auswahl nach Relevanz vorgenommen und anhand einer Stichprobe gearbeitet. Diese Relevanz wird auf Basis von GitHub-Statistiken bestimmt. Die Projekte werden nach der Anzahl von Sterne-Markierungen auf GitHub sortiert, wobei nur jene berücksichtigt werden, die mehr als 5000 dieser Markierungen haben (vgl. Tabelle 5.1). Zur finalen Auswahl zählen somit die Projekte ESP-IDF-Framework, Zephyr, RT-Thread, PX4-Autopilot und RIOT.

Tabelle 5.1: FF_1 - Projekte mit zugehörigen GitHub-Statistiken

Projekt	Mitwirkende	Sterne	Forks
ESP-IDF-Framework	960	16700	7980
Zephyr	2990	13800	8300
RT-Thread	780	11570	5300
PX4-Autopilot	760	10500	470
RIOT	380	5600	2000
Sming	90	1540	340
sdk-nrf	440	1200	1400
TuyaOpen	15	1100	190
legato-af	50	150	120
TF-M	200	30	20
nxp_tf-m	130	1	1

Diese Forschungsfrage analysiert die fünf ausgewählten Projekte systematisch und untersucht, ob und in welchem Umfang der Transformationsprototyp angewendet werden kann. Primär benötigt der Prototyp den Pfad zu dem Wurzelverzeichnis des jeweils betrachteten Projekts. Deswegen werden die zu betrachtenden Projekte manuell aus ihren jeweiligen GitHub-Repositories lokal bezogen und die Testumgebung manuell vorbereitetet. Eine detaillierte Beschreibung dieser Vorbereitung ist im Verzeichnis `transform_projects` des GitHub-Repository `wsMA_prototyp` dokumentiert.

Jedes Projekt enthält ein eigenes Unterverzeichnis, das der Dokumentation des durchgeführten Experiments dient und vier Bestandteile umfasst (vgl. Quelltext 5.1). Zunächst sind zwei Unterverzeichnisse enthalten, welche jeweils die automatisch erzeugte Transformationsausgabe sowie die zugehörigen Log- und Excel-Dateien

enthalten. Die beiden übrigen Bestandteile sind manuell gepflegte Textdateien, die als technische Anleitungen zur Replikation der Experimente dienen.

Das Verzeichnis mit dem Suffix `output` enthält die Transformationsausgabe, das heißt alle automatisch transformierten Spezifikationsdateien des jeweiligen Projekts. Der Transformationsprototyp wurde insgesamt dreimal auf jedes Projekt angewendet, um den Median der Transformationszeit (FF_3 5.5) zu bestimmen. Diese dreifache Ausführung ist erforderlich, da der Median erst ab mindestens drei Messwerten eindeutig bestimmt werden kann und zugleich der Einfluss von Ausreißern und zufälligen Laufzeitschwankungen reduziert wird. Die drei Log- und Excel-Dateien, die der Transformationsprototyp automatisch erzeugt hat, sind im Verzeichnis mit dem Suffix `log` abgelegt.

Die manuell gepflegten Textdateien beschreiben die Vorbereitung der Testumgebung sowie die Evaluierung manueller Nacharbeiten. Die Textdatei mit dem Suffix `setup` dokumentiert die für die Erstellung der Testumgebung relevanten Schritte. Im Unterverzeichnis mit dem Suffix `check` ist eine weitere Textdatei enthalten, in der die bei der Evaluierung manueller Nacharbeiten mittels des `mconf`-Konfigurations-Frontends durchgeführten Schritte festgehalten sind.

Quelltext 5.1: Ausschnitt der Dokumentationsstruktur

```
wsMA_prototyp
 |-transform_projects
    |-transform_zrtos
    |- [...]
    |-transform_px4
    |  |- PX4_output
    |  |- PX4_log
    |  |- PX4_check
    |  |- px4_autopilot_setup.txt
```

Für die Steuerung der Transformationsschritte wurde jeweils projektweise ein Skript erstellt, das entsprechende Methoden des Transformationsprototyps aufruft. Der Name des verwendeten Skripts wird stets in der jeweiligen `setup`-Datei des Projekts vermerkt (zum Beispiel `target_riot_workflow.py`) und der Quellcode dieses Skripts ist im Verzeichnis `wsMA_prototyp/core` zu finden. Die Skripte unterscheiden sich ausschließlich durch die Angaben des verwendeten Projektverzeichnisses, sowie die Pfade zu dem gewünschten Ausgabeverzeichnis der Transformation und den dazugehörigen Excel- und Log-Dateien. Diese Angaben wurden vor jeder Ausführung manuell angepasst, um drei unterschiedlich benannte Log- und Excel-Dateien zu erzielen. Die Skripte werden stets in einer Python-virtuellen Umgebung ausgeführt, die in der jeweiligen projektspezifischen Anleitung `_setup.txt` vermerkt ist.

Wir rufen diese vorbereiteten Skripte direkt im Terminal auf und erfahren, ob beim Einlesen des Konfigurationsraums eine Warnmeldung ausgelöst wird. In der Regel betrifft diese Warnmeldung nicht gesetzte projektspezifische Variablen wie `$(KCONFIG_BUILD)`, die beispielsweise in den `source`-Optionen referenziert werden. Wenn diese Variable erst nach dem erfolgreichen Build-Prozess des Projekts gesetzt wird, kennt der Parser sie nicht und der Transformationsprototyp kann nicht direkt

angewendet werden. In diesem Fall ist es notwendig, die Dokumentation des betrachteten Projekts zu analysieren und das Skript manuell als neues CMake-Target in den Build-Prozess des Projekts einzubinden und es anschließend über das Terminal zu starten.

5.3.2 Ergebnisse

Tabelle 5.2 stellt einen kompakten Überblick über die Ergebnisse des Experiments bereit, bevor diese anschließend projektweise dokumentiert werden. Sie gibt zunächst projektrelevante Informationen zur Projektversion sowie zur Anzahl der Spezifikationsdateien und Konfigurationsoptionen des Projekts. Zudem wird dokumentiert, ob der Prototyp vollständig automatisiert angewendet werden konnte. Darüber hinaus wird vermerkt, ob der Transformationsprototyp als zusätzliches Target in den Build-Prozess integriert oder direkt auf den Quellcode des Projekts angewendet wurde. Abschließend dokumentiert die Tabelle, ob manuelle Anpassungen erforderlich sind, damit das Transformationsergebnis vom `mconf`-Konfigurations-Frontend des Linux-Kconfig-Konfigurationssystems verarbeitet werden kann.[89] Bleibt keine Nacharbeit offen, deutet dies darauf hin, dass alle Transformationsschritte vollständig automatisiert durchgeführt werden konnten.

Tabelle 5.2: FF_1 - Anwendbarkeit der Transformation

Eigenschaft	**Zephyr**	**Zephyr** modifiziert	**ESP-IDF**	**RT-Thread**	**PX4-Autopilot**	**RIOT**
Projektversion	4.2.1	4.2.1	5.5.2	5.2.2	1.16.0	2025.10
Spezifikations-dateien	4466	3882	N/A	118	345	116
(menu)config-Optionen	21780	21769	N/A	1196	834	843
vollständig automatisierte Anwendbarkeit	Teilweise	Teilweise	N/A	Teilweise	Teilweise	Teilsweise
Art der Anwendbarkeit	Als Target	Als Target	Direkt und als Target	Direkt	Direkt	Als Target
Manuelle Nacharbeiten	Ja	Ja	N/A	Ja	Ja (1)	Ja

Zephyr

Im Fall von Zephyr konnte der Transformationsprototyp auf die Version 4.2.1 nur teilweise automatisiert angewendet werden. Zephyr verwendet zahlreiche projektspezifische Variablen, die erst nach einem vollständigen Build aufgelöst werden. Daher war es erforderlich, das für Zephyr vorbereitete Skript (`target_workflow.py`) manuell als neues CMake-Target (`transform`) in den Build-Prozess einzubinden. Da das Build-System von Zephyr anwendungsorientiert ist, wurde die von Zephyr bereitgestellte Beispielapplikation (`samples/hello_world`) als Grundlage verwendet,

[89] https://github.com/torvalds/linux/blob/master/scripts/kconfig/mconf.c

um das Build-System zu initialisieren und das Target zu registrieren. Anschließend wurde der Transformationsprototyp mittels Anweisung `west build -t transform` ausgeführt.

Die Ausführung des Transformationsprototyps war zunächst erfolgreich. Die Analyse der Transformationsausgabe hat aber gezeigt, dass der aktuelle Prototyp bei der Verarbeitung von `named choice`-Optionen an seine Grenzen stößt. Insbesondere treten Probleme auf bei der Verarbeitung von `.defconfig`-Spezifikationsdateien, die der Festlegung von Standardwerten ausgewählter `named choice`-Optionen und Konfigurationsoptionen dienen und in der Regel ausschließlich Attribute, aber keine Elemente innerhalb von `choice`-Optionen enthalten. Aus diesem Grund wurde noch ein weiteres Experiment definiert (vgl. `transform_zrtos/zrtos_demo2_setup.txt`).

