RWTH RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LEHRSTUHL INFORMATIK XI SOFTWARE FÜR EINGEBETTETE SYSTEME

Bachelorarbeit

Modulare und Boolesche Abstraktion von SPS-Programmen

Modular and Boolean Abstraction of PLC-Programs

Dimitri Bohlender

31. August 2012

Gutachter:
Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Thomas Noll

Betreuer:
Dipl.-Inform. Sebastian Biallas

Zusammenfassung

Speicherprogrammierbare Steuerungen (SPSen) werden in der Steuerung und Regelung von Anlagen sowie sicherheitskritischen Anwendungen eingesetzt. Gerade in diesem Bereich sind Fehler in der Software oft kritisch. Dieser Herausforderung kann man nur mit formaler Verifikation begegnen.

Arcade.PLC ist die auf Verifikation von SPSen ausgelegte Komponente der am Lehrstuhl Informatik 11 entwickelten Software Arcade, die der Verifizierung von eingebetteten Systemen dient. Arcade.PLC verwendet dazu einen expliziten Model-Checker, welcher auf einem abstrakten Zustandsraum arbeitet, um die grundlegende Problematik der Speicherbeschränktheit bei der Erzeugung vieler Zustände, die sog. Zustandsexplosion, zu bewältigen.

Diese Arbeit beschäftigt sich mit dem Problem der Zustandsexplosion, die aus der frühzeitigen Auswertung von für Arcade.PLC problematischen Ausdrücken in SPS-Programmen hervorgeht. Es werden Abstraktionen entwickelt, welche die für eine Verifikation nicht essentiellen Auswertungen ausblenden und sich in das Konzept der Gegenbeispiel-geleiteten Abstraktionsverfeinerung einbetten lassen. Diese Techniken ermöglichen die Verifikation von Programmen, deren Untersuchung zuvor nicht effektiv möglich war.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 31. August 2012

(Dimitri Bohlender)

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Aufgabenstellung 1
- 1.2 Gliederung 2

2 Grundlagen **3**
- 2.1 Speicherprogrammierbare Steuerungen 3
  - 2.1.1 Aufbau und Funktionsweise 3
  - 2.1.2 Programmierung 4
- 2.2 Model-Checking 4
  - 2.2.1 Grundlagen 4
  - 2.2.2 Modell 5
  - 2.2.3 Spezifikation 5
  - 2.2.4 Gegenbeispiele und Zeugen 7
- 2.3 Abstraktionen 9
  - 2.3.1 Abstraktion als Relation 9
  - 2.3.2 Überapproximation 10
  - 2.3.3 Gegenbeispiel-geleitete Abstraktionsverfeinerung 11
  - 2.3.4 Relevante Abstraktionen 12
- 2.4 Arcade.PLC 14
  - 2.4.1 Grundlegender Ablauf 15
  - 2.4.2 Erzeugung des Zustandsraums 16
  - 2.4.3 Abstraktionsverfeinerung der Variablen 16

3 Abstraktionen **21**
- 3.1 Boolesche Abstraktion 21
  - 3.1.1 Motivation 21
  - 3.1.2 Grundidee 23
  - 3.1.3 Initiale Abstraktion 23
  - 3.1.4 Abstraktionsverfeinerung 25
- 3.2 Modulare Abstraktion 30
  - 3.2.1 Motivation 30
  - 3.2.2 Grundidee 32
  - 3.2.3 Initiale Abstraktion 32

3.2.4 Abstraktionsverfeinerung 34

4 Fallstudien 37

4.1 Verifikation der Problemfälle 37
4.2 Verifikation einer beschränkten Funktion 38
4.3 Verifikation eines Funktionsbausteins nach PLCopen 39

5 Verwandte Arbeiten 41

5.1 Model-Checking speicherprogrammierbarer Steuerungen 41
5.2 Abstraktionstechniken 41
5.3 Gegenbeispiel-geleitete Abstraktionsverfeinerung 42

6 Zusammenfassung und Ausblick 43

6.1 Ausblick 43
6.1.1 Erweiterung der Heuristiken 43
6.1.2 Erweiterung der Überprüfung der Realisierbarkeit 45
6.1.3 Erweiterung des Zusammenspiels verschiedener Abstraktionen 45
6.2 Fazit 46

Literaturverzeichnis 47

1 Einleitung

Die weite Verbreitung von eingebetteten Systemen ist nicht von der Hand zu weisen. Von Kühlschränken über Unterhaltungselektronik bis hin zu Fahrzeugen und Fertigungsanlagen werden speziell an die Aufgabe angepasste eingebettete Systeme verwendet, um ihren Dienst im Hintergrund zu verrichten. Dabei wird meist eine Kombinationslösung aus Hard- und Software verwendet, um die Flexibilität einer Software, aber auch die Leistungsfähigkeit von direkt auf Hardware realisierter Funktionalität nutzen zu können.

Im Bereich der Steuerung und Regelung von Maschinen und Anlagen ist die Verwendung von Verbünden aus Kontrollgeräten, insbesondere den sogenannten speicherprogrammierbaren Steuerungen (SPSen), etabliert. Gerade in diesem Anwendungsfall kommt es darauf an, dass die Software keine Fehler aufweist. In solchen automatisierten Prozessen kann das Resultat einer unbemerkten Fehlfunktion von Produktionsausfällen, über Fehler im Produkt bis hin zur Beschädigung der Produktionsanlage reichen. Wird das Fehlverhalten der Software also nicht frühzeitig erkannt, kann dies teure Konsequenzen haben.

Der naive Ansatz, um solchen Fehlern zu begegnen, wäre das Testen des Programms. Gründliches Testen ist allerdings ein sehr zeitintensiver Prozess. Des Weiteren lassen sich allein durch Testen möglicherweise nicht alle Fehler aufdecken, da man nicht alle Spezifikationen an eine Software testen kann. So kann eine Spezifikation, die besagt, dass ein bestimmter Zustand nie erreicht wird, nicht durch Testen belegt, sondern höchstens widerlegt werden, da ein Programm potenziell unendlich viele Zustände haben kann.

Model-Checker vermögen allerdings, auf mathematisch fundierter Basis, auch solche Spezifikationen an einem System zu prüfen und im Falle eines Fehlschlags als Nebenprodukt der Verifikation einen Beispielablauf auszugeben, welcher die Spezifikation verletzt. Daher sind formale Methoden für solche Systeme von größerer Bedeutung und nach IEC 61508 sogar empfohlen [Int98].

1.1 Aufgabenstellung

Arcade ist eine am Lehrstuhl 11 für Informatik an der RWTH entwickelte Software zur Verifizierung von eingebetteten Systemen und Arcade.PLC der auf die Überprüfung von SPSen ausgelegte Teil. In einer früheren Veröffentlichung [BBK10] wurde der Model-Checker von Arcade.PLC um die Unterstützung für Abstraktionen

und das Konzept der Gegenbeispiel geleiteten Abstraktionsverfeinerungen erweitert. Hierbei wurde im Ausblick dargestellt, dass die Auflösung der Bedingungen an Variablen insbesondere dann Probleme bereitet, wenn Abhängigkeiten von mehr als einer Variablen auftreten. Dies führte dazu, dass Programme, die solche problematischen Auflösungen enthielten, nicht untersucht werden konnten, da es stets zu einer *state explosion*[1][CGJ+00b] kam – unabhängig davon, ob die problematische Stelle relevant für die zu verifizierende Spezifikation war.

Ziel dieser Arbeit ist die Entwicklung zweier Abstraktionsmethoden, um der Zustandsexplosion, die aus der frühzeitigen Auswertung solcher problematischer Ausdrücke in SPS-Programmen hervorgeht, zu entgehen und die Menge, der durch ARCADE.PLC verifizierbaren Programme, zu erweitern. Die Verifikation einer bestimmten Spezifikation erfordert in der Regel nicht die Auswertung eines jeden Ausdrucks eines SPS-Programms. Die Idee ist die problematischen Ausdrücke aggressiv zu abstrahieren und zur Laufzeit zu entscheiden, welche abstrahierten Ausdrücke wieder eingeblendet werden müssen. Da wir eine Überapproximation verwenden und zur Laufzeit wieder einblenden wollen, bietet sich hier die Einbindung der Abstraktion im Konzept der Gegenbeispiel-geleiteten Abstraktionsverfeinerung an.

So werden die Boolesche und die Modulare Abstraktion eingeführt, um unnötige Zustandsexplosionen bei Operationen mit mehreren Variablen bzw. Funktionsaufrufen zu umgehen und in ARCADE.PLC eingebettet.

1.2 Gliederung

Zunächst werden in Kapitel 2 Grundlagen des Model-Checking im Kontext von speicherprogrammierbaren Steuerungen eingeführt und wichtige Aspekte im Umgang mit abstrakten Zustandsräumen beleuchtet. Zusätzlich wird darauf eingegangen, wie diese Grundlagen in die Vorgehensweise von ARCADE.PLC eingehen. Anschließend werden in Kapitel 3 Probleme des Ansatzes von ARCADE.PLC dargestellt und die Boolesche und Modulare Abstraktion entwickelt, welche diese lösen sollen. Beide Abstraktionstechniken zielen auf die Abstraktion problematischer Ausdrücke ab, wobei die erste If-Statements abstrahiert, während die zweite für die Abstraktion von Aufrufen sorgt. In Kapitel 4 werden Fallstudien dieser Techniken anhand von zuvor problematischen Programmen durchgeführt. Darauf folgt in Kapitel 5 ein Überblick über Arbeiten, die in den Punkten Model-Checking von SPS-Programmen, Abstraktionen und Gegenbeispiel-geleitete Abstraktionsverfeinerung verwandt mit dieser Arbeit sind. Den Abschluss bildet Kapitel 6, welches eine Zusammenfassung und einen Ausblick über Ausbaumöglichkeiten der entwickelten Techniken gibt.

[1]state explosion, engl. Zustandsexplosion

2 Grundlagen

2.1 Speicherprogrammierbare Steuerungen

Eine speicherprogrammierbare Steuerung (SPS) ist ein Gerät, das in der Automatisierungsindustrie zur Anlagensteuerung und -regelung verwendet wird. Sie kann auf digitaler Basis programmiert werden und löst damit die zuvor eingesetzte verbindungsprogrammierte Steuerung ab.

Im Folgenden wird nun der Aufbau und die Funktionsweise einer SPS nahegebracht, bevor die Besonderheiten in der Programmierung dieser dargestellt werden.

2.1.1 Aufbau und Funktionsweise

Eine SPS verfügt über eine Menge von Ein- und Ausgängen sowie ein Programm, welches das Verhalten der SPS festlegt. Die Eingänge erhalten ihre Belegung in der Regel von Sensoren, während die Ausgänge mit den Aktoren der gesteuerten Anlage verbunden sind. Des Weiteren werden SPSen oft im Verbund eingesetzt, so dass an den Eingängen die Ausgangswerte einer vorherigen SPS eingehen können.

Es gibt verschiedene Konzepte der Funktionsweise einer SPS. Das am weitesten verbreitete ist das der zyklusorientierten SPS. Dabei werden zu Beginn eines Zyklus die Werte an den Eingängen abgefragt und dem auf der SPS ausgeführten Programm als Variablen zur Verfügung gestellt. Dieses Programm weist den Ausgängen in Abhängigkeit von der Belegung der Eingänge Werte zu. Abgesehen von den Variablen für Ein- und Ausgänge kann im Programm allerdings auch auf interne, nichttemporäre Variablen zugegriffen werden, deren Werte auch über einen Zyklus hinaus erhalten bleiben. Schließlich werden die den Ausgängen zugewiesenen Werte tatsächlich geschrieben und der Zyklus neu begonnen. Die Zwischenzustände, die während des Programmdurchlaufs angenommen werden, sind bei dieser Betriebsweise nach außen hin nicht sichtbar.

Es sei darauf hingewiesen, dass es zudem noch die ereignisgesteuerte SPS gibt, welche auf Statuswechsel – sogenannte Ereignisse – am Eingang reagieren und diese in ihrer Eingangsreihenfolge abarbeiten kann.

Die vorliegende Arbeit beschäftig sich allerdings nur mit den erstgenannten und bekannteren zyklusorientierten SPSen.

2.1.2 Programmierung

Wie bereits erwähnt, dient das Programm der Belegung der Ausgänge in Abhängigkeit von den Eingängen. Dabei kann das Programm aus mehreren Teilprogrammen, den Programmorganisationseinheiten (POEs), aufgebaut sein, die sich gegenseitig aufrufen können, wobei Rekursion jedoch nach IEC 61131-3 [Int03] unzulässig ist. Man beachte dabei den Unterschied zwischen dem Aufruf von Funktionen und dem von POEs. Letztere können, die in Abschnitt 2.1.1 erwähnten nichttemporären Variablen verwenden, die ihren Wert im Gegensatz zu Eingangs- und Ausgangsvariablen über einen Zyklus hinaus behalten. Funktionen können dagegen keine Seiteneffekte haben. Deren Rückgabewert hängt direkt von den Eingaben ab.

Der IEC 61131-3 Standard beschreibt zudem wie die Programme zu formulieren sind. Dazu werden fünf Programmiersprachen definiert, welche abgesehen von textuellen auch grafische Sprachen enthalten. Diese Vielseitigkeit in der Darstellung eines SPS-Programms ist eine Herausforderung in der Verifikation. Der Umgang mit dieser Problematik wird in Abschnitt 2.4.1 beschrieben. Nähere Informationen zur Programmierung sind für die folgende Ausführung nicht relevant. Für eine detaillierte Einführung sei auf [JT09] verwiesen.

2.2 Model-Checking

Model-Checking [CGP99] wurde Anfang der 80er Jahre ursprünglich als Methode zur Verifikation nebenläufiger Prozesse entwickelt [EC80, CES86], schließlich aber auch auf alleinstehende Prozesse angewandt [CGP99, Cla08].

Im Folgenden werden die Funktionsweise eines Model-Checkers, sowie dessen die Ein- und Ausgabeobjekte dargestellt.