Dieses zweite Experiment für Zephyr untersucht ebenfalls die Version 4.2.1. Vor der Transformation wurden aber manuell alle `.defconfig`-Spezifikationsdateien mittels Anweisung `find . -type f -name *.defconfigexec truncate -s 0 +` geleert. Diese Variante des Zephyr-Projekts wird im Folgenden mit dem Attribut **modifiziert** gekennzeichnet. Analog zu dem ersten Experiment wurde das vorbereitete Skript (`target_workflow.py`) manuell als neues CMake-Target (`transform`) eingebunden und die Beispielapplikation (`samples/hello_world`) zur Initialisierung des Build-System und zur Registrierung des Targets verwendet. Die Ausführung dieses Experiments war ebenfalls zunächst erfolgreich. Die Analyse der Transformationsausgabe zeigt aber denselben Fehler wie im ersten Experiment.

ESP-IDF-Framework

Im Fall von ESP-IDF-Framework (v5.5.2) konnte der Transformationsprototyp hingegen nicht erfolgreich angewendet werden. Zur Auflösung der projektspezifischen Variablen wurden zwei Ansätze verfolgt. Erstens wurde der Transformationsprototyp manuell als zusätzliches CMake-Target in den Build des vom ESP-IDF bereitgestellten Beispielprojekts `hello_world` integriert und der Startpunkt der Transformation auf die Haupt-Spezifikationsdatei des ESP-IDF-Frameworks gesetzt (vgl. das zugehörige Skript `target_espidf_workflow.py`). Zweitens wurden die entsprechenden Variablenwerte explizit im aufgerufenen Skript fest codiert. Keiner der beiden Ansätze führte dazu, dass der verwendete ESP-IDF-Parser die mittels `rsource "./components/[...]"` referenzierten Spezifikationsdatei korrekt erkennt. Stattdessen stellte der Parser dem Transformationsprototyp ausschließlich die Haupt-Spezifikationsdatei des ESP-IDF-Frameworks als Konfigurationsraum zur Verfügung. Darüber hinaus stimmte die vom Parser ermittelte Anzahl der `(menu)config`-Optionen nicht mit der in der Haupt-Spezifikation enthaltenen Anzahl überein. Aus diesem Grund wurde ESP-IDF von weiteren Analysen ausgeschlossen.

RT-Thread

Der Transformationsprototyp konnte auf RT-Thread (v5.2.2) zunächst angewendet werden. Hierfür war es nicht erforderlich, den Transformationsprototyp als CMake-Target einzubinden. Stattdessen konnte der Prototyp nach der korrekten Angabe der Dateipfade im Skript `target_rtthread_workflow.py` direkt ausgeführt werden. Als Startpunkt für die Transformation wurde das im RT-Thread enthaltene Beispielprojekt `/bsp/qemu-vexpress-a9` festgelegt. Dieses Projekt verfügt über eine

`Kconfig`-Spezifikation, die unter anderem die Haupt-Spezifikationsdatei im Wurzelverzeichnis von RT-Thread referenziert und die erforderlichen projektspezifische Variablen wie `BSP_DIR` automatisch setzt. Dadurch erhält der Transformationsprototyp Zugriff auf die Haupt-Spezifikationsdatei von RT-Thread.

Die Struktur der Spezifikationsdateien ist in den Beispielprojekten von RT-Thread stark verschachtelt. Daher enthalten diese Dateien auch `source`-Optionen mit Pfadangaben wie `"../../src/Kconfig"`, die auf übergeordnete Verzeichnisse verweisen. Wenn der Transformationsprototyp auf ein solches Beispielprojekt angewendet wird, wird dessen Verzeichnis als Wurzelverzeichnis für die Transformation gesetzt und alle referenzierten Spezifikationsdateien extrahiert. Durch die Transformation des Beispielprojekt und dessen Extraktion aus dem ursprünglichen RT-Thread-Verzeichnis, sind die Pfadangaben wie `"../../src/Kconfig"` gegebenenfalls in einem anderen relativen Verzeichnis zur Wurzel der Transformationsausgabe. Beispielsweise wären jetzt `"../src/Kconfig"` erforderlich. Infolgedessen kann das `mconf`-Frontend solche referenzierten Pfade nicht finden, wodurch weiterführende Analysen ohne eine manuelle Anpassung der Pfadangaben nicht möglich sind.

PX4-Autopilot

Für PX4-Autopilot (1.16.0) konnte der Transformationsprototyp zunächst angewendet werden. Als Startpunkt der Transformation wurde im Rahmen des zugehörigen Skripts (`target_px4_workflow.py`) die Haupt-Spezifikationsdatei des Projekts gesetzt. Diese Spezifikationsdatei enthält die projektspezifischen Variablen `VENDOR`, `MODEL` und `LABEL`, die ausschließlich in `comment`-Optionen verwendet werden. Daher konnte der Transformationsprototyp direkt durch den Aufruf des Skripts ausgeführt werden, ohne das er aufgrund der projektspezifischen Variablen gescheitert ist.

PX4-Autopilot ist das einzige Projekt, das lediglich eine manuelle Anpassung erfordert. Hierbei tritt dasselbe Problem bezüglich `named choice`-Optionen wie bei den Zephyr-Experimenten auf. Da aber nur eine `named choice`-Option existiert, kann nach dem Entfernen ihres Namens die Transformationsausgabe mit `mconf` verarbeitet werden (vgl. transform_projects/transform_px4/PX4_check/).

RIOT

Im Fall von RIOT (2025.10) konnte der Transformationsprototyp zunächst angewendet werden. Aufgrund projektspezifischer Variablen war es aber erforderlich, zunächst einen erfolgreichen Build durchzuführen. Zu diesem Zweck wurde das erstellte Skript `target_riot_workflow.py` manuell als zusätzliches Target in die Datei `kconfig.mk` im `makefiles`-Vezeichnis des Projekts eingebunden und anschließend wurde das Beispielprojekt `/examples/basic/hello-world/` gebaut. Abschließend wurde dieses Target durch den Befehl `make kconfig-transform SHOULD_RUN_KCONFIG=1` gestartet und die Transformation ausgeführt.

Bei RIOT liegt die zentrale Einschränkung darin, dass das `mconf`-Frontend bei der Verarbeitung von Zeilen wie `range 100 $(UINT32_MAX)` die referenzierten Variablen nicht erkennt. Dies deutet darauf hin, dass in diesem Fall das `mconf`-Frontend allein nicht mit den referenzierten Variablen umgehen kann, sondern ein vorheriger Aufruf von Skripten wie `preprocess.c` zur Variablenauflösung erforderlich ist.[90]

[90] https://github.com/torvalds/linux/blob/master/scripts/kconfig/preprocess.c

5.3.3 Diskussion

Die Ergebnisse des Experiments zeigen, dass die Anwendbarkeit des Transformationsprototyps maßgeblich durch die Einbettung von den Kconfiglib-Spezifikationen eines Projekts in das jeweilige Build-System bestimmt wird. Insbesondere der Zeitpunkt und die Art der Auflösung projektspezifischer Variablen erweisen sich als entscheidende Faktoren für eine erfolgreiche Anwendung des Transformationsprototyps. Bei fünf von sechs durchgeführten Experimenten (vgl. Tabelle 5.2) konnte der Transformationsprototyp angewendet werden, obwohl die zugrunde liegenden Projekte in unterschiedlichem Ausmaß projektspezifische Variablen verwenden. Gemeinsam haben diese Projekten, dass die für die Transformationseingabe benötigten Variablenwerte entweder bereits statisch definiert sind oder sich im Rahmen eines kontrollierten Build-Prozesses deterministisch ableiten lassen.

Demgegenüber zeigt der Fall von ESP-IDF eine Grenze des Ansatzes. Die Diskrepanz hinsichtlich der Anzahl von durch den Parser ermittelten und tatsächlich vorhandenen `(menu)config`-Optionen deutet darauf hin, dass die vollständige Verarbeitung von Spezifikationsdateien des ESP-IDF-Frameworks erst durch weiterführende, projektspezifische Mechanismen erfolgt. Diese werden vom Transformationsprototyp nicht abgedeckt, da der Prototyp nur den Parser der entsprechenden Kconfiglib-Variante als Hauptwerkzeug zum Einlesen der Spezifikationsdateien aufruft.

Im Rahmen der Evaluierung der Anwendbarkeit wurden Ausnahmefälle identifiziert, die dazu führen, dass der Transformationsprototyp auf die betrachteten Projekte nur eingeschränkt automatisiert anwendbar ist und manuelle Nacharbeiten erforderlich bleiben. Besonders relevant sind hierbei `named choice`-Optionen, die nur Attribute (wie `depends on` oder `default`) und keine Auswahlelemente (`(menu)config`-Optionen) haben. Solche Optionen stellen eine strukturelle Limitation des Prototyps dar, die eine Anpassung der Transformationslogik erforderlich machen. Der Fall des Projekts PX4-Autopilot zeigt, dass eine vollständige Verarbeitung nach entsprechender Anpassung möglich ist, was die prinzipielle Überwindbarkeit dieser Limitation unterstreicht (s. Diskussion 5.6.3).

Ein weiterer projektspezifischer Ausnahmefall ist die stark verschachtelte Struktur der Spezifikationsdateien im RT-Thread-Projekt, die zudem `source`-Optionen mit relativen Pfadangaben (wie `"../../Kconfig"`) verwendet. Auch dieser Fall erscheint prinzipiell überwindbar, erfordert aber eine Erweiterung der Transformationslogik zur automatisierten Anpassung der Pfadangaben (s. Diskussion 5.6.3).

Trotz dieser Ausnahmefälle konnten im Rahmen dieser Analyse die Mehrheit der Kconfiglib-Erweiterungen erfolgreich transformiert und zugleich die Auskünfte über deren Verwendung gewonnen werden (vgl. FF_2). Der Transformationsprototyp überführt `source`-Optionen und deren Alternativen, löst Glob-Muster auf, transformiert `def_*`-Attribute sowie `configdefault`- und `choice`-Optionen und verarbeitet die alternativen Schreibweise von Attributen (z.B. `---help---`), die der Rückwärtskompatibilität dienen.

5.4 FF_2 Nutzung der Kconfiglib-Erweiterungen

Aufbauend auf dem vorherigen Kapitel wird im Rahmen dieser Forschungsfrage die Verbreitung von Kconfiglib-Erweiterungen untersucht. Betrachtet werden die fünf Projekte, auf die der Transformationsprototyp angewendet werden konnte (vgl. Tabelle 5.2).

Forschungsfrage 2

Wie häufig verwenden bestehende Projekte die Kconfiglib-Erweiterungen?

5.4.1 Methodik

Diese Forschungsfrage untersucht anhand der vom Transformationsprototyp automatisch erhobenen statistischen Daten die Verbreitung von Kconfiglib-Erweiterungen. Die Häufigkeiten werden während der Transformation über interne Zählervariablen erfasst und in Excel- und Log-Datei abgelegt.

Die Log-Datei protokolliert Häufigkeitsinformationen für jede verarbeitete, von dem Parser bereitgestellte Spezifikationsdatei, sowie zusammenfassende projektweite Kennzahlen bezüglich Kconfiglib-Erweiterungen. Ausgewählte Statistiken dieser Datei, wie die Anzahl von `source`-Optionen oder `def_*`–Attribute, werden zusätzlich in der Excel-Datei abgelegt, um eine einfache Aggregation und Auswertung über alle Spezifikationsdateien hinweg zu ermöglichen. Im Abschlussbereich der Log-Datei befinden sich zudem Informationen über die Gesamtanzahl von `(menu)config`-Optionen, `configdefault`-Optionen, `choice`-Optionen und `named choice`-Optionen, da diese lediglich direkt aus den Parser-Informationen ausgelesen werden.

Im Rahmen der FF_1 wurden die fünf Projekte schon eingerichtet und mit dem Transformationsprototyp verarbeitet. Aufbauen auf diesen Verarbeitungsergebnissen sind die für dieses Kapitel relevanten Excel- und Log-Dateien projektweise in dem entsprechenden `_log`-Unterverzeichnis abgelegt (vgl. wsMA_prototyp/transform_projects). Für die Auswertung werden die in der Excel-Datei verfügbaren Daten aggregiert und anschließend mit den Informationen aus den Log-Dateien ergänzt.

5.4.2 Ergebnisse

Im Folgenden werden die Ergebnisse projektweise dargestellt und anschließend tabellarisch zusammengefasst. Die jeweilige Tabelle zeigt, wie häufig die jeweilige Erweiterung insgesamt verwendet und in wie vielen Spezifikationsdateien sie definiert wird. Die letzte Spalte der Tabelle zeigt, welchen Anteil diese Spezifikationsdateien im Verhältnis zu der Gesamtanzahl der Spezifikationsdateien des jeweiligen Projekts einnehmen.

Die Analyse des Zephyr-Projekts (v.2.4.1) umfasst einen Konfigurationsraum von insgesamt 4466 Spezifikationsdateien. Darin sind 21780 `(menu)config`-Optionen enthalten, von denen zwölf als `configdefault`-Optionen definiert sind. Zudem sind 939 `choice`-Optionen spezifiziert, von denen 755 (80,40%) als `named choice`-Optionen definiert sind. Am häufigsten werden die Erweiterungen `rsource` (782 Vorkommen) und `osource` (370 Vorkommen) eingesetzt, während `orsource`-Optionen lediglich achtmal verwendet werden. Das `optional`-Attribut der `(named)choice`-Optionen tritt nur dreimal auf. Die `source`-Optionen werden insgesamt 2300 mal verwendet, wobei nur zwei Optionen (0,09%) Glob-Muster aufweisen.

Die Analyse des modifizierten Zephyr-Projekts umfasst 3882 Spezifikationsdateien. Darin sind 21769 `(menu)config`-Optionen und eine `configdefault`-Option definiert. Die Anzahl der `choice` und `named choice`-Optionen, sowie die Anzahl der `source`- und `osource`-Optionen stimmt mit denen vom nicht modifizierten Zephyr-Projekt überein. Dahingegen hat das modifizierte Zephyr 626 `rsource`-Optionen und keine `orsource`-Optionen.

Die Analyse des Projekts RT-Thread (v.5.2.2) umfasst 118 Spezifikationen. Darin sind 1196 `(menu)config`-Optionen sowie 21 `choice`-Optionen erhalten, von denen 15 als `named choice`-Optionen (70,43%) definiert sind. Als weitere Erweiterungen werden am häufigsten `rsource`-Optionen verwendet (114), gefolgt von `osource`-Optionen (45). Von den zwei `source`-Optionen verwendet keine Glob-Muster. Darüber hinaus tritt eine `orsource`-Option auf.

Die Analyse des Projekts PX4-Autopilot (v.1.16.0) umfasst 345 Spezifikationen mit insgesamt 834 `(menu)config`-Optionen. Zudem sind sieben `choice`-Optionen definiert, von denen ausschließlich eine als `named choice`-Option vorliegt. Von den sechs `source`-Optionen verwendet eine Glob-Muster. Von allen Alternativen der `source`-Option wird lediglich die `rsource`-Option 50 mal verwendet. Weitere Kconfiglib-Erweiterungen werden nicht verwendet. PX4-Autopilot ist das einzige Projekt, das die Schreibweise `---help---` des `help`-Attribut verwendet (419 Vorkommen).

Die Analyse des RIOT-Projekts (v.2025.10) umfasst 116 Spezifikationen, die insgesamt 843 `(menu)config`-Optionen definieren. Zudem sind 45 `choice`-Optionen enthalten, von denen nur eine als `named choice`-Option vorliegt. Am häufigsten im Konfigurationsraum treten mit 160 Vorkommen `rsource`-Optionen auf, während deren optionale Alternative (`orsource`) lediglich zweimal in einer Spezifikation definiert ist. Darüber hinaus werden keine `source`-Optionen mit oder ohne Glob-Muster verwendet. Stattdessen treten ausschließlich zehn `osource`-Optionen auf.

Tabelle 5.3: FF_2 - Kconfiglib-Erweiterungen in Zephyr

Erweiterung	Gesamtanzahl der Vorkommen	Anzahl betroffener Spezifikations-dateien	Anteil betroffener Spezifikations-dateien [%]
`def_int,def_hex,def_string`	37	17	0,38
`source` mit Glob-Muster	2	2	0,04
`osource`	370	15	0,34
`rsource`	782	402	9,00
`orsource`	8	8	0,18
`option env`	0	0	0
`option modules`	0	0	0
`option allnoconfig_y`	0	0	0
`option defconfig_list`	0	0	0
`optional`	3	3	0,07

Tabelle 5.4: FF_2 - Kconfiglib-Erweiterungen im modifizierten Zephyr

Erweiterung	Gesamtanzahl der Vorkommen	Anzahl betroffener Spezifikations-dateien	Anteil betroffener Spezifikations-dateien[%]
`def_int,def_hex,def_string`	34	14	0,36
`source` mit Glob-Muster	2	2	0,05
`osource`	370	15	0,39
`rsource`	626	246	6,34
`orsource`	0	0	0
`option env`	0	0	0
`option modules`	0	0	0
`option allnoconfig_y`	0	0	0
`option defconfig_list`	0	0	0
`optional`	3	3	0,08

Tabelle 5.6: FF_2 - Kconfiglib-Erweiterungen in PX4-Autopilot

Erweiterung	Gesamtanzahl der Vorkommen	Anzahl betroffener Spezifikations-dateien	Anteil betroffener Spezifikations-dateien[%]
`def_int,def_hex,def_string`	0	0	0
`source` mit Glob-Muster	1	1	0,29
`osource`	0	0	0
`rsource`	50	49	14,20
`orsource`	0	0	0
`option env`	0	0	0
`option modules`	0	0	0
`option allnoconfig_y`	0	0	0
`option defconfig_list`	0	0	0
`optional`	0	0	0

Tabelle 5.5: FF_2 - Kconfiglib-Erweiterungen in RT-Thread

Erweiterung	Gesamtanzahl der Vorkommen	Anzahl betroffener Spezifikations-dateien	Anteil betroffener Spezifikations-dateien[%]
`def_int,def_hex,def_string`	0	0	0
`source` mit Glob-Muster	0	0	0
`osource`	45	42	35,59
`rsource`	114	24	20,34
`orsource`	1	1	0,85
`option env`	0	0	0
`option modules`	0	0	0
`option allnoconfig_y`	0	0	0
`option defconfig_list`	0	0	0
`optional`	0	0	0