2.2.1 Grundlagen

Ein Model-Checker ist eine Software, welche eine Programmbeschreibung in Form eines Modells, wie es im folgenden Abschnitt beschrieben wird, auf die Erfüllung einer Spezifikation prüfen kann. Oft liegt die Programmbeschreibung aber nur als Kontrollflussgraph oder als Quelltext vor und muss zunächst in ein für die Überprüfung geeignetes Modell überführt werden. Die zu verifizierende Spezifikation wird in Form einer Formel Φ einer temporalen Logik ausgedrückt. Der Model-Checker gibt daraufhin als Ergebnis aus, ob die Spezifikation Φ von dem eingegebenen Modell $\mathcal{T}$ eingehalten wurde. Wurde diese nicht eingehalten, wird zusätzlich ein Gegenbeispiel ausgegeben, das beschreibt, welche Eingaben zur Verletzung der Spezifikation geführt haben. Die Bezeichnung dieses Vorgangs als „Model-Checking“ geht daraus hervor, dass dieser Prozess, dem aus der mathematischen Logik bekannten Prüfen ob $\mathcal{T}$ Modell von Φ ist, geschrieben $\mathcal{T} \models \Phi$, gleichkommt.

2.2.2 Modell

Das beim Model-Checking verwendete Modell ist ein *Transitionssystem*. Im Allgemeinen können Transitionssysteme mehrere Startzustände haben. Im Kontext von ARCADE ist die Definition mit nur einem Startzustand allerdings angebracht, da die von ARCADE überprüfbaren Systeme stets genau einen definierten Startzustand haben. Formal ist das Transitionssystem in Anlehnung an [BK08] als Tupel $\mathcal{T} = (S, s_0, \rightarrow, L)$ definiert, wobei

- S die Menge der Zustände ist,
- $s_0 \in S$ der Startzustand ist,
- $\rightarrow \subseteq S \times S$ die Übergangsrelation ist,
- $L : S \rightarrow 2^{AP}$ die Beschriftungsfunktion ist, wobei AP die Menge der atomaren Aussagen ist.

Zur Veranschaulichung dieser Struktur zeigt Abbildung 2.1 ein Transitionssystem $\mathcal{T}$ über der Menge der atomaren Aussagen $AP = \{a, b\}$.

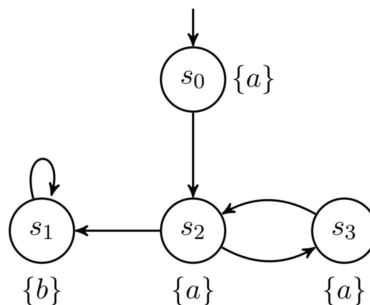

Abbildung 2.1: Transitionssystem $\mathcal{T}$ über $AP = \{a, b\}$

2.2.3 Spezifikation

Wie bereits erwähnt, liegt die Spezifikation in Form einer Formel Φ einer temporalen Logik vor. Die beiden wichtigsten Vertreter solcher Logiken sind gemäß [CV03] die *linear time logic* (LTL) [Pnu77] und die *computation tree logic* (CTL) [BAMP81]. Dabei beschreibt Erstere Aussagen über Pfade im Transitionssystem, während Letztere Aussagen über den ganzen Berechnungsbaum, hier die verzweigte Struktur der Zustände, formalisiert. Die Ausdrucksmächtigkeit beider Logiken ist allerdings unvergleichbar, da es jeweils Eigenschaften gibt, die in der einen Logik zwar ausgedrückt werden können, in der jeweils anderen aber kein Äquivalent haben. Die

genaue Definition von LTL ist in dieser Arbeit nicht von Relevanz. Daher werden im Folgenden lediglich die Syntax und Semantik von CTL in Anlehnung an [BK08] definiert.

CTL

Die Syntax von CTL Formeln kann wie folgt in Form von einer Grammatik beschrieben werden:

$$\Phi ::= \mathit{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

wobei a ein Element aus der Menge der atomaren Aussagen AP des zu prüfenden Transitionssystems, ist und φ eine *Pfadformel* darstellt. Pfadformeln werden über folgende Grammatik definiert:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2$$

wobei Φ, Φ_1 und Φ_2 die zuvor definierten *Zustandsformeln* sind. Intuitiv sagen Zustandsformeln etwas über die Eigenschaften in einem Zustand aus, während Pfadformeln Aussagen über Pfadeigenschaften machen.

Die Semantik solcher Formeln lässt sich aus der folgenden Definition der Erfüllbarkeitsrelation $\models$ entnehmen. Sei $\mathcal{T} = (S, s_0, \rightarrow, L)$ ein Transitionssystem ohne Endzustände[1] über einer Menge von atomaren Aussagen AP und s ein Zustand dieses Systems. Dann ist die Erfüllbarkeitsrelation gemäß [CV03] wie folgt definiert:

$$\begin{array}{lcl}
s \models a & \Leftrightarrow & a \in L(s) \\
s \models \neg\Phi & \Leftrightarrow & s \not\models \Phi \\
s \models \Phi \wedge \Psi & \Leftrightarrow & s \models \Phi \text{ und } s \models \Psi \\
s \models \exists\varphi & \Leftrightarrow & \pi \models \varphi \text{ für einen in } s \text{ beginnenden Pfad } \pi \\
s \models \forall\varphi & \Leftrightarrow & \pi \models \varphi \text{ für alle in } s \text{ beginnenden Pfade } \pi
\end{array}$$

wobei jeder Zustand s die Formel *true* erfüllt. Für einen Pfad π ist die Erfüllbarkeitsrelation definiert als

$$\begin{array}{lcl}
\pi \models \bigcirc\Phi & \Leftrightarrow & \pi = s_1 s_2 s_3 \ldots \text{ und } s_2 \models \Phi \\
\pi \models \Phi \mathsf{U} \Psi & \Leftrightarrow & \exists_{j \in \mathbb{N}_0} \pi = s_1 s_2 s_3 \ldots s_j \text{ und } \forall_{0 \leq i < j} s_i \models \Phi \text{ und } s_j \models \Psi
\end{array}$$

Oft werden auch weitere Symbole für Pfade verwendet, die allerdings keine Erweiterung der Aussagestärke darstellen, sondern „Syntaxzucker" sind. Die Semantik solcher Symbole lässt sich aus einer Kombination der beschriebenen Elemente ableiten. Im Folgenden werden die weit verbreiteten Symbole $\Diamond$ und $\Box$ vorgestellt, welche besagen, dass „schließlich ein Zustand" bzw. „alle Zustände des Pfades" ein bestimmtes Φ erfüllen müssen:

[1] Diese Vereinfachung beeinflusst die Aussagestärke des Transitionssystems gemäß [BK08] nicht

$$\pi \models \Diamond\Phi \quad \Leftrightarrow \quad \pi = s_1 s_2 \dots s_j \dots \;\wedge\; \exists_j s_j \models \Phi, j \in \mathbb{N}_0$$
$$\pi \models \Box\Phi \quad \Leftrightarrow \quad \pi = s_1 s_2 \dots \;\wedge\; \forall_i s_i \models \Phi$$

Die Definition der Erfüllbarkeitsrelation für das Transitionssystem selbst beruht folgendermaßen auf der Definition der Erfüllbarkeitsrelation für den Startzustand s_0:

$$\mathcal{T} \models \Phi \Leftrightarrow s_0 \models \Psi$$

∀CTL

Wie sich in Abschnitt 2.3.1 zeigen wird, ist ∀CTL eine im Kontext von Abstraktionen bedeutende Teilmenge von CTL.t ∀CTL ergibt sich aus CTL, indem man die größte geschlossene Formelmenge bildet, die bei Verbieten der Negation vor allen Zustandsformeln bis auf atomaren Aussagen möglich ist. Die sich ergebende Formelmenge lässt sich durch die folgende Grammatik beschreiben:

$$\Phi ::= \mathit{true} \mid \mathit{false} \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall\varphi$$

wobei a ein Element aus der Menge der atomaren Aussagen AP des zu prüfenden Transitionssystems ist und φ auch hier eine Pfadformel darstellt. Da $\forall\varphi$ die Negation von $\exists\neg\varphi$ ist, muss entsprechend auch eins der beiden Symbole weggelassen werden. Im Falle von ∀CTL wird ∃ weggelassen. Den symmetrischen Fall ∃CTL gibt es allerdings auch. Wie man sieht, sind Negationen in ∀CTL nur vor atomaren Aussagen möglich und der ∃-Quantor darf nicht verwendet werden. Die Pfadformeln werden über folgende Grammatik definiert:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2 \mid \Phi_1 \mathsf{R} \Phi_2$$

wobei Φ, Φ_1 und Φ_2 auch hier Zustandsformeln sind und R die Negation von U ist – Details sind an der Stelle nicht weiter von Relevanz.

Die genaue Kenntnis des Algorithmus zum Model-Checken von CTL Formeln ist zum Verständnis dieser Arbeit nicht nötig. Es reicht zu wissen, dass die Überprüfung einer CTL Formel durch schrittweise Überprüfung der Teilformeln erfolgen kann. Von besonderer Bedeutung ist dagegen das Verständnis des, im Falle eines negativen Ausgangs der Überprüfung, ausgegebenen Gegenbeispiels. Daher wird im folgenden Abschnitt der Fokus auf den Zusammenhang zwischen einer zu prüfenden Formel und einem möglichen Gegenbeispiel gelegt.

2.2.4 Gegenbeispiele und Zeugen

Wird eine Formel Φ an einem Modell $\mathcal{T}$ überprüft und festgestellt, dass diese Formel nicht erfüllt wird, also $\mathcal{T} \not\models \Phi$ gilt, so wird vom Model-Checker abgesehen von dieser Feststellung ein Gegenbeispiel ausgegeben, welches darstellt, weshalb die Formel nicht gilt. Hier wird, wie auch in [CV03], zwischen zwei Formelgruppen unterschieden:

1. *Allquantifizierte Formeln*: Dies sind Formeln, die mit einem ∀-Quantor beginnen. Gilt eine derartige Formel nicht, dann kann ein Gegenbeispiel gefunden werden, welches den Befund belegt.

2. *Existenzquantifizierte Formeln*: Dies sind Formeln, die mit einem ∃-Quantor beginnen. Analog können hier bei einer geltenden Formel sogenannte *Zeugen* der Gültigkeit gefunden werden.

Gegenbeispiele bzw. Zeugen können allerdings beide sowohl in Form von endlichen als auch unendlichen Pfaden vorliegen.

In dieser Arbeit werden mögliche Gegenbeispiele in zwei Kategorien eingeteilt – solche die *linear* sind und solche die es nicht sind. Linearität meint an dieser Stelle, dass der sich ergebende Pfad keine Zyklen oder Verzweigungen hat. Beispiele für beide Typen sind Abbildung 2.2 zu entnehmen.

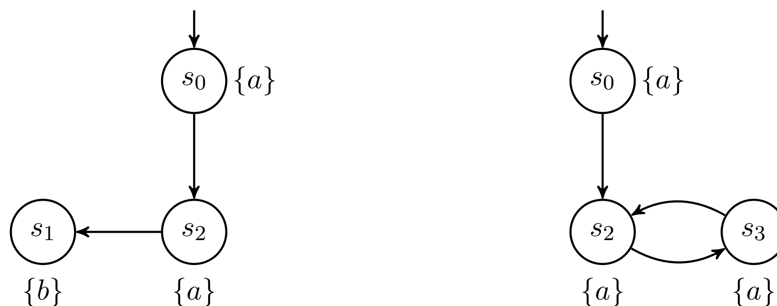

Abbildung 2.2: lineares (links) und nichtlineares Gegenbeispiel (rechts)

Diese Gegenbeispiele resultieren aus der Prüfung ob $\mathcal{T}$, aus Abbildung 2.1, die Formeln $\Phi_1 := \forall\Box a$ bzw. $\Phi_2 := \neg\exists\Box\exists\Diamond a$ erfüllt. Ebenso können diese aber auch als Zeugen für die Formeln $\Phi_1 := \neg\forall\Box a$ bzw. $\Phi_2 := \exists\Box\exists\Diamond a$ verstanden werden.

Anhand der Unterschiede in der Komplexität der Gegenbeispiele kann man bereits erkennen, dass man beim Arbeiten mit Gegenbeispielen leichter mit den stets gleich strukturierten, linearen umgehen kann, als mit verzweigten und zyklischen. Die in dieser Ausarbeitung entwickelten Abstraktionen machen daher die Unterscheidung zwischen den beiden Typen und wenden entsprechend verschiedene Algorithmen an, wobei der Algorithmus für lineare Gegenbeispiele das Wissen um die stets gleiche Struktur ausnutzt, während der für die nichtlinearen von heuristischer Natur ist. Um die Transitionssysteme und mögliche Gegenbeispiele möglichst klein zu halten, um der Zustandsexplosion zu entgehen, werden in der Regel Abstraktionen eingesetzt, welche im Fokus des folgenden Abschnitts liegen.

2.3 Abstraktionen

In den vorherigen Abschnitten wurde immer wieder auf das Problem der Zustandsexplosion hingewiesen. Eine gängige Methode diesem Problem zu begegnen, ist die Verwendung von Abstraktionen, welche dazu führen, dass Zustände die bestimmte Anforderungen erfüllen in Makrozuständen zusammengefasst werden können und somit einen kleineren Zustandsraum schaffen. Der Begriff der Abstraktion wird im Folgenden zunächst formal eingeführt und in den anschließenden Abschnitten an Beispielabstraktionen veranschaulicht.

2.3.1 Abstraktion als Relation

Damit die Zustände $\mathcal{T}$ des konkreten Transitionssystems von den Makrozuständen des abstrakten Zustandsraums $\mathcal{T}'$ so repräsentiert werden, dass eine Überprüfung einer Spezifikation in $\mathcal{T}'$ etwas über die Einhaltung der Spezifikation in $\mathcal{T}$ aussagt, muss $\mathcal{T}$ von $\mathcal{T}'$ *simuliert* werden können. Die Idee hinter der Definition der Simulationsrelation ist, alle Pfade, die im konkreten Transitionssystem möglich sind, auch im abstrahierten System zu erhalten.