Tabelle 5.7: FF_2 - Kconfiglib-Erweiterungen in RIOT

Erweiterung	Gesamtanzahl der Vorkommen	Anzahl betroffener Spezifikations-dateien	Anteil betroffener Spezifikations-dateien[%]
`def_int,def_hex,def_string`	0	0	0
`source` mit Glob-Muster	0	0	0
`osource`	10	7	4,22
`rsource`	160	33	19,88
`orsource`	2	1	0,60
`option env`	0	0	0
`option modules`	0	0	0
`option allnoconfig_y`	0	0	0
`option defconfig_list`	0	0	0
`optional`	0	0	0

5.4.3 Diskussion

Die Analyse zeigt, dass Zephyr unter den betrachteten Projekten den höchsten Anteil an Kconfiglib-Erweiterungen aufweist. Dies ist plausibel, da Zephyr die Weiterentwicklung von ursprünglichen Kconfiglib-Bibliothek übernommen hat und diese aktiv vorantreibt. In früheren Veröffentlichungen wurde Zephyr als „kleiner Bruder“ des Linux-Kernels bezeichnet.[55] Im Rahmen dieser Analyse ermittelte der Transformationsprototyp mittels des Parsers der ZRTOS-Kconfiglib, dass der Konfigurationsraum von Zephyr mit über 4000 Spezifikationsdateien insgesamt mehr als 21000 `(menu)config`-Optionen enthält. Diese Zahlen stützen die ursprüngliche Einordnung nur noch sehr eingeschränkt und deuten darauf hin, dass Zephyr inzwischen einen Konfigurationsraum mit hoher struktureller Komplexität aufweist. Entsprechend ist es plausibel, dass in Zephyr die `named choice`-Optionen die Mehrheit aller `choice`-Optionen darstellen und dass `source`-Optionen (sowie ihre Alternativen) die dominierenden Kconfiglib-Erweiterungen bilden. Diese Erweiterungen können als Strukturierungsmechanismen interpretiert werden, die bei der Bewältigung der hohen Komplexität des Konfigurationsraums unterstützen.

Projektübergreifend zeigt sich, dass `rsource`-Optionen in drei der fünf Projekte insbesondere im Vergleich zu anderen `source`-Alternativen dominant genutzt werden. Wie die Anteile in Abbildung 5.1 verdeutlichen, werden `rsource`-Optionen in RT-Thread, PX4-Autopilot und RIOT bevorzugt zur Strukturierung des Konfigurationsraums eingesetzt. Es ist zu beachten, dass die Anteile unter einem Prozent in der Abbildung nicht dargestellt werden. So werden beispielsweise `orsource`-Attribute mit 0,23% nicht im Zephyr-Diagramm angezeigt.

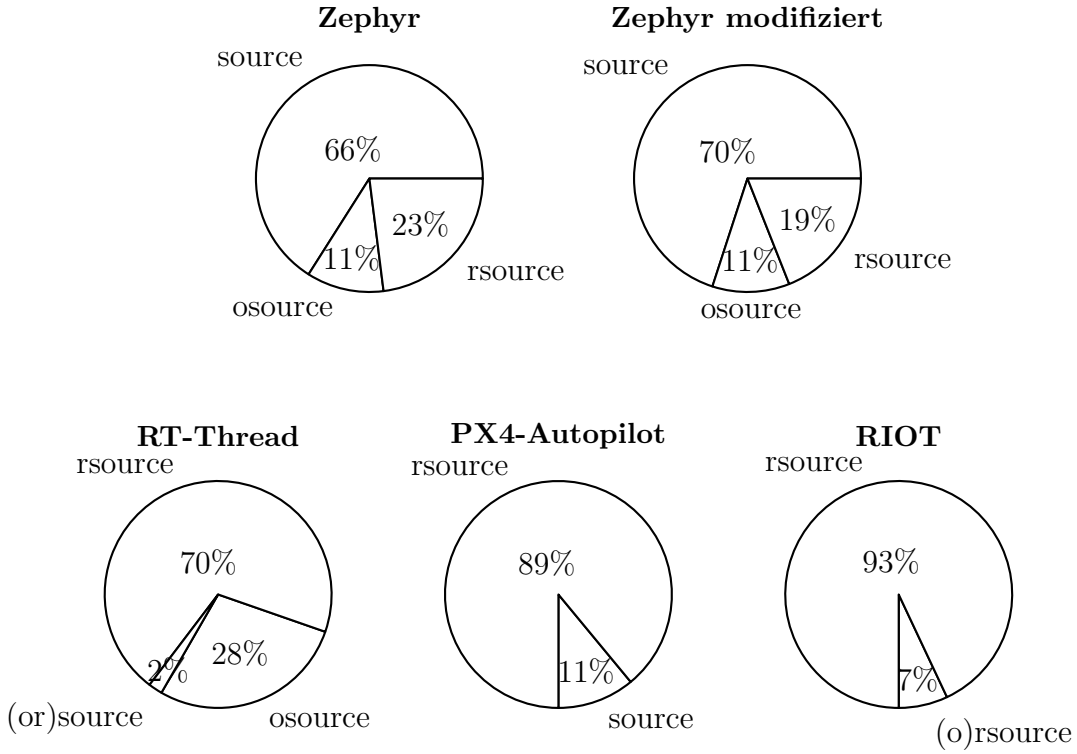

Abbildung 5.1: Anteil verschiedener `source`-Optionen in Projekten

5.5 FF_3 Effizienz

Aufbauend auf der vorherigen Forschungsfrage untersucht dieses Kapitel, wie sich die Transformation auf die Größe der resultierenden Spezifikationen auswirkt und welcher Zeitaufwand für die Transformation erforderlich ist.

Forschungsfrage 3

Wie wirkt sich die Transformation auf die Größe der resultierenden Spezifikationen aus? Wie lange dauert die Transformation?

5.5.1 Methodik

Die während der Transformation erzeugten Excel- und Log-Dateien erfassen nicht nur die Anzahl von verwendeten Kconfiglib-Erweiterungen, sondern messen auch die Zeilenanzahl der Ein- und Ausgabespezifikationen des betrachteten Projekts (vgl. wsMA_prototyp/transform_projects). Dadurch ist es möglich, die Größenänderung einzelner Spezifikationsdateien zu bestimmen und diese anschließend zu aggregieren, um einen projektübergreifenden Überblick zu erhalten. Darüber hinaus liefern die internen Zählervariablen des Prototyps Informationen darüber, wie viele Zeilen durch welche Transformationsregeln beeinflusst wurden. Konkret speichert der Prototyp die Anzahl der Änderungen, die die Größe der resultierenden Spezifikationen direkt beeinflussen. Dies ist beispielsweise der Fall, wenn Zeilen aufgrund fehlender Treffer für `osource`- oder `rsource`-Optionen entfernt werden oder wenn neue Zeilen infolge von `def_*`-Attributen zur Ausgabespezifikation hinzugefügt werden.

5.5.2 Ergebnisse

Im Folgenden werden die Ergebnisse projektweise dargestellt und abschließend in Tabelle 5.8 zusammengefasst.

Der Transformationsprototyp hat im Fall von Zephyr insgesamt 4466 Spezifikationsdateien mit 202078 Zeilen verarbeitet. Das Transformationsergebnis umfasst 201294 Zeilen und ist damit um 784 Zeilen kleiner. Von den 4466 Spezifikationsdateien bleiben 4162 (93%) nach der Transformation hinsichtlich der Zeilenanzahl unverändert. Die übrigen 304 (7%) Spezifikationsdateien veränderten sich, wobei 198 (4%) größer und 106 (2%) kleiner wurden. Die mittlere Transformationszeit beträgt 2621,14 Sekunden (ca. 44 Minuten).

Der Transformationsprototyp hat beim modifizierten Zephyr-Projekt 3882 Spezifikationsdateien mit insgesamt 187689 Zeilen verarbeitet. Das Transformationsergebnis ist um 526 Zeilen kleiner. Von allen verarbeiteten Spezifikationsdateien bleiben 96% nach der Transformation hinsichtlich der Zeilenanzahl unverändert. Die übrigen 153 (4%) Spezifikationsdateien veränderten sich, wobei 111 (3%) größer und 111 (1%) kleiner wurden. Die mittlere Transformationszeit beträgt 1942,43 Sekunden (ca. 32 Minuten).

Für RT-Thread hat der Transformationsprototyp 118 Spezifikationsdateien mit insgesamt 6556 Zeilen verarbeitet. Nach der Verarbeitung verringerte sich die Zeilenanzahl um 48 Zeilen, ausschließlich infolge der Entfernung von `(o)rsource`-Optionen ohne

Treffer. Von den 118 Spezifikationsdateien wurden weniger als die Hälfte (44) durch diese Reduzierung beeinflusst. Der Transformationsprototyp hat 5,81 Sekunden in Anspruch genommen.

Bei PX4-Autopilot wurden insgesamt 345 Spezifikationsdateien verarbeitet. Die Gesamtanzahl der Zeilen erhöhte sich nach der Transformation von 4829 auf 5113, was einem Zuwachs von 5,9% (284 Zeilen) entspricht. Insgesamt wurden 36 (0,7%) Spezifikationsdateien verändert, von denen lediglich eine aufgrund einer `(o)rsource`-Option ohne Treffer verkleinert wurde. Die mittlere Transformationszeit beträgt 10,1 Sekunden.

Der Transformationsprototyp hat für RIOT 166 Spezifikationsdateien mit insgesamt 7299 Zeilen verarbeitet. Nach der Transformation verringerte sich die Zeilenanzahl um 26 Zeilen. Dabei wurden 16 Spezifikationsdateien beeinflusst, ausschließlich durch die Entfernung von `o(r)source`-Optionen ohne Treffer. Der Transformationsprototyp hat 8,97 Sekunden in Anspruch genommen.

Tabelle 5.8 zeigt für jedes Projekt, wie viele Zeilen über alle verarbeiteten Spezifikationsdateien hinweg durch die Transformation hinzugefügt oder entfernt wurden. Für beide Änderungsarten werden die jeweiligen Ursachen aufgeführt (siehe erste Spalte). Die letzten Zeilen der Tabelle fassen die Gesamtanzahl der Zeilen vor und nach der Anwendung des Transformationsprototyps, die daraus resultierende Differenz sowie die durchschnittliche Transformationszeit zusammen. Die angegebene Transformationszeit entspricht dem Medianwert aus drei Anwendungen des Transformationsprototyps.

Tabelle 5.8: FF_3 - Zeilenanzahl nach Transformation

	Zephyr	**Zephyr modifiziert**	**RT-Thread**	**PX4-Autopilot**	**RIOT**
Erweitert aufgrund					
`def_*`	37	34	0	0	0
Auflösung von Glob-Muster	971	539	0	285	0
`configdefault`	44	1	0	0	0
`named choice`	151	118	0	0	0
Entfernt aufgrund					
Kein Treffer für `o(r)source`	105	153	48	1	26
`configdefault`	66	2	0	0	0
`named choice`	1836	1062	0	0	0
`optional`	3	3	0	0	0
Eingabe	**202078**	**187689**	**6556**	**4829**	**7299**
Ausgabe	**201294**	**187163**	**6508**	**5113**	**7273**
Differenz	**-784**	**-526**	**-48**	**+284**	**-26**
Laufzeit (Median)[s]	**2621,14s**	**1942,43s**	**5,81s**	**10,1s**	**8,97s**

5.5.3 Diskussion

Wie bereits in Diskussion 5.3.3 erläutert, führte insbesondere der Ausnahmefall `named choice`-Optionen dazu, dass die Transformationsausgabe nicht vollständig automatisiert erzeugt werden konnte. Infolgedessen entsteht in Tabelle 5.8 bei den Angaben der Zeilenänderungen aufgrund von `named choice`-Optionen eine Diskrepanz zwischen den gezählten und den tatsächlichen Zeilen der Ausgabespezifikation. Dies liegt daran, dass die internen Zählervariablen eng mit der Transformation der jeweiligen Kconfiglib-Erweiterung verknüpft sind. Wenn die Transformation nicht vollständig erfolgreich ist, wird die Korrektheit der Zähler negativ beeinflusst.

Die Größenänderung der Transformationsausgabe hängt stark von den im jeweiligen Projekt dominierenden Kconfiglib-Erweiterungen ab. Bei Projekten wie PX4-Autopilot ist eine Größenzunahme plausibel, da die Auflösung von Glob-Mustern die dominierende Transformationsregel ist und ausschließlich eine optionale `source`-Option entfernt wird. Kconfiglib-Erweiterungen existieren, erfahren solche Wenn stattdessen `named choice`-Optionen in den Spezifikationsdateien des Projekts dominieren oder das Projekt viele optionale Spezifikationsdateien ohne Treffer referenziert, entsteht tendenziell eine eine Reduktion der Zeilenanzahl.

Der Laufzeitanalyse zufolge beanspruchen die beiden Zephyr-Varianten den höchsten Zeitbedarf. Einerseits ist dies auf den erheblich größeren Umfang der verarbeiteten Zeilen, nämlich über 200000 Zeilen gegenüber weniger als 8000 bei den anderen Projekten, zurückzuführen. Andererseits wird angenommen, dass sich die hohe Anzahl an Glob-Muster-Auflösungen in Zephyr negativ auf die Laufzeit auswirkt. Dies ist darauf zurückzuführen, dass der Prototyp Glob-Muster durch den Vergleich extrahierter Parser-Informationen auf Menüknotenebene auflöst, was bei einem großen Projekt mit vielen Menüknoten zu einer großen Anzahl von Vergleichen führt. Dieser Ansatz wurde ursprünglich aufgrund der zentralen Rolle der Extraktion von Parser-Informationen im Transformationsprototyp bevorzugt. Der alternative Ansatz der Verwendung der `iglob`-Funktion der `glob`-Bibliothek zur Auflösung von Glob-Mustern, wurde im Prototyp als eine optionale Prüffunktion des ersten Ansatzes implementiert. Es wird angenommen, dass der Einsatz der `glob`-Funktion als primärer Mechanismus die Gesamtlaufzeit reduzieren könnte, da dadurch Vergleiche auf Menübaumebene entfallen.

5.6 FF_4 Genauigkeit

In diesem Kapitel wird die Plausibilität der Korrektheit der Transformationsregeln überprüft.

Forschungsfrage 4

Werden Features und deren Abhängigkeiten durch die Transformation korrekt behandelt?

5.6.1 Methodik

Die in Kapitel 3.4 hergeleiteten Transformationsregeln wurden systematisch basierend auf dem Vergleich der Funktionsweise von Kconfiglib und der Linux-Kconfig-Sprache konzipiert. Das erzeugte Konzept begründet den angestrebten Transformationsprozess somit zunächst theoretisch und hebt die semantischen Modifikationen hervor, die unmittelbare Folge der Transformation sind. Beispielsweise bewirkt die Entfernung des `optional`-Attributs, dass die betreffende `(named)choice`-Option verpflichtend ausgewählt werden muss. Die erstellten Transformationsregeln wurden im Rahmen des in Kapitel 4 entwickelten Transformationsprototyps umgesetzt.

Ein formaler Beweis oder eine umfassende empirische Analyse der Korrektheit liegt außerhalb des Rahmens dieser Arbeit. Daher wurde das stichprobenbasierte Testen herangezogen. Hierfür wurde jede zu transformierende Kconfiglib-Erweiterung im Rahmen eines Minimal Working Example (MWE) gekapselt. Die verwendeten MWEs sind im Verzeichnis `test_dir_` des GitHub-Repository nach dem Namen der Kconfiglib-Erweiterung gruppiert. Ergänzend dazu bietet Tabelle 5.9 einen Überblick darüber, welche Transformationsregeln in welchem Unterverzeichnis getestet werden.

Tabelle 5.9: FF_4 - `MWE` und zugehörige Transformationsregeln

Unterverzeichnis des `test_dir_`-Verzeichnis	**abgedeckte Transformationsregeln**
transform_source	3.8
transform_def	3.9
transform_option	3.10 und 3.11
transform_choice	3.12 und 3.13
transform_configdefault	3.14

Zur Sicherstellung der syntaktischen Korrektheit von erstellen MWEs wird evaluiert, ob diese von ZRTOS-Kconfiglib verarbeitet werden können. Das bedeutet, es ist entscheidend, ob das zugehörige `menuconfig`-Konfigurations-Frontend dieser Kconfiglib-Variante das betrachtete MWE einlesen kann oder seine internen, automatisch ausgeführten Syntax-Prüfungen aufgrund von Fehlern im MWE scheitern.[91] Das Frontend von ZRTOS-Kconfiglib wird bevorzugt, da die MWEs nach den Syntax-Regeln von ZRTOS-Kconfiglib erstellt wurden. Dies liegt daran, dass

[91] https://github.com/zephyrproject-rtos/zephyr/blob/main/scripts/kconfig/menuconfig.py

ZRTOS-Kconfiglib den Sprachumfang von Zephyr-Kconfiglib um die projektspezifische Erweiterung `configdefault` ergänzt, deren Transformationsregel wiederum Bestandteil der Analyse ist.

Der Transformationsprototyp wird über das Skript `workflow_test.py` auf einzelne MWEs angewendet, wobei die im Skript enthaltenen Pfadangaben vor jeder Ausführung an das Unterverzeichnis des jeweiligen MWE angepasst werden. Die Transformationsergebnisse sind jeweils in dem entsprechenden Unterverzeichnis des MWE (vgl. Tabelle 5.9) durch das Suffix `_output` zu erkennen. Darin enthalten sind die transformierten Spezifikationen des MWE sowie eine Log-Datei (`transform.log`), die einen ausführlichen Überblick über die während der Transformation durchgeführten Transformationsschritte gibt.

Die syntaktische Plausibilität der Transformationsergebnisse wird evaluiert, indem die transformierten Spezifikationen mit dem `mconf`-Frontend aus dem Linux-Kernel eingelesen werden.[89] Wenn das `mconf`-Frontend einen Fehler beim Einlesen anzeigt, deutet das auf die Stellen der Transformation hin, die manuell angepasst werden müssen. Kann die transformierte Spezifikation von beiden Werkzeugen fehlerfrei eingelesen werden, spricht dies für eine syntaktisch korrekte Transformation, da beide Frontends intern eine umfangreiche Prüfung der eingelesenen Spezifikationen vornehmen.

Sowohl das `menuconfig`-Frontend als auch das `mconf`-Frontend von Linux werden direkt aus dem Terminal aufgerufen und verlangen somit die korrekte Setzung von Parametern. Dazu gehören `SRCTREE` und gegebenenfalls Umgebungsvariablen, die erforderlich sind um die Spezifikationen des MWE einlesen zu können. Die technische Dokumentation hierfür ist in `test_dir/set_up_notes.txt` zu finden.