Sei $\mathcal{T} = (S, s_0, \to, L)$ ein konkretes Transitionssystem und $\mathcal{R}$ eine Äquivalenzrelation auf einer Zustandsmenge S, so dass

$$[s]_{\mathcal{R}} := \{t \in S \mid (s,t) \in \mathcal{R}\}$$

die Äquivalenzklasse von $s \in S$ beschreibt. Dann kann, in Anlehnung an [BK08], ein abstrakter Zustandsraum wie folgt definiert werden:

$$\mathcal{T}' = (S/\mathcal{R}, [s_0]_{\mathcal{R}}, \to', L')$$

wobei

- $S/\mathcal{R} = \{[s]_{\mathcal{R}} \mid s \in S\}$ der neue Zustandsraum ist, dessen Makrozustände die Äquivalenzklassen der Zustände von $\mathcal{T}$ sind
- $s_i \to s_j$ impliziert, dass $[s_i]_{\mathcal{R}} \to' [s_j]_{\mathcal{R}}$ gelten muss
- $L'([s_i]_{\mathcal{R}}) = L(s_i)$

Existiert eine solche Relation, sagt man, dass $\mathcal{T}$ von $\mathcal{T}'$ simuliert werden kann, geschrieben $\mathcal{T} \preceq \mathcal{T}'$.

Wie man leicht erkennt, ist die Simulation nicht symmetrisch. Soll also für eine Teilmenge der Formeln von CTL das gewünschte Resultat $\mathcal{T}' \models \Phi \Rightarrow \mathcal{T} \models \Phi$ gelten, so kann diese Teilmenge nicht unter Negation abgeschlossen sein. Letzteres würde implizieren, dass die Simulationsrelation symmetrisch sei, was ein Widerspruch wäre.

Entsprechend dieser Vorgabe wurde die aus Abschnitt 2.2.3 bekannte Teillogik ∀CTL definiert. Das Resultat, dass die Simulation mit sich bringt, ist also das gewünschte Schlussfolgern von $\mathcal{T}' \models \Phi$ auf $\mathcal{T} \models \Phi$ mit der Einschränkung, dass $\Phi \in \forall\text{CTL}$ gelten muss.

2.3.2 Überapproximation

Will man nun also prüfen, ob eine Formel Φ vom Transitionssystem $\mathcal{T}$ erfüllt wird, das Transitionssystem sich aber aufgrund seiner Größe der Untersuchung widersetzt, so kann ein abstrakter Zustandsraum $\mathcal{T}'$ gebildet werden, dessen Makrozustände Gruppierungen von Zuständen von $\mathcal{T}$ repräsentieren und von $\mathcal{T}' \models \Phi$ auf $\mathcal{T} \models \Phi$ schließen. Je größer die Gruppierungen gewählt werden, desto kleiner fällt der Zustandsraum $\mathcal{T}'$ aus. Solche Gruppierungen lassen sich aber nicht groß und exakt zugleich wählen, weshalb Makrozustände in der Regel mehr Zustände repräsentieren als im konkreten Zustandsraum $\mathcal{T}$ erreichbar sind. Insbesondere folgt aus der Zusammenfassung der Zustände aus $\mathcal{T}$, dass in $\mathcal{T}'$ Transitionen enthalten sind, die in $\mathcal{T}$ nicht existieren.

Daraus resultiert, dass der Umkehrschluss $\mathcal{T}' \not\models \Phi \Rightarrow \mathcal{T} \not\models \Phi$ im Allgemeinen nicht gilt. Abbildung 2.3 verdeutlicht diesen Tatbestand. Hier ist das rechte Transitionssystem $\mathcal{T}'$ eine mögliche Abstraktion des konkreten Transitionssystems $\mathcal{T}$ der linken Seite. Die Zustände s_0, s_1, s_3 werden vom Zustand t_0 simuliert, während der Zustand s_2 vom Zustand t_1 repräsentiert wird. Man kann leicht erkennen, dass $\mathcal{T}'$ das Verhalten von $\mathcal{T}$ insofern überapproximiert, dass es in $\mathcal{T}'$ einen unendlichen Pfad gibt auf dem immer a gilt, während im konkreten Zustandsraum ein solcher nicht existiert.

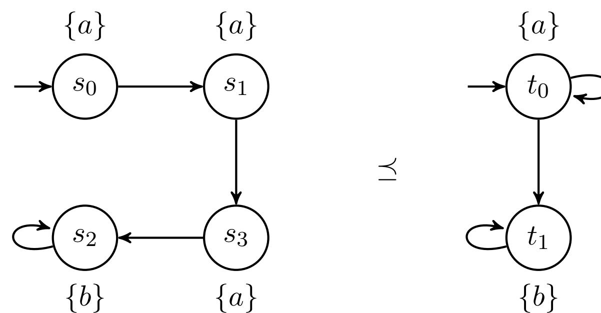

Abbildung 2.3: Konkretes Transitionssystem $\mathcal{T}$ (links) und abstraktes $\mathcal{T}'$ (rechts)

Ergibt sich also dass $\mathcal{T}' \not\models \Phi$ gilt, so liegt dies entweder daran, dass tatsächlich $\mathcal{T} \not\models \Phi$ gilt oder daran, dass das abstrakte Transitionssystem $\mathcal{T}'$ Pfade enthält, die die Spezifikation verletzen, diese Pfade aber im konkreten Transitionssystem nicht möglich sind, da sie das Resultat einer Überapproximation sind. Gegenbeispiele die

nur im abstrakten Transitionssystem möglich sind nennen wir deshalb *scheinbare*[2] Gegenbeispiele [CGJ+00a].

Es wird deutlich, dass die Wahl der Abstraktion von großer Bedeutung für die Größe des zu überprüfenden Zustandsraums ist und dass das Definieren einer derartigen Abstraktion eine zentrale Aufgabe ist. Dies kann im Grunde auf zwei Arten geschehen. Es kann entweder a priori eine Abstraktion festgelegt werden, welche z. B. aus einer vorangehenden statischen Analyse resultiert oder dynamisch während des Verifikationsvorgangs, konstruiert werden. Letzterer Ansatz ist entsprechend flexibler, da er iterativ verläuft und eine Abstraktion bei Feststellung einer problematischen Überapproximation gleich bei der Konstruktion angepasst wird. Die in dieser Ausarbeitung entwickelten Abstraktionen setzen auf die *Gegenbeispiel-geleitete Abstraktionsverfeierung*, die ein Repräsentant des zuletzt genannten Verfahrens ist und im folgenden Abschnitt vorgestellt wird.

2.3.3 Gegenbeispiel-geleitete Abstraktionsverfeinerung

Counterexample-Guided Abstraction Refinement[3] (CEGAR) [CGJ+00a] ist ein iteratives Verfahren, welches versucht Abstraktionen so grob wie möglich, aber so fein wie nötig zu wählen. Der Kern des Verfahrens spiegelt sich im folgenden, in [Kur94] ähnlich beschriebenen, *Refinement-Loop* wider:

1. Es wird eine initiale Abstraktion $\mathcal{R}$ gebildet, welche die Bedingungen an eine Simulationsrelation erfüllt und der abstrakte Zustandsraum $\mathcal{T}'$ entsprechend der Abstraktion aus $\mathcal{T}$ abgeleitet.

2. Es wird geprüft, ob $\mathcal{T}' \vDash \Phi$ gilt. Ist dies der Fall, impliziert dies $\mathcal{T} \vDash \Phi$ und der Algorithmus terminiert. Gilt allerdings $\mathcal{T}' \nvDash \Phi$, wird ein Gegenbeispiel ausgegeben. Ist dieses ein echtes Gegenbeispiel, welches sich auch im konkreten Zustandsraum reproduzieren lässt, gilt $\mathcal{T} \nvDash \Phi$ und der Algorithmus terminiert. Ist das Gegenbeispiel aber ein scheinbares, geht es im nächsten Schritt weiter.

3. Die Untersuchung des scheinbaren Gegenbeispiels liefert eine neue, feinere Abstraktion, die weniger Zustände aus $\mathcal{T}$ subsumiert. Es wird wieder nun bei Schritt 2 angesetzt.

Durch die stetige Verfeinerung konvergiert $\mathcal{T}'$ immer mehr gegen $\mathcal{T}$ und wird im schlimmsten Fall auch zu diesem. Dementsprechend ist die Terminierung dieses Algorithmus im Allgemeinen gewährleistet.

[2]urspr. spurious, engl. unecht
[3]engl. Gegenbeispiel-geleitete Abstraktionsverfeinerung

2.3.4 Relevante Abstraktionen

Die in dieser Arbeit entwickelten Abstraktionen setzen das Verständnis spezieller Abstraktionen voraus. Daher werden in diesem Abschnitt die *Intervalldomäne* [CC77] zur Abstraktion von expliziten Zahlenwerten und die *Booleschen Programme* [BR00b] zur Abstraktion von konkreten Variablenwerten und des Kontrollflusses dargestellt.

Intervalldomäne

Oft ist es der Fall, dass man beim Model-Checken nicht an dem speziellen Wert einer Variable interessiert ist, sondern nur daran, ob sie in einem bestimmten Bereich liegt. In solchen Fällen wird auf die Intervalldomäne zurückgegriffen, welche von den expliziten Werten einer Variable abstrahiert und stattdessen das Intervall, in dem die expliziten Werte liegen, symbolisch repräsentiert. Betrachten wir dazu Beispielprogramm 1 in Pseudocode:

Algorithm 1 Clamp-Funktion

```
1: procedure CLAMP(a)
2:     if a < 127 then
3:         res ← a
4:     else
5:         res ← 127
6:     end if
7:     return res
8: end procedure
```

Die dargestellte Funktion ist ein Spezialfall der aus dem Bereich der Computergrafik bekannten Clamp-Funktion. Diese Funktion bekommt eine Eingabe und gibt diese unverändert zurück, solange der Wert 127 unterschritten wurde. Liegt die Eingabe allerdings über diesem Wert, so wird sie auf 127 geklemmt[4] und so zurückgegeben.

Angenommen die Variable a des Beispielprogramms könnte aufgrund ihres Datentyps den Wertebereich [0,255] annehmen und der Model-Checker würde Zustände als speicherschonende Gegenüberstellung von Ein- und Ausgabe repräsentieren. Dann würde sich ohne Anwendung der Intervalldomäne das in Abbildung 2.4 links dargestellte Transitionssystem ergeben. Abgesehen vom Startzustand, welcher als Hilfskonstruktion verwendet wird, werden 256 Zustände benötigt um das Programm zu modellieren. Verwendet man allerdings die Intervalldomäne, so kann man allein mit der Partitionierung des Wertebereichs von a auf die Intervalle [0,126] und [127,255], wie in Abbildung 2.4 dargestellt, bereits Aussagen über den Programmfluss und das Ergebnis machen.

[4] clamp, engl. festklemmen

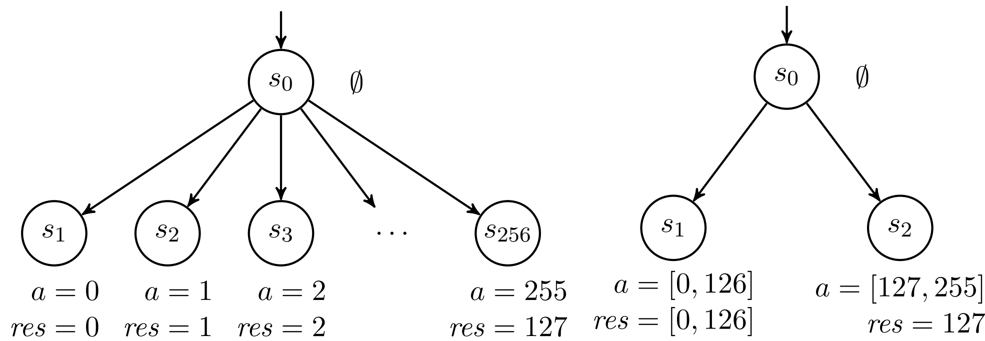

Abbildung 2.4: Transitionssystem ohne (links) und mit Intervalldomäne (ohne)

Werden genauere Informationen über den Wert von a benötigt, wie z.B. bei der Prüfung von $\Phi := \forall\Box(a = 0)$, wo abgesehen von der Information, ob a im Intervall [0,126] liegt, auch relevant ist, ob $a = 0$ gilt, so genügt es feinere Intervalle zu verwenden. In diesem Fall müssen dazu lediglich statt dem Intervall [0,126] die Intervalle [0,0] und [1,126] verwendet werden, um ein Transitionssystem zu konstruieren, in welchem die in Φ vorkommenden atomaren Aussagen Zuständen zuordnet werden können. Ohne Verfeinerung des Intervalls könnte man hier nicht sagen, ob $s_1 \models (a = 0)$ gilt, da a einen beliebigen Wert aus dem Bereich [0,126] annehmen könnte.

Boolesche Programme

Eine weitere für diese Arbeit relevante Abstraktionstechnik ist die Verwendung des Modells für *Boolesche Programme* [BR00b]. Boolesche Programme sind Programme, deren Variablen alle vom Typ *Boolean* sind und somit nur die zwei Wahrheitswerte *wahr* und *falsch* annehmen können. Die Idee hinter der Verwendung eines solchen Modells istt den gesamten Kontrollfluss eines Programms durch einige wenige Variablen mit einem kleinem Wertebereich zu beschreiben.

Diese Technik ist insbesondere für die Überprüfung von Formeln der Form $\Phi := \forall\Box\Psi$, wobei Ψ sich lediglich aus Booleschen Operationen auf atomaren Aussagen zusammensetzt, geeignet, da die Überprüfung einer solchen Formel sich auf die Erreichbarkeit eines Zustands s, welcher Ψ nicht erfüllt, reduzieren lässt. Es bleibt dann lediglich zu prüfen, ob es eine Belegung der Booleschen Variablen gibt, welche den Kontrollfluss zu diesem problematischen Zustand führt. Überführt man beispielsweise das Programm 1 aus dem vorherigen Abschnitt in ein Boolesches Programm, erhält man als ein mögliches Ergebnis das Pseudocode-Programm 2.

Algorithm 2 Clamp-Funktion als Boolesches Programm

```
1: procedure CLAMP(a)
2:     if aLessThan127 then
3:         res127 ← false
4:     else
5:         res127 ← true
6:     end if
7:     return res127
8: end procedure
```

Es handelt sich hierbei um nur eine von vielen möglichen Überführungsmöglichkeiten des Programms. Die in Listing 2 dargestelle Ausprägung ergibt sich, wenn man darauf abzielt, die Invariante $\Phi := \forall\Box(res = 127)$ zu überprüfen. Die Variable *aLessThan127* repräsentiert hier also die Möglichkeit den Kontrollfluss in den *then*- oder *else*-Zweig zu leiten, während die Variable *res127* so gewählt wird, dass sie die zu prüfende Semantik widerspiegelt. Die Prüfung der Gültigkeit von Φ kann im gegebenen Fall entsprechend auf die Prüfung der Erreichbarkeit von Zeile 3 zurückgeführt werden.

Hier wird bereits deutlich, dass diese Methode für Model-Checker beschrieben wurde, die Zustände innerhalb eines Programmdurchlaufs – genauer gesagt für jede Anweisung des Programms – erzeugen. Im Falle von SPSen bietet es sich an die Zustände, wie sich im Kapitel 2.4 noch zeigen wird, wie in Abbildung 2.4 als Gegenüberstellung der Ein- und Ausgangswerte eines Programmdurchlaufs darzustellen.

Des Weiteren fällt auf, dass bei der Überführung in ein Boolesches Programm Semantik verloren geht. In unserem Beispielprogramm 2 sieht man zum Beispiel nicht mehr, dass *res* nur dann ungleich 127 ist, wenn $a < 127$ ist. Im Booleschen Programm lässt sich dies höchstens über den Variablennamen erahnen. Diese Abstraktion ist folglich ein geeigneter Kandidat für die Einbindung im CEGAR refinement loop.

2.4 Arcade.PLC

ARCADE ist eine am Lehrstuhl 11[5] für Informatik an der RWTH entwickelte Software zur Verifizierung von eingebetteten Systemen durch Model-Checking und statische Analysen. Teil davon ist ARCADE.PLC, welches speziell auf die Verifizierung von SPSen ausgelegt ist.

ARCADE.PLC unterstützt dazu die Erzeugung des Zustandsraums aus Programmcode, so dass SPS-Programme ohne weitere Modifikation auf Spezifikationen überprüft werden können. Der Model-Checker von ARCADE verfolgt im Gegensatz zum symbo-

[5]Software für eingebettete Systeme

lischen Model-Checking [BCM+90] bei dem die Interaktion mit dem Zustandsraum über die Manipulation von sogenannten *shared ROBDDs* [6] [Bry86] geschieht, den Ansatz der expliziten Bildung der Zustände. Entsprechend implementiert Arcade.PLC Abstraktionen wie z. B. die in Abschnitt 2.3.4 beschriebene Intervalldomäne und verwendet im Kern den CEGAR-refinement loop, um mit der Zustandsexplosion umzugehen. Wie aus Abschnitt 2.2.3 bekannt ist, können hinsichtlich der Überapproximation von Abstraktionen prinzipiell auch nur in ∀CTL beschriebene Spezifikationen überprüft werden. Aufgrund des ∀CTL-∃CTL-Dualismus lassen sich allerdings auch ∃CTL-Formeln überprüfen, da die Negation einer solchen eine ∀CTL-Formeln bildet.

Auf die Struktur des Model-Checking-Vorgangs, die all dies ermöglicht, wird im folgenden Abschnitt eingegangen.

2.4.1 Grundlegender Ablauf

Der grundlegende Ablauf der Überprüfung einer Spezifikation in Arcade.PLC wird in Abbildung 2.5 schematisch dargestellt.

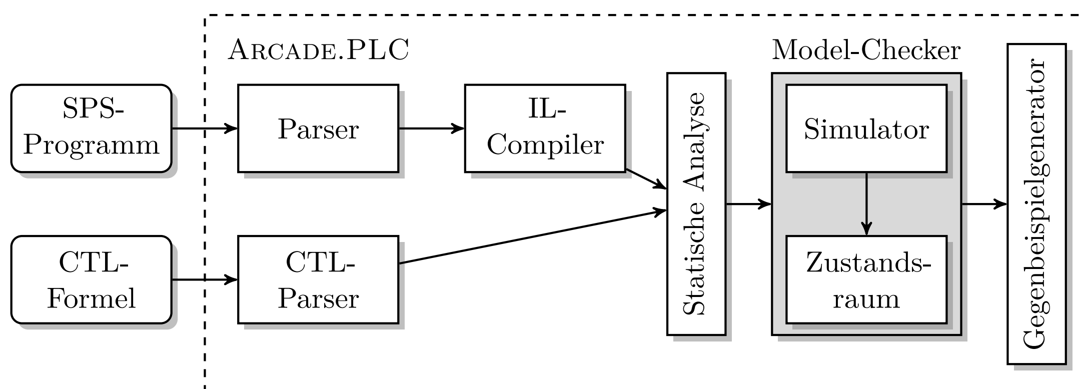

Abbildung 2.5: Schematischer Ablauf einer Überprüfung

Als Eingaben dienen Arcade.PLC eine Spezifikation in Form einer CTL-Formel und ein SPS-Programm. Wie schon in Abschnitt 2.1.2 angesprochen, können SPSen in mehreren Sprachen programmiert werden. Um eine einheitliche Basis für Untersuchungen zu haben, werden diese Sprachen zuerst geparst und schließlich in eine *intermediate language*[7] (IL) kompiliert. Derzeit ist Arcade.PLC dazu in der Lage die IEC 61131-Programmiersprachen *Strukturierter Text*, *Anweisungsliste* und den proprietären STEP 7 Dialekt *Statement List* zu parsen.

[6]shared reduced ordered binary decision diagram, engl. mehrfach genutztes , reduziertes, geordnetes, binäres Entscheidungsdiagramm

[7]intermediate language, engl. Zwischensprache

Mit dem IL-Code und der übersetzten CTL-Formel wird nun der initiale Zustand des Simulators eingestellt und ausgewählte statische Analysen durchgeführt. Diese können die Kontrollflussgraphen der Verwendeten POEs und Funktionen generieren, Invarianten erkennen oder auch unbenutzte Variablen bestimmen und sich nicht benötigter Teile mittels *Slicing*[Wei81] entledigen.

Anschließend beginnt der eigentliche Model-Checking-Prozess, in dem der abstrakte Zustandsraum im CEGAR refinement loop aufgebaut wird und zeitgleich die Gültigkeit der zu prüfenden Formel untersucht wird. Im Falle der Verletzung einer Formel wird aus dem zu diesem Zeitpunkt aufgebauten abstrakten Zustandsraum vom Gegenbeispielgenerator ein konkretes für den Anwender aussagekräftiges Gegenbeispiel generiert.

2.4.2 Erzeugung des Zustandsraums

Wie in Abschnitt 2.1.1 beschrieben, sind Zwischenzustände, die während des Programmdurchlaufs angenommen werden, bei einer SPS nach außen hin nicht sichtbar. Entsprechend beziehen sich die an eine SPS definierbaren Spezifikationen stets auf die Werte am Ende eines Zyklus. Würde man im Zustandsraum auch die inkonsistenten Zwischenzustände modellieren, könnten diese für eine Verletzung der Spezifikation sorgen, obwohl die Spezifikation an den nach außen beobachtbaren Zuständen erfüllt sein mag und so falsche Ergebnisse produzieren. Um derartige Komplikationen zu vermeiden und die Zyklusorientiertheit einer SPS auszunutzen, werden Zustände beim SPS-Model-Checking ähnlich zu Abbildung 2.4 als Tupel aus Ein-, Ausgängen und den nichttemporären Variablen dargestellt.

Beginnend im Startzustand, in dem alle Variablen der simulierten SPS auf 0 gesetzt sind, wird der Zustandsraum aufgebaut, indem alle möglichen Eingangsbelegungen simuliert und die sich jeweils am Zyklusende ergebenden Zustände als Nachfolger des Startzustands eingetragen werden. Diese Arbeitsweise setzt sich entsprechend weiter fort, wobei zu Beginn eines Zyklus stets alle Ein- und Ausgänge zurückgesetzt werden, um das Verhalten einer SPS nachzuahmen. Man beachte allerdings, dass bei diesem Ansatz eine 32-Bit Variable mit 2^{32} möglichen Belegungen, ohne jegliche Abstraktionen, für die Erzeugung von mindestens entsprechend vielen Nachfolgern verantwortlich ist. Dieser Problematik versucht Arcade.PLC durch die Verwendung verschiedener Abstraktionen, wie z. B. der in Abschnitt 2.3.4 beschriebenen Intervalldomäne zu begegnen.

2.4.3 Abstraktionsverfeinerung der Variablen

Dieser Abschnitt beleuchtet den in [BBK10] beschriebenen Algorithmus, den Arcade.PLC verwendet um Abstraktionen für die in einem Programm verwendeten

Variablen abzuleiten. Dazu werden zunächst *Constraints*[8] auf Variablen definiert, die beschreiben, welche abstrakten Werte eine Variable an einem Zeitpunkt in der Ausführung des Programms haben kann. Aus diesen Constraints wird dann mittels eines *Constraint Solvers* abgeleitet, welche Werte die Variablen zu Beginn eines Zyklus gehabt haben mussten. Dabei wird beachtet, dass der Kontrollfluss stets deterministisch bleibt, da wie in Abschnitt 2.4.2 beschrieben, die Zustände stets Tupel aus Ein-, Ausgängen und nichttemporären Variablen sind und keine Zwischenzustände beschrieben werden sollen. Schließlich wird der Constraint Solver innerhalb der Abstraktionsverfeinerung verwendet, um die abstrakten Eingangsbelegung für die Nachfolgerbildung festzulegen.

Constraint Solver

[BBK10] beschreibt ein Constraint $cs_f(v)$ als eine Bedingung f an einen abstrakten Wert v. Ein Constraint ist erfüllt, wenn die Menge der Werte, die v repräsentiert, *konsistent* unter f ist. In Arcade.PLC wird zwischen folgenden Constraints unterschieden:

1. Das Constraint $cs_{sing}(v)$ ist konsistent, wenn v ein konkreter Wert ist.

2. Die Vergleichs-Constraints $cs_{\bowtie c}(v), \bowtie \in \{=, \neq, <, \leq, >, \geq\}$ ist konsistent, wenn $\forall_{x,y \in v} x \bowtie c \Leftrightarrow y \bowtie c$ gilt, wobei c eine Konstante ist.

3. Das Bitmasken-Constraint $cs_{\&c}(v)$ ist konsistent, wenn $\forall_{x,y \in v}\ x \& c = y \& c$ gilt, wobei & für das bitweise UND steht und c auch hier eine Konstante ist.

Es ist ersichtlich, dass wenn man ein Constraint und eine Variable hat, man dieser Variable leicht einen abstrakten Wert zuweisen kann, der das Constraint erfüllt. Zu diesem Zweck verwendet Arcade.PLC *Splitter*, welche für eine Variable v der Domäne d und einem Constraint $cs_f(v)$ die Menge der Abstrakten Werte a_i aufzählt, sodass diese konsistent unter $cs_f(d)$ sind und die Domäne so grob wie möglich partitionieren. Ein Splitter gibt also die kleinste Menge von abstrakten Werten an, die konsistent bzgl. des Constraints sind. So würde z. B. die Variable v des Typs USINT (Wertebereich [0,255]) und das Constraint $cs_{<127}(v)$ dazu führen, dass der Splitter die konsistenten abstrakten Werte [0,126] und [127,255] ausgeben würde. Dies entspricht genau dem, was bereits in Abschnitt 2.3.4 als Beispiel aufgeführt wurde. Ungeklärt blieb bisher, wie der Constraint Solver von Arcade.PLC Constraints ableitet. Dies wird in [BBK10] im Detail beschrieben – wir wollen uns hier jedoch nur auf die Grundidee konzentrieren.

[8]constraint, engl. Zwangsbedingung

Anweisungsliste	SSA	Semantik der Anweisungen
LD input0	$acc^{(0)} := input_0^{(0)}$	Wert von input0 in Akkumulator laden
ADD 50	$acc^{(1)} := acc^{(0)} + 50$	50 zum Akkumulatorwert addieren
GT 100	$acc^{(2)} := acc^{(1)} > 100$	Ist der Akkumulatorwert größer als 100? Ergebnis wird in Akkumulator geschrieben
JMPC Label	$\text{guard}\Big(cs_{sing}(acc^{(2)})\Big)$	Springe zu Label, wenn Akkumulatorwert *true* ist. Überspringe Anweisung sonst.
...	...	...

Tabelle 2.1: Programmrepräsentation in SSA

Der Constraint Solver arbeitet intern auf einer *static single assignment*-Darstellung [CFR⁺91] des Programmcodes, bei der alle Anweisungen als Operationen zwischen einem Akkumulator und den Variablen repräsentiert werden. Tabelle 2.1 veranschaulicht diese Darstellung exemplarisch an einem Programmausschnitt in der assemblerartigen Programmiersprache Anweisungsliste [Int03], aus welcher die SSA-Darstellung besonders leicht abzuleiten ist.