5.6.2 Ergebnisse

Tabelle 5.10 fasst die Ergebnisse des durchgeführten Experiments für jede Kconfiglib-Erweiterung beziehungsweise ihre zugehörige Transformationsregel zusammen.

Tabelle 5.10: FF_4 - Plausibilitätsprüfung (`MWE`)

	vollständig automatisierte Transformation	**Syntax-Prüfung durch `menuconfig`**	**Syntax-Prüfung durch `mconf`**	**semantische Änderungen des MWE**
source	3.8 ✓	✓	✓	keine
osource	3.8 ✓	✓	✓	keine
rsource	3.8 ✓	✓	✓	keine
orsource	3.8 ✓	✓	✓	keine
def_int	3.9 ✓	✓	✓	keine
def_string	3.9 ✓	✓	✓	keine
def_hex	3.9 ✓	✓	✓	keine
option env	3.10 ✓	✓	✓	✓
option modules	3.11 ✓	✓	✓	keine
named choice	3.12 ✓	✓	✓	✓
choice	3.13 ✓	✓	✓	✓
configdefault	3.14 ✓	✓	✓	keine

Die zweite Spalte der Tabelle gibt einen Überblick darüber, ob alle Transformationsschritte einer Transformationsregel vollständig durch den Transformationsprototyp verarbeitet werden können oder manuelle Anpassungen des Transformationsergebnisses erforderlich sind. Die dritte und vierte Spalte weisen darauf hin, ob die jeweilige Syntax-Prüfung erfolgreich war. Abschließend notiert die letzte Spalte, ob jeweilige Transformationsregel dafür sorgt, dass das Transformationsergebnis semantisch modifiziert ist, weil beispielsweise bestimmte Attribute, Elemente oder Konfigurationsoptionen aufgrund der Transformationsregel ausgeschlossen oder zusätzlich hinzugefügt werden.

5.6.3 Diskussion

Im Rahmen eines stichprobenbasierten Testens mittels MWE wurden alle Transformationsregeln von beiden Werkzeugen erfolgreich verarbeitet, was zunächst auf eine syntaktisch korrekte Transformation hindeutet. Trotzdem wurden, wie bereits in FF_1 diskutiert, bei der Anwendung des Transformationsprototyps auf reale Projekte Ausnahmefälle identifiziert. Diese betreffen insbesondere die Transformation von `named choice`-Optionen sowie von `source`-Optionen mit relativen Pfadangaben.

Der erste Ausnahmefall betrifft die Transformation von `named choice`-Optionen, die lediglich Attribute (wie `default` oder `help`) enthalten und keine `(menu)config`-Optionen als Auswahlelemente definieren. Dieser Fall erfordert eine detaillierte Ursachenanalyse, die aufgrund des begrenzten zeitlichen Rahmens dieser Arbeit nicht durchgeführt werden konnte. Es wird angenommen, dass eine Anpassung der internen Logik des Prototyps im Schritt der Zusammenführung mehrerer `named choice`-Definitionen notwendig ist, insbesondere im Hinblick auf die Reihenfolge der Verarbeitung von Attributen aus unterschiedlichen Definitionen einer `named choice`-Option.

Der zweite Ausnahmefall tritt auf, wenn der Transformationsprototyp auf ein Beispielprojekt innerhalb einer stark verschachtelten Struktur von Spezifikationsdateien angewendet wird. In diesem Fall ist ein zusätzlicher Prüf- und Überführungsschritt erforderlich, um relative Pfadangaben korrekt an das neue Wurzelverzeichnis der Transformationsausgabe anzupassen, bevor diese in die transformierten Spezifikationsdatei geschrieben wird. Die Anpassung dieser projektspezifischen Fall konnte aufgrund des begrenzten zeitlichen Umfangs nicht durchgeführt werden.

5.7 Angriffspunkte der Evaluierung

Als Grundlage für die Evaluierung der Forschungsfragen dient der im Rahmen dieser Arbeit entwickelte Transformationsprototyp. Dieser implementiert sämtliche im Kapitel 3.4 konzipierten Transformationsregeln und zeigt gemäß den Ergebnissen der Forschungsfrage 4 eine grundsätzlich syntaktisch korrekte Umsetzung der intendierten Konzepte, sofern diese auf MWE angewendet werden.

Die erstellten Transformationsregeln und die dazugehörigen MWE umfassen aus theoretischer Sicht ein breites Spektrum möglicher Sprachelemente und deren Attributen. Die Analyse im Rahmen der Forschungsfrage 1 hat aber gezeigt, dass bei der Verarbeitung realer Projekte projektspezifische Ausnahmefälle auftreten, bei denen die aktuelle Version des Prototyps an ihre implementierungsspezifischen Grenzen stößt.

Die Evaluierung der Anwendbarkeit des erstellten Transformationsprototyps (FF_1) beschränkt sich auf fünf von insgesamt elf in Kapitel 3.2 identifizierten Projekten. Die Auswahl dieser Projekte erfolgte gezielt anhand ihrer Relevanz für die Weiterentwicklung der ursprünglichen Kconfiglib-Bibliothek sowie unter Berücksichtigung ihrer Verbreitung und Nutzung, wie sich aus den zugehörigen GitHub-Statistiken ableiten lässt. Dadurch werden führende Projekte in die Evaluierung einbezogen.

Die im Rahmen der Evaluierung erhobenen Statistiken zur Nutzung von Kconfiglib-Erweiterungen (FF_2) werden teilweise direkt aus Parser-Informationen und teilweise aus eigenen, im Prototyp implementierten Zählervariablen ermittelt (vgl. 5.4.1). Die korrekte Bestimmung der Gesamtanzahl wird dadurch unterstützt, dass jede Zeile der Spezifikationsdateien einzeln verarbeitet wird. Potenziell fehleranfällig ist hingegen die Ermittlung der Zeilenanzahl einzelner Änderungen aufgrund spezifischer Kconfiglib-Erweiterungen, da diese eine direkte Korrelation zur korrekten Transformation der Spezifikationsdateien voraussetzt. Konkret bedeutet dies, dass bei einem nicht vollständig korrekt transformierten projektspezifischen Fall (s. Zepyhr in FF_1) eine Diskrepanz zwischen der gemessenen Größe der Transformationsausgabe und ermittelten Anzahl der Änderungen entstehen kann.

Die im Rahmen dieses Experiments ermittelte Transformationszeit wurde auf einen Laptop mit begrenzten Hardware-Ressourcen gemessen. Abhängig von der Leistungsfähigkeit der eingesetzten Hardware können die Transformationszeiten kürzer oder länger ausfallen. Darüber hinaus werden die gemessenen Transformationszeiten durch den Detailgrad der aktivierten Logging-Informationen beeinflusst. Eine detaillierte Ausgabe von Parser-Informationen erhöht den Laufzeitaufwand, insbesondere bei größeren Projekten wie Zephyr, die mehr als 20000 Konfigurationsoptionen umfassen. Entsprechend sind die Transformationszeiten nicht als absolute Referenzwerte, sondern als relative Vergleichswerte unter den gegebenen experimentellen Bedingungen zu interpretieren, was bei der Interpretation von Ergebnissen zu berücksichtigen ist.

Obwohl das stichprobenbasierte Testen in FF_4 eine grundlegende syntaktische Korrektheit der Transformationsregeln gezeigt hat, erlaubt es keine allgemeingültige Aussage über die Korrektheit der Transformationen. Dies liegt insbesondere daran, dass stichprobenbasierte Tests keine vollständige Abdeckung aller Elemente der Kconfig-Syntax und deren Kombinationen gewährleisten, was durch einige identifizierte Ausnahmefälle bestätigt wurde. Die möglichen Ursachen und Problemstellen

in der Prototyplogik wurden erläutert, um eine Grundlage für zukünftige Verbesserungen zu schaffen. Trotzdem bedarf es einer umfassenden empirischen Analyse oder eines formalen Beweises der Transformationsregeln, um ihre Korrektheit abschließend sicherzustellen.

5.8 Zusammenfassung

In diesem Kapitel wurde die Evaluation des Transformationsprototyps vorgestellt. Die Ergebnisse zeigten, dass der Transformationsprototyp grundsätzlich für eine breite Klasse von Kconfiglib-basierten Open-Source-Projekten geeignet ist, aber die Anwendung stark von der Auflösung der projektspezifischen Variablen abhängig ist. Die Evaluation hat gezeigt, dass Kconfiglib-Erweiterungen projektabhängig unterschiedlich eingesetzt werden. Dominierend sind dabei Kconfiglib-Erweiterungen, welche der strukturellen Einbindung von Spezifikationsdateien (`source`- und `rsource`-Optionen) oder der Definition von Auswahloptionen (`named choice`) dienen. Zudem sind die Größe der Transformationsausgabe und die Laufzeit der Transformation eng mit den jeweils dominierenden Kconfiglib-Erweiterungen verknüpft. Zusammenfassend transformiert der Prototyp die Spezifikationsdateien weitgehend automatisiert und stößt bei Ausnahmefällen (`(named)choice`-Optionen) an seine Grenzen. Diese Ausnahmefälle lassen sich wie in Diskussion 5.3.3 geschildert, prinzipiell durch zukünftige Verbesserungen überwinden.