Der dargestellte Code beschreibt den typischen Fall eines *bedingten Sprungs*, bei dem der Wahrheitsgehalt der Bedingung von vorherigen Operationen abhängig ist, da diese den Akkumulator manipulieren. Wie bereits vermerkt, muss der Kontrollfluss durch das Programm bei jeder Eingangsbelegung deterministisch sein. Entsprechend wird in ARCADE.PLC das Constraint cs_{sing} jedes mal an den Akkumulator acc zum Zeitpunkt eines Sprungs gestellt, um ein eindeutiges Sprungziel zu gewährleisten. Welche Auswirkungen die Anweisungen zwischen dem Programmstart und dem Sprung haben, kann nun aus der SSA-Darstellung abgeleitet werden.

Das Auflösen eines Constraints geschieht über eine induktiv aufgebaute Transformation, die für jede Kombination von jeweils zwei Constraints definiert ist und so iterativ von der aktuellen bis zur ersten Anweisung des Programms vorgeht [BBK10]. Der Operator $\vdash$ steht hierbei für einen Transformationsschritt.

$$\begin{aligned}
cs_{sing}\left(acc^{(2)}\right) &\vdash cs_{sing}\left(acc^{(1)} > 100\right) \\
&\vdash cs_{>100}\left(acc^{(1)}\right) \\
&\vdash cs_{>100}\left(acc^{(0)} + 50\right) \\
&\vdash cs_{>100-50}\left(acc^{(0)}\right) \\
&\vdash cs_{>50}\left(input_0^{(0)}\right)
\end{aligned}$$

Zu Beginn gilt es $cs_{sing}(acc^{(2)})$ aufzulösen, indem der Akkumulator zum Zeitpunkt 2, mit $acc^{(1)} > 100$, als Ergebnis einer Operation mit dem Akkumulator zum Zeitpunkt 1 dargestellt wird. Das resultierende Constraint $cs_{sing}\left(acc^{(1)} > 100\right)$ bezieht sich nun nicht mehr auf den Akkumulator allein, sondern auf das Ergebnis von $acc^{(1)} > 100$. In solchen Fällen wird das Constraint gemäß der induktiven Transformation umgeschrieben, sodass sich wieder ein Constraint ergibt, welches sich allein auf den Akkumulator bezieht. Der Vorgang wird wiederholt, bis das Constraint sich nur noch auf den Akkumulator zum Zeitpunkt 0 bezieht.

Die oben dargestellte Ableitung liefert nach diesem Verfahren das Constraint $cs_{>50}(input_0^{(0)})$. Dieses Constraint wird erfüllt, wenn *input*0 einen Wert aus dem Bereich [51,255] hat, was auch korrekt ist, da nur wenn der Wert größer als 50 ist, man nach einer Addition von weiteren 50, einen Wert, der größer als 100 ist, im Akkumulator stehen hat.

Verfeinerung der Eingangsvariablen

Die Verfeinerung der Eingangsvariablen ist in Arcade.PLC direkt in die Bildung der Folgezustände eingewoben und wird in einem CEGAR-ähnlichen Ansatz durchgeführt. Dabei werden nichttemporäre Variablen vorerst nicht betrachtet und zunächst als konkret angenommen. Sollen nun also die Nachfolger eines Zustands gebildet werden, wird wie folgt vorgegangen.

1. Es wird ein Stack für Splitter angelegt und der Splitter, der allen Variablen ihren gröbsten abstrakten Wert zuweist, draufgelegt.

2. Der oberste Splitter wird verwendet, um die Eingänge mit Werten zu belegen.

3. Das Programm wird mit den gegebenen Eingängen simuliert bis eine Stelle erreicht wird, an der die Simulation aufgrund von Nichtdeterminismus abgebrochen werden muss. Hier wird der Constraint Solver gestartet um ein entsprechendes Constraint $cs_{sing}(acc^{(i)})$ aufzulösen und aus dem Ergebnis einen neuen Splitter zu erzeugen, welcher auf den Stack kommt. Musste kein Nichtdeterminismus aufgelöst werden, geht es mit Schritt 3 weiter – ansonsten wird Schritt 2 wiederholt.

4. Die atomaren Aussagen werden ausgewertet. Kann ein Wahrheitswert aufgrund von Nichtdeterminismus nicht eindeutig bestimmt werden, wird auch hier ein entsprechendes Constraint festgelegt und aus dem Resultat des Constraint Solvers ein neuer Splitter erzeugt, der auf den Stack kommt. Schritt 2 wird in dem Fall wiederholt.

5. Der so erzeugte Zustand wird im Zustandsraum gespeichert.

6. Der oberste Splitter des Stacks wird nun auf den nächsten abstrakten Wert gestellt und Schritt 2 wiederholt. Wurden im obersten Splitter alle abstrakten Werte einmal zugewiesen, wird er vom Stack gelöscht. Ist der Stack leer, wurden alle Nachfolgezustände erzeugt.

Verfeinerung des nichttemporären Variablen

Für die Erweiterung des Konzepts um die Verfeinerung von nichttemporären Variablen, führt [BBK10] in Arcade.PLC *Lemmata* ein, welche Constraints festhalten, die an nichttemporäre Variablen gestellt werden und bei einem Neuaufbau des Zustandsraums beachtet werden. Dabei wird im Grunde der im vorherigen Abschnitt vorgestellte Algorithmus ausgeführt und im Falle der Erzeugung eines Constraints an eine nichttemporäre Variable, dieses in einem Lemma gespeichert. Der Zustandsraum wird von nun an stets unter Berücksichtigung dieses neuen Constraints aufgebaut. Auf die genauen Details wird hier nicht weiter eingegangen – für diese sei auf [BBK10] verwiesen.

3 Abstraktionen

Ein generelles Problem von ARCADE.PLC, so wie es beschrieben wurde, ist die Tatsache, dass viele Ausdrücke ausgewertet werden, deren genaues Ergebnis nicht von Relevanz für die zu überprüfende Spezifikation ist. Dieses Kapitel beleuchtet zwei verschiedene Ausprägungen dieser Problematik und stellt Abstraktionen vor, die zur Lösung dieses Problems beitragen.

3.1 Boolesche Abstraktion

Im diesem Abschnitt wird ein Problem von ARCADE.PLC vorgestellt, welches aus der Verfeinerung von Constraints für Eingangsvariablen resultiert und bei Ausdrücken mit mehreren Variablen auftritt. Im Folgenden wird das Problem erst anhand eines Beispielprogramms beschrieben und die entwickelte Lösung im Anschluss vorgestellt.

3.1.1 Motivation

Um die Notwendigkeit der in diesem Abschnitt beschriebenen Abstraktion deutlich zu machen, betrachten wir zunächst das in Listing 3.1 dargestellte, in Structured Text geschriebene SPS-Programm.

```
1  FUNCTION_BLOCK PROBLEM_1
2  [...]
3  IF (A + B + C< 127) THEN
4      RES:=A+B+C;
5  ELSE
6      RES:=127;
7  END_IF;
8
9  IF (A > 0) THEN
10     RES2:=A;
11 ELSE
12     RES2:=B;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 3.1: Clamp-Funktion mit drei Parametern und Zusatz

Zwecks Übersichtlichkeit beinhaltet das Listing die Variablendeklaration nicht, was durch die Punkte in Zeile 2 angedeutet wird. Das Programm besitzt allerdings drei Eingangsvariablen A, B, C und zwei Ausgänge *RES* und *RES2*, die alle einen Wertebereich von 0 bis 255 haben.

Wie man sieht, handelt es sich bei dem Programm lediglich um eine modifizierte Version der aus Abschnitt 2.3.4 bekannten Clamp-Funktion. Ist die Summe der Eingangsvariablen kleiner als 127, so wird diese in den Ausgang *RES* geschrieben. Beträgt die Summe mindestens 127, so wird stattdessen der auf 127 geklemmte Wert in die Ausgabe *RES* geleitet. In der zweiten Hälfte des Programms wird in Abhängigkeit von A entweder A oder B an den Ausgang *RES2* weitergeleitet.

Aus Abschnitt 2.4.3 ist bekannt, dass ARCADE.PLC nicht alle Zustände explizit darstellt, sondern abstrakte Zustände verwendet, bei denen die Eingangsbelegung so weit verfeinert ist, dass sowohl der Kontrollfluss durch das Programm als auch der Wahrheitsgehalt der zu prüfenden Formel eindeutig ist. Versucht man nun allerdings das SPS-Programm aus Listing 3.1 zu überprüfen, wird ein Problem dieses Verfahrens deutlich.

Bei der Simulation des Programms versucht ARCADE.PLC in Zeile 3 die Intervalle für die Variablen A, B und C so zu bestimmen, dass der Kontrollfluss eindeutig ist. Dies wäre auch kein Problem, wenn die Bedingung die folgende Form hätte: $A < c$, wobei $c = const$ sei. In dem Fall würde die Verfeinerung des ursprünglichen Intervalls [0,255] für A lediglich in der Aufspaltung auf die beiden Intervalle [0, c-1] und [c, 255] resultieren und für beide möglichen, abstrakten Ausprägungen von A einen eindeutigen Kontrollfluss garantieren. Aber bereits bei der Verwendung von zwei Variablen in der Bedingung, kommt man bei Verwendung der Intervalldomäne um eine Zustandsexplosion nicht umher. Tabelle 3.1 deutet an, wie die Wertintervalle am Beispiel einer Bedingung $A + B < 127$ aufgespalten werden würden:

$A =$	[0,126]	[127,255]	[0,125]	[126,255]	[0,124]	[125,255]	...
$B =$	0	0	1	1	2	2	...

Tabelle 3.1: Bestimmung abstrakter Werte für A und B am Beispiel $A + B < 127$

Es wird deutlich, dass bei Bedingungen dieser Form nur der Wert einer Variablen als Intervall beschrieben werden kann und die anderen Variablen oftmals in ihre konkreten Werte aufgespalten werden müssen.

Ein derartiges Problem könnte durch die Verwendung einer anderen Abstraktionsdomäne gelöst werden. So könnte man zum Beispiel die Oktagondomäne [MNS01] verwenden, um Constraints der Form $\pm A \pm B \leq c$, wobei $c = const$ ist, zu repräsentieren. Allerdings würde man bereits bei einer anderen Operation wie der Multiplikation oder der Verwendung von weiteren Variablen auch mit anderen bekannten Abstraktionsdomänen wieder auf die Zustandsexplosion stoßen. Ebenso

würde die Verwendung eines anderen Datentypen wie z. B. *Float* oder *Pointer* dazu führen, dass eine Abstraktion nicht durchführbar wäre.

Will man nun z. B. überprüfen, ob die Spezifikation $\Phi := \forall\Box(\mathit{RES2} < 127)$ vom Programm aus Listing 3.1 erfüllt wird, so resultiert dies in Arcade.PLC in der Überprüfung eines über 100.000 Zustände großen Zustandraums. Bei einer minimalen Erhöhung des Wertebereiches der Eingangsvariablen oder der Erweiterung der Bedingung des If-Statements um eine weitere Variable bricht Arcade.PLC den Überprüfungsvorgang aufgrund der nicht handhabbaren Zustandsexplosion sogar ab. Dies ist insofern problematisch, als die Auswertung von $A + B + C < 127$ für die Überprüfung der Spezifikation überhaupt nicht relevant ist und diese Spezifikation auch ohne die Entwicklung einer für solche Bedingungen geeigneten Abstraktionsdomäne überprüfbar sein sollte. Ebenso sollte sich auch die Spezifikation $\Phi := \exists\bigcirc\exists\Diamond(\mathit{RES} \leq 127)$ ohne Wissen um das Ergebnis von $A + B + C < 127$ prüfen lassen.

3.1.2 Grundidee

Die Boolesche Abstraktion soll das im letzten Abschnitt beschriebene Problem lösen, indem sie die Bedingungen von If-Statements so abstrahiert, dass lediglich die möglichen Ausgänge der Prüfung dieser Bedingungen, d.h. ob der Then- oder der Else-Zweig gewählt wird, beschrieben werden. Sollte für die Auswertung einer Spezifikation der genaue Wert einer abstrahierten Bedingung benötigt werden, so soll die Abstraktion verfeinert werden und die entsprechende Bedingung wieder konkretisiert werden. Da diese Abstraktion im Grunde aus den für den Refinement-Loop charakteristischen Schritten besteht, wird diese Technik im CEGAR-Verfahren realisiert.

3.1.3 Initiale Abstraktion

Die initiale Abstraktion geht aus dem konkreten Programm hervor, indem man für jedes zu abstrahierende If-Statement eine Boolesche Eingangsvariable einführt, welche die konkrete Bedingung des If-Statements ersetzt. Da beim Aufbau des Zustandsraumes alle möglichen Eingangsbelegungen simuliert werden, werden durch die eingeführte Boolesche Variable alle möglichen Kontrollflüsse durch das If-Statement simuliert. Es handelt sich dabei um eine Überapproximation, da nun unabhängig davon, ob die ursprüngliche Bedingung erfüllt worden wäre, immer beide möglichen Ausgänge betrachtet werden.

Allerdings ist nicht jedes If-Statement für eine derartige Abstraktion geeignet. Einfache If-Statements, deren Bedingungen nur eine Variable haben, werden von Arcade.PLC bereits effektiv abstrahiert, weshalb eine Abstraktion solcher den Verifikationsvorgang eher behindern würde. Weitere If-Statements, die explizit ignoriert werden müssen, sind solche, die in Schleifen auftauchen. Taucht ein If-Statement in

einer Schleife auf, ist es sehr wahrscheinlich, dass die Bedingung dieses Statements nicht über alle Durchläufe konsistent ist, sondern vom Laufindex abhängig ist. Würde man ein solches Statement abstrahieren und die Bedingung durch eine Boolesche Variable ersetzen, würde man nicht alle Kontrollflusspfade, die das konkrete Programm haben kann, simulieren können. Die Boolesche Variable würde dafür sorgen, dass für alle Iterationen der Schleife der gleiche Zweig des If-Statements genommen werden würde. Das folgende Listing 3.2 veranschaulicht die initiale Abstraktion am Beispiel des aus Listing 3.1 bekannten Programms.

```
1  FUNCTION_BLOCK SOLUTION_1
2  [...]
3  IF (D) THEN
4      RES:=A+B+C;
5  ELSE
6      RES:=127;
7  END_IF;
8
9  IF (A > 0) THEN
10     RES2:=A;
11 ELSE
12     RES2:=B;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 3.2: Abstrahierte Clamp-Funktion mit drei Parametern und Zusatz

Hier wurde die konkrete Bedingung $A + B + C < 127$ aus Zeile 3 durch die neu eingeführte Boolesche Variable D ersetzt, während das If-Statement in Zeile 9 unverändert geblieben ist. Betrachten wir nun den Zustandsraum des zuvor problematischen Programms, welcher in Abbildung 3.1 nur schematisch abgebildet ist, da die Kanten zwischen den Zuständen s_1, s_2, s_3 und s_4 und die atomaren Aussagen an s_0 zwecks Transparenz in der Darstellung ignoriert wurden.

Wie man sieht, besteht der Zustandsraum nun prinzipiell aus nur vier Zuständen und dem Initialzustand. Da die Information über den im If-Statement gewählten Pfad symbolisch über die Boolesche Variable kodiert ist, würde sich der Zustandsraum selbst bei einer Änderung der Datentypen von A, B oder C nicht ändern.

Überprüft man nun die in Abschnitt 3.1.1 diskutierte problematische Formel $\Phi := \forall\Box(\mathit{RES2} < 127)$, so wird zunächst festgestellt, dass diese bereits vom Zustand s_1 verletzt wird und erhält ein lineares Gegenbeispiel. Damit kommen wir zum zweiten Schritt des CEGAR Refinement-Loops, in dem überprüft wird, ob ein Gegenbeispiel auch im konkreten Zustandsraum möglich ist oder es sich lediglich um ein scheinbares Gegenbeispiel handelt.

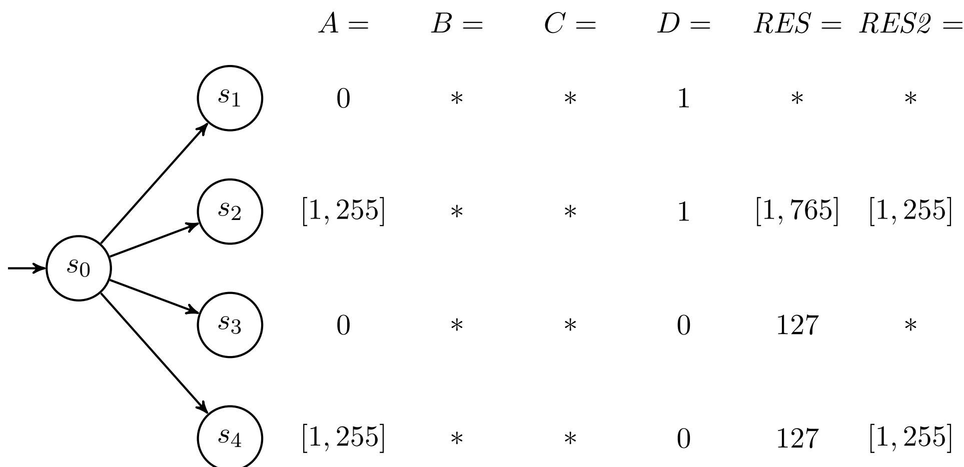

Abbildung 3.1: Abstraktes Transitionssystem $\mathcal{T}'$

3.1.4 Abstraktionsverfeinerung

Der klassische Verlauf des CEGAR Refinement-Loops sieht vor, dass in diesem Schritt die Realisierbarkeit eines ausgegebenen Gegenbeispiels überprüft wird und erst im nächsten Schritt eine feinere Abstraktion aus der Analyse des Gegenbeispiels abgeleitet wird. Die Abstraktionsverfeinerung der Booleschen Abstraktion weicht von diesem Aufbau allerdings ab, da sie den Schritt der Überprüfung der Realisierbarkeit mit dem der Konstruktion einer feineren Abstraktion kombiniert. Der Grundgedanke dahinter ist, dass das Finden des für das Gegenbeispiel verantwortlichen abstrahierten If-Statements schwer ist und daher erst verfeinert und anschließend geprüft wird, ob diese Verfeinerung Einfluss auf die Realisierbarkeit des Gegenbeispiels hat.

Wie bereits in Abschnitt 2.2.4 beschrieben, unterscheiden wir zwischen linearen und nichtlinearen Gegenbeispielen. Die folgenden Abschnitte beschreiben wie die Abstraktionsverfeinerung in beiden Fällen realisiert wird. Die Überprüfung, ob ein Gegenbeispiel im verfeinerten Zustandsraum noch existiert, wird aus dieser Beschreibung zwecks Übersichtlichkeit ausgelassen und in einem separaten, anschließenden Abschnitt dargestellt.

Analyse linearer Gegenbeispiele

Die Analyse eines linearen Gegenbeispiels und der daraus resultierenden Abstraktionsverfeinerung wird im Folgenden exemplarisch an dem in Abbildung 3.2 dargestellten abstrakten Gegenbeispiel beschrieben.

Das schematisch abgebildete Gegenbeispiel besteht aus vier Zuständen, die aus einem abstrakten Zustandsraum $\mathcal{T}'$ stammen, in welchem die Boolesche Abstraktion

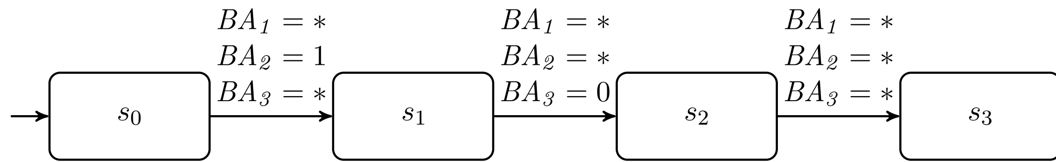

Abbildung 3.2: Lineares Gegenbeispiel aus $\mathcal{T}'$

dazu geführt hat, dass die Variablen BA_1, BA_2 und BA_3 eingeführt wurden, um die ursprünglichen Bedingungen von If-Statements zu ersetzen. Zur Anschauung wurde in der Abbildung auf atomare Aussagen, die nicht direkt aus der Abstraktion resultieren, verzichtet und die Eingangsbelegung, die bei der Simulation zum jeweils nächsten Folgezustand führt, über die entsprechende Transition geschrieben, obwohl diese prinzipiell als atomare Aussagen des Folgezustands zu verstehen ist.

Wie man sieht, wurde die Verletzung der überprüften Spezifikation erst in Zustand s_3 festgestellt, da dieser sonst nicht der letzte Zustand des Gegenbeispiels wäre. Des Weiteren ist zu beachten, dass die Belegung der Booleschen Variablen, bei der Frage der Erreichbarkeit von s_3 von s_2 aus, keine Rolle spielt. Die letzte entscheidende Verwendung einer Booleschen Variable tritt bei der Transition $s_1 \to s_2$ auf, bei der nur der Wert der Booleschen Variablen BA_3 relevant ist – die Belegung von BA_1 und BA_2 ändert nichts daran, dass der Pfad $s_1 \to s_2 \to s_3$ möglich ist. Die nächstliegende Vermutung ist also, dass die Abstraktion BA_3 zu der Verletzung geführt hat und entsprechend zurückgesetzt werden muss.

Wird statt BA_3 nun also wieder der ursprüngliche Ausdruck eingesetzt und überprüft, ob das Gegenbeispiel in diesem verfeinerten Programm realisierbar ist, so kann aus diesem Ergebnis darauf geschlossen werden, ob BA_3 das Gegenbeispiel verursacht hat. Stellt sich heraus, dass das Gegenbeispiel im verfeinerten Programm immer noch existiert, so resultierte es nicht aus der Abstraktion BA_3, da das Gegenbeispiel unabhängig von der Verwendung der Abstraktion existiert. Die Abstraktion wird in dem Fall entsprechend wieder verwendet und das Problem bei den restlichen Abstraktionen gesucht. Ist das Gegenbeispiel im verfeinerten Programm allerdings nicht realisierbar, so resultierte das Gegenbeispiel tatsächlich aus der Abstraktion BA_3 und war nur ein scheinbares Gegenbeispiel. Die Abstraktion wird in dem Fall entsprechend verfeinert gelassen und die Spezifikation wird erneut überprüft, ohne dass das scheinbare Gegenbeispiel nochmal auftauchen kann.

An diesem Beispiel wird der allgemeine Ablauf der Analyse eines linearen Gegenbeispiels deutlich:

1. Es wird vom letzten Zustand des Gegenbeispiels rückwärts nach einem Zustand gesucht, in dem die Belegung der Variablen relevant für die Erreichbarkeit des Problemzustands ist.

2. In diesem Zustand wird iterativ für jede Variable BA_i mit $BA_i \neq *$ wie oben beschrieben geprüft, ob diese für das scheinbare Gegenbeispiel verantwortlich ist und die entsprechende Abstraktion wieder konkretisiert werden soll oder ob sie keinen Einfluss darauf hat und die Abstraktion bestehen bleiben darf. Wurde die Abstraktion konkretisiert, wird die Überprüfung der Spezifikation erneut auf dem so verfeinerten Transitionssystem ausgeführt.

3. Stellt sich heraus, dass keine der Variablen verantwortlich für das Gegenbeispiel ist, handelt es sich um ein konkretes Gegenbeispiel und die Spezifikation ist tatsächlich verletzt.

Zusammenfassend werden also solange Abstraktionen an If-Statements rückgängig gemacht bis eindeutig gesagt werden kann, ob das Gegenbeispiel auch im konkreten Transitionssystem vorliegt. Die Reihenfolge und Menge der in Betracht zu ziehenden Variablen werden durch Ausnutzung der Struktur des Gegenbeispiels erlangt.

Analyse nichtlinearer Gegenbeispiele

Wird im Laufe einer Verifikation nun aber ein nichtlineares Gegenbeispiel ausgegeben, so kann man die für lineare Gegenbeispiele vorgestellte Methodik nicht mehr anwenden, da aufgrund der Nichtlinearität nicht mehr ersichtlich ist, in welcher Reihenfolge die Variablen zu Prüfen sind. Zusätzlich ist die Überprüfung, ob ein Gegenbeispiel in einem verfeinerten Zustandsraum realisierbar ist, in diesem Fall deutlich schwerer als im linearen Fall. Entsprechend wird hier heuristisch vorgegangen.

Hintergrund der verwendeten Heuristik ist Folgender. Wie aus Abschnitt 2.1.2 bekannt ist, bestehen SPS-Programme aus POEs und Funktionsblöcken, welche sich gegenseitig ohne Rekursion aufrufen. Das heißt, dass Abstraktionen im Hauptblock in größeren Überapproximationen resultieren als Abstraktionen in den Blöcken von Unteraufrufen.

Im Folgenden wird der Ablauf der Analyse des nichtlinearen Gegenbeispiels exemplarisch am *Call-Graph*[1] [Ryd79] aus Abbildung 3.3 beschrieben, welcher die Aufrufhierarchie des zu prüfenden Programms darstellt.

1. Der Call-Graph wird von der Haupt-POE aus in einer Breitensuche nach abstrahierten If-Statements durchsucht.

2. Werden in einer Tiefe Abstraktionen gefunden, so werden diese konkretisiert und die Überprüfung der Spezifikation erneut durchgeführt.

[1] call, engl. Aufruf

3. Werden keine Abstraktionen mehr verwendet, so handelt es sich bei dem Gegenbeispiel um ein konkretes Gegenbeispiel und die Spezifikation ist verletzt.

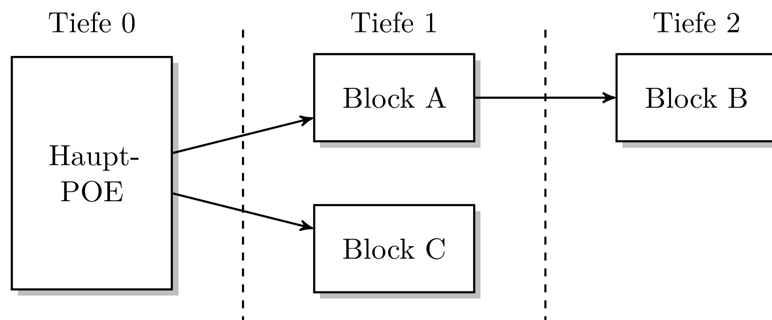

Abbildung 3.3: Call-Graph eines zu überprüfenden SPS-Programms

Zusammenfassend werden also Abstraktionen an If-Statements solange Tiefe für Tiefe rückgängig gemacht bis eindeutig gesagt werden kann, ob das Gegenbeispiel auch im konkreten Transitionssystem vorliegt.

Der allgemeine Ablauf der Analyse eines Gegenbeispiels ist nun zwar dargestellt, aber auf die Überprüfung der Realisierbarkeit eines Gegenbeispiels wurde bisher nicht eingegangen. Der folgende Abschnitt beschreibt das zugrunde liegende Verfahren.

Prüfung der Realisierbarkeit eines Gegenbeispiels

Die Überprüfung der Realisierbarkeit eines Gegenbeispiels aus einem abstrakten Zustandsraum $\mathcal{T}'$ in einem verfeinerten bzw. konkreten Zustandsraum $\mathcal{T}$ wird im Folgenden anhand der Abbildung 3.4 beschrieben. Die gestrichelt umrandeten Zustände stellen dabei den abstrakten Zustandsraum $\mathcal{T}'$ dar, welcher bereits aus Abbildung 3.2 bekannt ist, während die restlichen Zustände aus dem verfeinerten Zustandsraum $\mathcal{T}$ kommen.

Der Grundgedanke ist, dass sich das abstrakte Gegenbeispiel auch in Zuständen des verfeinerten Zustandsraums darstellbar ist, sofern es sich nicht um ein scheinbares Gegenbeispiel handelt. Dazu wird jeder abstrakte Zustand s durch einen verfeinerten Zustand t ausgetauscht, der zuvor in diesem abstrakten Zustand zusammengefasst wurde und die Erreichbarkeit des Problemzustands wahrt.