6. Verwandte Arbeiten

Im Rahmen dieser Arbeit war es insbesondere für die Konzeption der Transformationsregeln erforderlich, zunächst ein Verständnis über die Syntax und Semantik der Linux-Kconfig-Sprache zu erlangen. Während [35] eine formale Semantik der Linux-Kconfig-Sprache definiert, beschränkt sich die vorliegende Arbeit lediglich auf die Darstellung einzelner Sprachelemente der Linux-Kconfig-Sprache in Form einer informellen BNF-Notation. Diese Syntaxdarstellung ermöglicht es, die vom Sprachumfang abweichenden Kconfiglib-Erweiterungen zu identifizieren.

Im Rahmen dieser Arbeit spielen die Kconfiglib-Spezifikationsdateien, die das Variabilitätsmodell eines Projekts wie Zephyr kapseln, eine zentrale Rolle. Diese Dateien dienen als Eingabe des entwickelten Transformationsprototyps. Im Vergleich zu Werkzeugen wie Undertaker [19] oder KConfigReader [28], handelt es sich bei dem Transformationsprototyp nicht um ein Analysewerkzeug, das gegebene Linux-Kconfig-Spezifikationsdateien in aussagenlogische Formeln überführt. Hingegen überführt der Transformationsprototyp die Kconfiglib-basierten Spezifikationsdateien weitgehend automatisiert in eine annähernd äquivalente Darstellung der Linux-Kconfig-Sprache. Undertaker berücksichtigt sowohl das durch Kconfig-Spezifikationen definierte Variabilitätsmodell als auch die tatsächlich im Quellcode realisierte Variabilität, während die vorliegende Arbeit ausschließlich definierte Variabilität berücksichtigt.

Der Transformationsprototyp nutzt die vom Parser bereitgestellten Informationen zu den Sprachelementen der Kconfiglib-Spezifikationsdateien, um die für die Transformationsregeln erforderlichen Inhalte zu extrahieren und in internen Datenstrukturen abzulegen. Ähnlich dazu wird im Framework BYOS [42] die Kconfiglib-Bibliothek zur Extraktion des Variabilitätsmodells aus den Kconfig-Spezifikationsdateien des Linux-Kernels eingesetzt. Diese extrahierte Wissensbasis wird dann in Kombination mit Large Language Models (LLMs) verwendet, um den Herausforderungen von LLMs beim automatisierten Kernel-Tuning im Linux-Kernel zu begegnen. Im Gegensatz dazu ist das Kernel-Tuning nicht Bestandteil der Untersuchung dieser Arbeit. Der hier vorgestellte Lösungsansatz bindet keine LLMs ein und wendet den Transformationsprototyp auf Kconfiglib-basierte Spezifikationen an.

7. Fazit

Das Linux-Kconfig-Konfigurationssystem und die grundlegenden Linux-Kconfig-basierten Spezifikationsdateien unterstützen die Verwaltung der hochkomplexen Variabilität des Linux-Kernels. Diese Mechanismen inspirierten die Entwicklung von Projekten wie Kconfiglib, das den Sprachumfang der Linux-Kconfig-Sprache erweitert. Diese Kconfiglib-Erweiterungen verhindern allerdings die automatisierten Analysen von Kconfiglib-basierten Softwareproduktlinien wie Zephyr, da deren zugrunde liegende Spezifikationsdateien von etablierten Werkzeugen wie KConfigReader [28] nicht verarbeitet werden können.

In Kapitel 3 wurde zunächst die Funktionsweise von Kconfiglib verdeutlicht und die Verarbeitung dieser Bibliothek in Open-Source-Projekten untersucht. Die manuelle Analyse von GitHub-Projekten hat gezeigt, dass Kconfiglib in mehr als 1600 Projekten eingesetzt wird. Die 774 öffentlich sichtbaren Projekte wurden kategorisiert, gefiltert und abschließend ausgewertet. Dadurch konnten elf Projekte identifiziert werden, welche überwiegend für IoT-Anwendungen zugeschnitten sind und Kconfiglib aufgrund der einfachen Integration, der plattformübergreifenden Einsetzbarkeit, sowie der bereitgestellten Funktionen aktiv verwenden.

Darüber hinaus wurden zusätzlich zu der ursprünglichen Kconfiglib-Bibliothek drei weitere Kconfiglib-Varianten identifiziert. Eine Variante hat die Weiterentwicklung von Kconfiglib übernommen, während die beiden anderen Varianten projektspezifische Abspaltungen darstellen. Die Untersuchung des Sprachumfangs dieser Kconfiglib-Varianten hat gezeigt, dass Kconfiglib-Erweiterungen auf hoher Ebene in zwei Gruppen aufgeteilt werden können, nämlich in für die Linux-Kconfig-Sprache unbekannte Sprachelemente wie `rsource`-Optionen und Elemente wie `optional`-Attribute, die in der Linux-Kconfig-Sprache nicht mehr vorgesehen sind. Da Kconfiglib-Varianten projektspezifische Attribute wie `configdefault` oder `warning` umfassen, existiert kein Parser, der Spezifikationsdateien aller Kconfiglib-Varianten verarbeiten kann.

Im nächsten Schritt wurden die ermittelten Kconfiglib-Erweiterungen systematisch auf die Syntax der Kconfig-Sprache abgebildet und basierend darauf die zugehörigen Transformationsregeln abgeleitet. Die Relevanz der vom Parser bereitgestellten

Informationen wurde erkennbar, insbesondere bei den Transformationsregeln für `configdefault`- und `named choice`-Optionen. Diese Regeln erfordern eine Extraktion der relevanten Parser-Informationen sowie eine Berücksichtigung zahlreicher propagierter Abhängigkeiten aus den übergeordneten Sprachelementen.

In Kapitel 4 wurden die konzipierten Transformationsregeln in Form eines Python-basierten Transformationsprototyps umgesetzt. Dieser Prototyp bindet zum initialen Einlesen der zu transformierenden Spezifikationsdateien den für die Kconfiglib-Variante benötigten Parser ein. Anschließend extrahiert der Prototyp die erforderlichen Informationen aus der Parser-Ausgabe und wendet die einzelnen Transformationsregeln auf die Spezifikationsdateien an. Dadurch werden die Kconfiglib-basierten Spezifikationsdateien weitgehend automatisiert in eine annähernd äquivalente Darstellung der Linux-Kconfig-Sprache überführt.

In Kapitel 5 wurde der Lösungsansatz evaluiert, indem der Transformationsprototyp auf ausgewählte Open-Source-Projekte angewendet wurde und ein stichprobenbasiertes Testen von eigenständig erstellten MWEs erfolgte. Im Rahmen der Ergebnisdiskussion wurde festgestellt, dass die Auflösung projektspezifischer Variablen einen entscheidenden Einfluss auf die Anwendbarkeit des Prototyps hat. Zudem wurde erkannt, dass der Prototyp weitgehend automatisiert die Spezifikationsdateien der Open-Source-Projekte transformieren kann. Es wurden Ausnahmefälle identifiziert, die eine vollständige Automatisierung verhindern, aber prinzipiell überwindbar sind und eine zukünftige Erweiterung des Transformationsprototyps erfordern. Darüber hinaus zeigte sich, dass mehrere Projekte Kconfiglib-Erweiterungen wie `rsource`- und `source`-Optionen am häufigsten verwenden und dass der Einfluss des Transformationsprototyps auf die resultierende Transformationsausgabe stark von den im Projekt dominierenden Erweiterungen abhängt.

7.1 Zukünftige Arbeiten

Zukünftige Arbeiten sollten sich zunächst auf die Behebung identifizierter Ausnahmefälle konzentrieren, um das angestrebte Ziel einer vollständig automatisierten Transformation anzusprechen. Hierzu verweist Diskussion 5.6.3 auf die mögliche Vorgehensweise. Anschließend daran könnten sich zukünftige Arbeiten auf einen formalen Beweis oder eine umfassende empirische Analyse der Transformationsregeln konzentrieren, um ihre Korrektheit sicherzustellen. Abschließend, da der Transformationsprototyp einen Vorverarbeitungsschritt zur Extraktion von Feature-Modellen bereitstellt, könnten sich zukünftige Arbeiten auf die Extraktion von Feature-Modellen aus der Transformationsausgabe mithilfe von Werkzeugen wie KConfigReader [28] oder Torte beschäftigen.[92]

[92] https://github.com/ekuiter/torte

Literaturverzeichnis

[1] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three Cases of Feature-Based Variability Modeling in Industry. pages 302–319, 2014. ISBN 978-3-319-11653-2. doi: 10.1007/978-3-319-11653-2_19. (zitiert auf Seite 1)

[2] Yufei Li, Liang Bao, Kaipeng Huang, and Chase Wu. Csat: Configuration structure-aware tuning for highly configurable software systems. 222(C):112316, April 2025. ISSN 0164-1212. doi: 10.1016/j.jss.2024.112316. (zitiert auf Seite 1)

[3] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. 41(8):820–841, August 2015. ISSN 0098-5589. doi: 10.1109/TSE.2015.2415793. (zitiert auf Seite 1)

[4] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. pages 45–54, 2001. (zitiert auf Seite 1)

[5] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in Software Systems-a Systematic Literature Review. 40(3): 282–306, 2013. (zitiert auf Seite 1)

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Berlin, Heidelberg, 2013. ISBN 978-3-642-37520-0. doi: 10.1007/978-3-642-37521-7. (zitiert auf Seite 1)

[7] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA, USA, 1990. (zitiert auf Seite 1)