Aufgrund der Überapproximation existiert aber nicht jeder in einem abstrakten Zustand zusammengefasste Zustand auch im verfeinerten Zustandsraum. Die Entscheidung, ob ein Zustand t des verfeinerten Zustandsraums tatsächlich von einem Zustand s des abstrakten Zustandsraums repräsentiert wird, wird daher durch die Überprüfung, ob die abstrakten Werte in den verfeinerten Zuständen Teilmengen der abstrakten Werte in den abstrakten Zuständen sind, realisiert. Dies kann man sich am

Beispiel der Intervalldomäne leicht verdeutlichen. Ein Zustand t mit den verfeinerten Werten $A = [0, 127], B = 2$ würde z. B. vom Zustand s mit den abstrakten Werten $A = [0, 200], B = *$ repräsentiert werden können. In der Abbildung wird ein solcher Zusammenhang entsprechend so abgebildet, dass ein durch s repräsentierter Zustand t innerhalb dessen dargestellt wird.

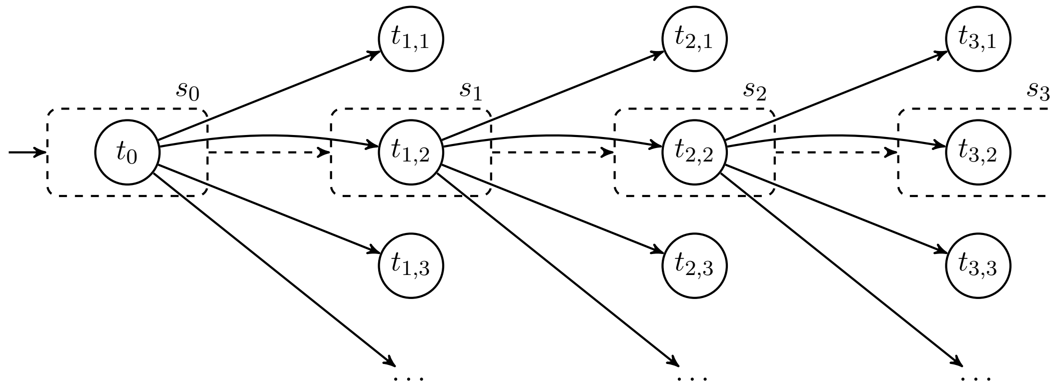

Abbildung 3.4: Überprüfung der Realisierbarkeit

Soll nun überprüft werden, ob das gestrichelt dargestellte abstrakte Gegenbeispiel in einem verfeinerten Zustandsraum möglich ist, wird daher folgendermaßen vorgegangen.

1. Es wird der eindeutige Startzustand t_0 des verfeinerten Zustandsraums erzeugt und sichergestellt, dass dieser vom Startzustand s_0 des abstrakten Gegenbeispiels repräsentiert wird.

2. In einer Breitensuche wird nun versucht einen Pfad durch den verfeinerten Zustandsraum zu finden, dessen Zustände $t_0, \dots t_n$ jeweils von den Zuständen $s_0, \dots s_n$ repräsentiert werden. Pfade die Zustände wie $t_{1,1}$ enthalten, welche offensichtlich nicht mehr im abstrakten Gegenbeispiel repräsentiert werden, werden direkt verworfen. In der Abbildung entspricht dieser Schritt zunächst der Erzeugung der Zustände $t_{1,i}$ und der Feststellung, dass nur der Nachfolger $t_{1,2}$ von s_1 repräsentiert wird. Die weiterführende Breitensuche führt zur Auffindung des Pfades $t_0 \to t_{1,2} \to t_{2,2} \to t_{3,2}$.

3. Wird das Gegenbeispiel komplett traversiert und der Pfad zum einer Verfeinerung des Problemzustands entsprechend im verfeinerten Zustandsraum konstruiert, ist die Realisierbarkeit des Gegenbeispiels im verfeinerten Zustandsraum gegeben. Dies entspricht dem abgebildeten Fall, da mit Auffinden des

Zustands $t_{3,2}$ das abstrakte Gegenbeispiel tatsächlich eine Ausprägung im verfeinerten Zustandsraum besitzt. Wird kein entsprechender Pfad im verfeinerten Zustandsraum gefunden, ist das Gegenbeispiel nicht realisierbar.

Zusammenfassend wird also das abstrakte Gegenbeispiel in einer Breitensuche traversiert, während parallel dazu versucht wird Zustände des verfeinerten Zustandsraums zu finden, welche durch die abstrakten Zustände zusammengefasst werden. Lässt sich das abstrakte Gegenbeispiel entsprechend im verfeinerten Zustandsraum konstruieren, ist die Realisierbarkeit nachgewiesen.

Hiermit schließen wir mit der Beschreibung der Booleschen Abstraktion ab, welche dafür sorgt, dass If-Statements erst dann ausgewertet werden müssen, wenn das Wissen um den konkreten Wert der Bedingung unerlässlich für die Überprüfung eines Gegenbeispiels ist und kommen zu einer anderen Ausprägung der Problematik der vorzeitigen Auswertung in Arcade.PLC.

3.2 Modulare Abstraktion

Im diesem Abschnitt wird ein Problem von Arcade.PLC vorgestellt, welches aus der Verfeinerung von Constraints bei der Auswertung von Aufrufen resultiert und dessen Lösung konzeptionell stark verwandt mit der Booleschen Abstraktion ist. Im Folgenden wird das Problem erst anhand eines Beispielprogramms illustriert und im Anschluss die entwickelte Lösung vorgestellt.

3.2.1 Motivation

Um die Notwendigkeit der in diesem Abschnitt beschriebenen Abstraktion deutlich zu machen, betrachten wir auch hier zunächst ein Beispielprogramm, welches das zu lösende Problem beinhaltet. Listing 3.3 beinhaltet ein in Structured Text geschriebenes SPS-Programm, welches große Ähnlichkeit zu Listing 3.1 aus der Motivation der Booleschen Abstraktion aufweist. Der wesentliche Unterschied ist hier jedoch, dass in der Bedingung statt der Addition nun eine nicht näher beschriebene Funktion aufgerufen wird, von der lediglich bekannt sei, dass der Rückgabewert im Bereich [0,255] liegt. Zwecks Übersichtlichkeit beinhaltet auch dieses Listing keine Variablendeklaration, was durch die Punkte in Zeile 2 angedeutet wird. Das Programm besitzt allerdings drei Eingangsvariablen A, B, C und zwei Ausgänge RES und $RES2$, die ebenfalls alle einen Wertebereich von 0 bis 255 haben.

Die Semantik des Programms ist simpel gehalten. Ist das Ergebnis des Aufrufs $anon(A, B, C)$ kleiner als 127, so wird die Summe von A, B und C in den Ausgang RES geschrieben. Beträgt das Ergebnis mindestens 127, so wird stattdessen der Wert 127 in die Ausgabe RES geleitet. In der zweite Hälfte des Programms wird in Abhängigkeit von A entweder A oder B an den Ausgang $RES2$ weitergeleitet.

```
1  FUNCTION_BLOCK PROBLEM_2
2  [...]
3  IF (anon(A,B,C) < 127) THEN
4      RES:=A+B+C;
5  ELSE
6      RES:=127;
7  END_IF;
8
9  IF (A > 0) THEN
10     RES2:=A;
11 ELSE
12     RES2:=B;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 3.3: Clamp-Funktion mit Aufruf in Bedingung

Wie schon in Abschnitt 2.4.3 beschrieben und in Abschnitt 3.1.1 aufgegriffen, wird in ARCADE.PLC bei der Bildung des abstrakten Zustandsraums die Eingangsbelegung so weit verfeinert bis sowohl der Kontrollfluss durch das Programm als auch der Wahrheitsgehalt der zu prüfenden Formel eindeutig ist. Versucht man nun allerdings das SPS-Programm aus Listing 3.3 zu Überprüfen, wird ein weiteres Problem dieses Verfahrens deutlich.

Man beachte zunächst, dass die Boolesche Abstraktion bei diesem If-Statement nicht eingesetzt werden könnte. Dies kommt daher, dass die Bedingung des If-Statements im IL-Code die Form $temp_1 < 127$ hat, welche sehr gut von ARCADE.PLC verarbeitet wird und nicht Boolesch abstrahiert werden sollte. Der Wert von $temp_1$ wird vor der Auswertung der Bedingung aus dem Ergebnis des Aufrufs von *anon* übernommen.

Man kann sich leicht vorstellen, dass *anon* die Summenfunktion sein könnte, so dass die Semantik des Programms wieder der des Listings 3.1 entsprechen würde und es zur selben Zustandsexplosion kommen würde wie sie in Abschnitt 3.1.1 beschrieben wurde. Ebenso könnte die Funktion aber auch weitere Unteraufrufe oder komplexe Ausdrücke beinhalten, die den Zustandsraum bis zur Unprüfbarkeit anwachsen lassen. Dies stellt sich als besonders hinderlich heraus, wenn man die aus der Motivation der Booleschen Abstraktion bekannten Formeln $\Phi := \forall\Box(RES2 < 127)$ oder $\Phi := \exists \bigcirc \exists\Diamond(RES \leq 127)$ prüfen will, da diese sich auch ohne Wissen um das Ergebnis von $anon(A, B, C) < 127$ prüfen lassen sollten, sich aber der Untersuchung durch die Auswertung einer entsprechend problematischen Funktion widersetzen. Das Problem beschränkt sich dabei nicht nur auf Funktionen in If-Statements, sondern generell auf Aufrufe im Programm, deren Auswertung zu einer Zustandsexplosion führt.

3.2.2 Grundidee

Die Modulare Abstraktion soll die im letzten Abschnitt dargestellte Problematik lösen, indem sie Aufrufe von Funktionen und POEs so abstrahiert, dass lediglich beschrieben wird in welchem Bereich das Ergebnis liegen kann und die Überprüfung einer Spezifikation vorerst mit einem abstrakten Rückgabewert ausgeführt wird, um die explizite Simulation des problematischen Programmteils zu unterbinden. Sollte für die Auswertung einer Spezifikation der genaue Wert eines abstrahierten Aufrufs benötigt werden, so soll die Abstraktion verfeinert werden und der entsprechende Aufruf wieder konkretisiert werden. Da auch diese Abstraktion im Grunde aus den für den Refinement-Loop charakteristischen Schritten besteht, wird diese Technik analog zur Booleschen Abstraktion im CEGAR-Verfahren realisiert.

3.2.3 Initiale Abstraktion

Die initiale Abstraktion geht aus dem konkreten Programm hervor, indem jeder Call in eine sogenannte *Call-Summary* umgewandelt wird. Diese beinhaltet eine Referenz auf das ursprünglich aufzurufende Objekt und eine *Zusammenfassung* des Calls. Eine Zusammenfassung einer Funktion entspricht dabei dem gröbsten abstrakten Rückgabewert, während eine Zusamenfassung einer POE wiederum eine POE gleichen Typs ist, deren Variablen ihre gröbsten Werte zugewiesen bekommen. So würde z. B. die Zusammenfassung der Funktion Clamp aus Listing 3.4 den abstrakten Wert [0,255] beinhalten, da der Typ USINT den Wertebereich 0 bis 255 abdeckt und dies der Rückgabewert der Funktion ist. Die Zusammenfassung der Clamp-POE aus Listing 3.5 würde dagegen als abstrakten Wert eine Instanz der Clamp-POE sein, deren Variablen X,Y und RES den abstrakten Wert [0,255] haben.

```
1  FUNCTION clamp:USINT
2  VAR_INPUT
3      X:USINT;
4      Y:USINT;
5  END_VAR
6  IF (X+ Y < 127) THEN
7      clamp:= X + Y;
8  ELSE
9      clamp:=127;
10 END_IF;
11 END_FUNCTION
```

Listing 3.4: Clamp-Funktion

```
1  FUNCTION_BLOCK CLAMP
2  VAR_INPUT
3      X:USINT;
4      Y:USINT;
5  END_VAR
6  VAR_OUTPUT
7      RES:USINT;
8  END_VAR
9  IF (X+ Y < 127) THEN
10     RES:= X + Y;
11 ELSE
12     RES:=127;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 3.5: Clamp-POE

Beim Model-Checking würde nun statt dem expliziten Ergebnis eines Aufrufs, der abstrakte Wert der entsprechenden Zusammenfassung zurückgegeben werden, ohne dass der Aufruf selbst simuliert werden muss.

Um unnötige Verfeinerungen zu vermeiden und somit die Qualität der Zusammenfassung zu erhöhen, wird zusätzlich auf die in [BKS12] beschriebene und in ARCADE.PLC implementierte Methode zur *Wertebereichsanalyse* einer Funktion bzw. POE zurückgegriffen. Diese simuliert intern den zusammenzufassenden Aufruf und bildet eine Vereinigung der möglichen resultierenden abstrakten Werte, um nur die tatsächlich erreichbaren Werte zu subsumieren und unerreichbare Ergebnisse wie z. B. 128 im Falle der Clamp-Funktion auszuschließen. Sofern eine derartige Analyse möglich ist, wird die Zusammenfassung des Calls daher aus dem Ergebnis einer solchen Analyse konstruiert. Ansonsten wird, wie zuvor beschrieben, auf den gröbsten abstrakten Wert zurückgefallen. Den großen Vorteil einer derartigen Präparation der Zusammenfassung besteht darin, dass der abstrakte Wert der Zusammenfassung der Funktion aus Listing 3.4 so statt [0,255] nur noch [0,127] beträgt, da der Rückgabewert der Clamp-Funktion per Konstruktion immer in diesem Intervall liegt.

Wie sich die Modulare Abstraktion auf den Zustandsraum auswirkt, wird nun in Abbildung 3.5 am abstrakten Zustandsraum des Programms aus Listing 3.3 dargestellt, wobei es sich bei der Variable *MA* um um die abstrakte Variable der Call-Summary von *anon* handelt. Zwecks Vergleichbarkeit zum Beispiel aus Abschnitt 3.1.3 sei die nicht näher beschriebene Funktion *anon* hier die Summenfunktion.