[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. 39(12):1611–1640, 2013. ISSN 0098-5589. doi: 10.1109/TSE.2013.34. (zitiert auf Seite 1, 6 und 7)

[9] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. Towards complex product line variability modelling: Mining relationships from non-boolean descriptions. 156(C):341–360, October 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.06.002. (zitiert auf Seite 1)

[10] Necip Fazıl Yıldıran, Jeho Oh, Julia Lawall, and Paul Gazzillo. Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times. 1(FSE), July 2024. doi: 10.1145/3643746. (zitiert auf Seite 1 und 5)

[11] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. pages 893–905, New York, NY, USA, 2021. ISBN 9781450385626. doi: 10.1145/3468264.3468578. (zitiert auf Seite 1 und 5)

[12] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. 35(1), December 2025. doi: 10.1145/3729423. (zitiert auf Seite 1 und 5)

[13] Jude Gyimah, Jan Sollmann, Ole Schuerks, Patrick Franz, and Thorsten Berger. A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export. pages 91–96, New York, NY, USA, February 2025. doi: 10.1145/3715340.3715445. (zitiert auf Seite 1 und 6)

[14] David Romero-Organvídez, Pablo Neira, José A. Galindo, and David Benavides. Kconfig Metamodel: A First Approach. pages 55–60, New York, NY, USA, September 2024. doi: 10.1145/3646548.3676548. (zitiert auf Seite 1, 5 und 7)

[15] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the Linux Kernel Variability Model. pages 136–150, Berlin, Heidelberg, 2010. ISBN 3642155782. (zitiert auf Seite 1 und 5)

[16] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The Variability Model of the Linux Kernel. pages 45–51, 2010. (zitiert auf Seite 1, 5 und 6)

[17] Julio Sincero and Wolfgang Schröder-Preikschat. The Linux Kernel Configurator as a Feature Modeling Tool. pages 257–260. University of Limerick, Lero, 2008. (zitiert auf Seite 1 und 6)

[18] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. pages 33–42, New York, NY, USA, 2010. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868300. (zitiert auf Seite 1 und 7)

[19] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. pages 47–60, New York, NY, USA, 2011. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966451. (zitiert auf Seite 1, 6, 7 und 81)

[20] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. An Empirical Study of Configuration Mismatches in Linux. pages 19–28, New York, NY, USA, 2017. ISBN 978-1-4503-5221-5. doi: 10.1145/3106195.3106208. (zitiert auf Seite 1)

[21] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. 47(1):6:1–6:45, June 2014. ISSN 0360-0300. doi: 10.1145/2580950. (zitiert auf Seite 1, 2 und 5)

[22] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models Is Easy. pages 231–240, Pittsburgh, PA, USA, 2009. (zitiert auf Seite 2 und 5)

[23] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. Applications of #SAT Solvers on Feature Models. New York, NY, USA, February 2021. ISBN 9781450388245. doi: 10.1145/3442391.3442404. (zitiert auf Seite 2 und 5)

[24] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. doi: 10.1016/J.IS.2010.01.001. (zitiert auf Seite 2 und 5)

[25] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature Model Differences. pages 629–645, Berlin, Heidelberg, June 2012. ISBN 978-3-642-31094-2. doi: 10.1007/978-3-642-31095-9_41. (zitiert auf Seite 2 und 5)

[26] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. pages 459–468, Washington, DC, USA, April 2010. doi: 10.1109/ICST.2010.43. (zitiert auf Seite 2 und 5)

[27] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. pages 289–301, New York, NY, USA, 2019. ISBN 9781450371384. doi: 10.1145/3336294.3336297. (zitiert auf Seite 2 und 5)

[28] Christian Kästner. Differential Testing for Variational Analyses: Experience From Developing KConfigReader. Technical Report arXiv:1706.09357, 2017. URL http://arxiv.org/abs/1706.09357. (zitiert auf Seite 2, 81, 83 und 84)

[29] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. pages 91–100, 2021. ISBN 9780738146690. doi: 10.1109/ICSE-SEIP52600.2021.00018. (zitiert auf Seite 2 und 6)

[30] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform Sampling From Kconfig Feature Models. Technical Report TR-19-02, University of Texas at Austin, Department of Computer Science, 2019. (zitiert auf Seite 2)

[31] David Bretthauer. Open Source Software: A History. Website, 2001. URL https://opencommons.uconn.edu/libr_pubs/7. (abgerufen am 05.01.2026). (zitiert auf Seite 5)

[32] Linus Torvalds. The Linux Edge. *Communications of the ACM*, 42(4):38–39, 1999. (zitiert auf Seite 5)

[33] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing, And Who Is Sponsoring It. Website, 2016. URL https://www.linuxfoundation.jp/events/2016/08/linux-kernel-development-2016/. (abgerufen am 05.01.2026). (zitiert auf Seite 5)

[34] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting linux for mobile platforms: An empirical study of android. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 629–632. ieee, 2012. doi: 10.1109/ICSM.2012.6405339. (zitiert auf Seite 5)

[35] Steven She and Thorsten Berger. Formal Semantics of the Kconfig Language. Technical report, University of Waterloo, Canada, 2010. (zitiert auf Seite 6, 7 und 81)

[36] Kaan Berk Yaman, Jan Willem Wittler, and Christopher Gerking. Kfeature: Rendering the Kconfig System into Feature Models. pages 134–138, New York, NY, USA, February 2024. ISBN 9798400708770. doi: 10.1145/3634713.3634731. (zitiert auf Seite 6)

[37] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the KConfig Semantics and its Analysis Tools. pages 45–54, New York, NY, USA, 2015. ISBN 978-1-4503-3687-1. doi: 10.1145/2814204.2814222. (zitiert auf Seite 6)

[38] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. pages 421–432, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. (zitiert auf Seite 7)

[39] Sarah Nadi and Ric Holt. The Linux Kernel: A Case Study of Build System Variability. 26(8):730–746, 2014. doi: 10.1002/smr.1595. (zitiert auf Seite 7)

[40] Kconfig Language — The Linux Kernel documentation, 2025. URL https://docs.kernel.org/kbuild/kconfig-language.html. (abgerufen am 20.10.2025). (zitiert auf Seite 7, 8 und 12)

[41] Ulf Magnusson. A linux-based, web-oriented operating system designed to boot quickly, 2011. URL https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-73492. (abgerufen am 10.09.2025). (zitiert auf Seite 19 und 20)

[42] Hongyu Lin, Yuchen Li, Haoran Luo, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Byos: Knowledge-driven large language models bring your own operating system more excellent, 2025. URL https://arxiv.org/abs/2503.09663. (zitiert auf Seite 81)

Verwendung von KI-Systemen

Folgende Abschnitte fassen die Verwendung von KI-Systemen im Rahmen dieser Arbeit zusammen.

Zur Korrektur von Rechtschreib- und Grammatikfehlern wurden die Werkzeuge **LanguageTool** in der kostenfreien Web-Version sowie das kostenlose Sprachmodell **GPT-5.2** von ChatGPT verwendet.

Zur Behebung kleinerer syntaktischer und logischer Fehler während der Implementierung des Prototyps (vgl. Kapitel 4) wurden die eingeschränkten, kostenfreien Sprachmodelle **GPT-5.2** (ChatGPT) und **Sonnet 4.5** (claude.ai) unterstützend eingesetzt. Diese Modelle haben ausschließlich unterstützende Funktionen übernommen, insbesondere bei der Identifikation von Fehlern wie falsch platzierten Abbruchbedingungen in Schleifen oder Problemen beim Iterieren über komplexe Datenstrukturen. Vor allem die internen Methoden `_get_all_choice_configs` und `transform_choice` wurden mithilfe dieser Modelle überprüft. Sämtliche vorgeschlagene Lösungen wurden vor der Übernahme in den Quellcode manuell evaluiert und in vielen Fällen angepasst oder nicht akzeptiert, da sie nicht mit der Gesamtlogik des Transformationsprototyps übereinstimmten.

Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig verfasst habe, dass die Arbeit weder teilweise noch vollständig als benotete akademische Leistung eingereicht wurde und dass ich keine anderen als die angegebenen Hilfsmittel verwendet habe. Ich habe alle Stellen der Arbeit, in denen Quellen wörtlich oder dem Sinn nach verwendet wurden, entsprechend kenntlich gemacht.

Ich bin mir bewusst, dass Verstöße gegen das Urheberrecht zu Unterlassungsansprüchen und Schadensersatzforderungen des Urhebers sowie zu strafrechtlichen Konsequenzen führen können.

Des Weiteren bestätige ich, dass mir bekannt ist, dass die Verwendung von durch Künstliche Intelligenz (KI) generierten Inhalten (einschließlich, aber nicht beschränkt auf Text, Abbildungen, Bilder und Code) in der Arbeit offengelegt werden muss. In diesen Fällen habe ich das verwendete KI-System angegeben, die spezifischen Abschnitte der Arbeit markiert, in denen KI-generierte Inhalte genutzt wurden, und eine kurze Erklärung zum Detaillierungsgrad der Nutzung des KI-Systems bereitgestellt. Zudem habe ich den Grund für die Verwendung dieser Werkzeuge angegeben.

Ich versichere, dass auch bei der Nutzung eines generativen KI-Systems der wissenschaftliche Beitrag vollständig von mir selbst erbracht wurde.

Magdeburg, 20. Januar 2026