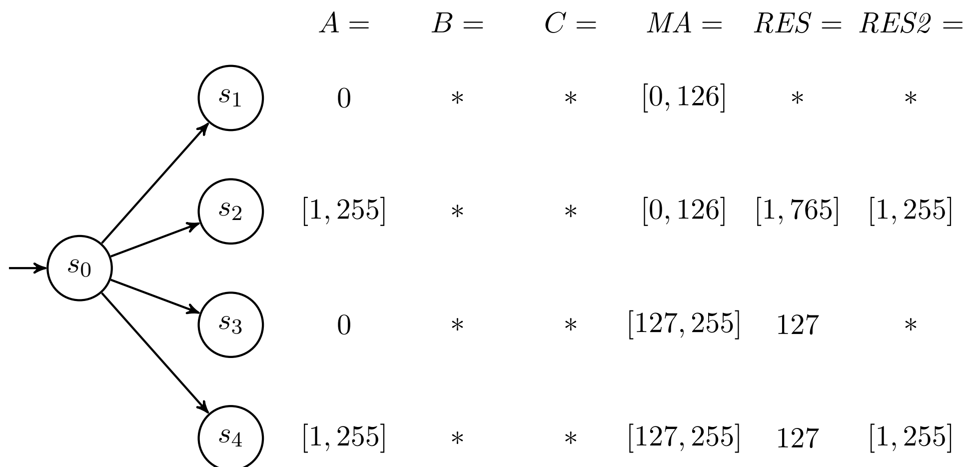

Abbildung 3.5: Abstraktes Transitionssystem $\mathcal{T}'$

Wie es zu diesem Zustandsraum kommt, wird im Folgenden erläutert. Wie aus Abschnitt 2.4.3 bekannt ist, werden die abstrakten Werte in den Eingangsvariablen und den Variablen aus den Call-Summarys so lange verfeinert, bis der Kontrollfluss

durch deren Belegung eindeutig beschrieben werden kann. Im Falle des Listings 3.3 muss also nach der Abstraktion nur noch unterschieden werden, ob $A = 0$ oder $A = [1, 255]$ gilt und ob die Call-Summary von $anon(A, B, C)$ kleiner oder größer-gleich 127 ist. Da der Körper der *anon*-Funktion durch den zusammengefassten Wert [0,255] abstrahiert wurde, muss dieser für die Auflösung des Constraints $[0, 255] < 127$ nicht mehr analysiert werden, sondern lediglich der abstrakte Wert der Zusammenfassung einmal in [0,126] und [127,255] aufgespalten werden. Den entsprechend resultierenden Zustandsraum sieht man in Abbildung 3.5. Der resultierende Zustandsraum ist, wie leicht zu erkennen ist, genau so klein wie bei Anwendung der Booleschen Abstraktion auf das semantisch äquivalente Listing 3.1, obwohl die Boolesche Abstraktion an Listing 3.3 nicht anwendbar wäre.

3.2.4 Abstraktionsverfeinerung

Da die Modulare Abstraktion vom Konzept her verwandt mit der Booleschen Abstraktion ist, läuft auch die Abstraktionsverfeinerung analog ab. Auch hier wird iterativ nach der für ein Gegenbeispiel verantwortlichen Abstraktion gesucht, sofern es sich um ein scheinbares Gegenbeispiel handelt. Dabei konvergiert der abstrakte Zustandsraum $\mathcal{T}'$ stets gegen den konkreten Zustandsraum $\mathcal{T}$.

Zur Überprüfung der Realisierbarkeit eignet sich auch hier das in Abschnitt 3.1.4 vorgestellte Verfahren, da dieses nicht direkt an die Boolesche Abstraktion gebunden ist und auf die gleiche Art und Weise bei der Modularen Abstraktion durchgeführt werden kann.

Analyse linearer Gegenbeispiele

Die Analyse eines linearen Gegenbeispiels und der daraus resultierenden Abstraktionsverfeinerung funktioniert von der Idee her ähnlich wie auch schon bei der Booleschen Abstraktion. Der grundlegende Unterschied zwischen den beiden Verfahren ist, dass die bei der Modularen Abstraktion verwendeten abstrakten Werte nicht mehr als atomare Aussagen eines Zustands vorliegen, sondern in den Call-Summarys der abstrahierten Aufrufe gespeichert sind und dort ausgelesen werden.

Abbildung 3.6 zeigt ein schematisch abgebildetes, lineares Gegenbeispiel eines abstrakten Zustandsraums $\mathcal{T}'$ wie es in ähnlicher Form bereits in Abschnitt 3.1.4 vorkam. Im Gegensatz zur Abbildung 3.2 sind die über den Transitionen aufgetragenen Variablen MA_1, MA_2 und MA_3, wie bereits erwähnt, keine atomaren Aussagen eines Folgezustands, sondern die abstrakten Werte der Zusammenfassungen dreier abstrahierter Aufrufe. Aufgrund der Analogie im Umgang mit solchen Gegenbeispielen bei der Booleschen Abstraktion, wurde die Darstellung entsprechend ähnlich gewählt.

Man beachte, dass die Argumentation hinter dem Vorgehen bei der Booleschen Abstraktion auch hier gilt und die Struktur des linearen Gegenbeispiels auf die bereits

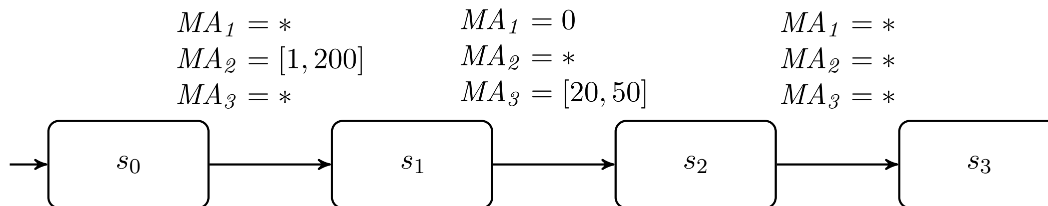

Abbildung 3.6: Lineares Gegenbeispiel aus $\mathcal{T}'$

bekannte Art und Weise ausgenutzt werden kann. Wie man dem dargestellten Gegenbeispiel entnehmen kann, spielt die Belegung der Variablen in den Call-Summarys zuletzt bei dem Übergang von s_1 nach s_2 eine Rolle – ab Zustand s_2 wird der Problemzustand bei jeder Belegung erreicht. Entsprechend wird hier wie bei der Analyse nichtlinearer Gegenbeispiele bei der Booleschen Abstraktion verfahren und zunächst durch iteratives Konkretisieren der Abstraktionen, die MA_1 und MA_2 beinhalten, festgestellt, welche für das Gegenbeispiel verantwortlich ist. Stellt sich heraus, dass das Gegenbeispiel im verfeinerten Programm immer noch existiert, so resultierte es nicht aus der konkretisierten Abstraktion, da das Gegenbeispiel unabhängig von der Verwendung dieser existiert. Die Abstraktion wird in dem Fall entsprechend wieder verwendet und das Problem bei den restlichen Abstraktionen gesucht. Ist das Gegenbeispiel im verfeinerten Programm allerdings nicht realisierbar, so resultierte das Gegenbeispiel tatsächlich aus der konkretisierten Abstraktion und war nur ein scheinbares Gegenbeispiel. Die Abstraktion wird in dem Fall entsprechend verfeinert gelassen und die Spezifikation wird erneut überprüft, ohne dass das scheinbare Gegenbeispiel nochmal auftauchen kann.

Der grundlegende Ablauf der Analyse entspricht daher dem aus Abschnitt 3.1.4. Es müssen hier lediglich statt der atomaren Aussagen die abstrakten Werte der Call-Summarys verwendet werden. Zusammenfassend werden hier also solange Abstraktionen von Calls rückgängig gemacht bis eindeutig gesagt werden kann, ob das Gegenbeispiel auch im konkreten Transitionssystem vorliegt. Die Reihenfolge und Menge der in Betracht zu ziehenden Variablen werden aus der Struktur des Gegenbeispiels abgeleitet.

Analyse nichtlinearer Gegenbeispiele

Im Fall eines nichtlinearen Gegenbeispiels wird auch bei der Modularen Abstraktion heuristisch vorgegangen, da die nichtlineare Struktur nicht preisgibt in welcher Reihenfolge die Variablen in den Zusammenfassungen der Call-Summarys auf Relevanz für das Gegenbeispiel zu prüfen sind. Entsprechend wird ähnlich wie bei der Booleschen Abstraktion vorgegangen – jedoch mit einem anderen Hintergrund.

Die Idee bei der Heuristik aus Abschnitt 3.1.4 war, dass Abstraktionen im Hauptblock in größeren Überapproximationen resultieren als Abstraktionen in den Blöcken von Unteraufrufen. Es wären aber auch *bottom-up*-Heuristiken denkbar, die gerade die Unteraufrufe zuerst verfeinern, um nicht so schnell gegen das konkrete Programm zu konvergieren. Bei der Modularen Abstraktion ist dagegen nur eine *top-down*-Heuristik sinnvoll wie im Folgenden am Beispiel der Aufrufhierarchie aus der bereits bekannten Abbildung 3.3 argumentiert wird.

Die Modulare Abstraktion ersetzt einen Aufruf durch eine Call-Summary, welche die explizite Simulation des *Callee*-Körpers[2] verhindert und stattdessen eine Zusammenfassung des Aufrufs zurückgibt. Würde nun z. B. der Aufruf von Block B in Block A, aus dem Call-Graph aus Abbildung 3.3, wieder konkretisiert werden, aber der Aufruf von Block A im Hauptblock noch abstrahiert sein, so würde die Konkretisierung keinen Einfluss auf die Realisierbarkeit des Gegenbeispiels haben, da der Körper von Block A nach wie vor durch die Abstraktion des Aufrufs im Hauptblock verdeckt bleiben und nicht simuliert werden würde.

Die sich aus dieser Überlegung ergebende Heuristik geht im Grunde also genau so vor wie die im Kontext der Booleschen Abstraktion vorgestellte Heuristik, ist allerdings anders motiviert. Da das Verfahren in Abschnitt 3.1.4 bereits geschildert wurde, sei hier lediglich darauf verwiesen. Es gilt nur zu beachten, dass die abstrakten Werte aus den Call-Summarys zu verwenden sind, statt nach diesen in den atomaren Aussagen zu suchen.

Zusammenfassend werden hier also Abstraktionen an Aufrufen solange Tiefe für Tiefe rückgängig gemacht bis eindeutig gesagt werden kann, ob das Gegenbeispiel auch im konkreten Transitionssystem vorliegt.

[2] callee, engl. Gerufener

4 Fallstudien

In Kapitel 3 wurden zwei Abstraktionstechniken vorgestellt, deren Einfluss auf den resultierenden Zustandsraum im Folgenden anhand einiger Fallstudien untersucht wird. Dabei werden zunächst Spezifikationen an den in den Motivationen der Abstraktionstechniken vorgestellten, Programmen untersucht und an einem weiteren Beispielprogramm der Vorteil der Wertebereichsanalyse bei der Modularen Abstraktion hervorgehoben. Anschließend wird ein Funktionsbaustein nach PLCopen [PLC06] verifiziert, um auch eine reale Anwendung in der Auswertung zu vertreten.

4.1 Verifikation der Problemfälle

Die in dieser Arbeit entwickelten Abstraktionen wurden in Kapitel 3 mit den Listings 3.1 und 3.3 motiviert, die ARCADE.PLC Probleme bereiteten, da die Auswertung bestimmter Ausdrücke die Überprüfung jeglicher Spezifikation erschwerte bzw. unmöglich machte.

Tabelle 4.1 hält die Anzahl der Zustände fest, die der resultierende Zustandsraum bei der Überprüfung einer Spezifikation mit bzw. ohne Verwendung der Booleschen Abstraktion hat. Im einfachsten Fall $\forall\Box true$ erhält man den in Abschnitt 3.1.3 hergeleiteten Zustandsraum der mit nur 5 Zuständen signifikant kleiner ist als der sonst von ARCADE.PLC erzeugte Zustandsraum aus 73.284 Zuständen.

Abstraktion	$\forall\Box true$	$\forall\Box(RES2 < 127)$	$\exists\bigcirc\exists\Diamond(RES \leq 127)$	$\forall\bigcirc\forall\Box(RES \leq 127)$
Ohne	73.284	106.053	73.284	73.284
Boolesche	5	9	388	73.284

Tabelle 4.1: Größe des Zustandsraums bei Überprüfung von Listing 3.1

Ebenso kann man bei den mittleren geprüften Spezifikationen eine immense Verkleinerung des Zustandsraums feststellen. Zu beachten ist allerdings, dass es immer noch Spezifikationen wie z. B. die zuletzt geprüfte gibt, die die Konkretisierung sämtlicher Abstraktionen für ihre Auswertung erfordern.

Ähnliche Ergebnisse liefert auch die Überprüfung der selben Spezifikationen an Listing 3.3, in welchem die Funktion *anon* zwecks Vergleichbarkeit als Summenfunktion angenommen wird und damit ein zu Listing 3.1 semantisch äquivalentes Programm

darstellt, bei dem lediglich die Addition der Variablen in eine Funktion ausgelagert wurde. Tabelle 4.2 veranschaulicht die Resultate der Überprüfungen.

Abstraktion	$\forall\Box true$	$\forall\Box(RES2 < 127)$	$\exists\bigcirc\exists\Diamond(RES \leq 127)$	$\forall\bigcirc\forall\Box(RES \leq 127)$
Ohne	73.284	106.053	73.284	73.284
Modulare	5	9	41.156	73.284

Tabelle 4.2: Größe des Zustandsraums bei Überprüfung von Listing 3.3

Auch hier sieht man, dass die Überprüfung der Spezifikation $\forall\Box true$ in der Erzeugung eines Zustandsraums mit nur 5 Zuständen ausreicht, wie er in Abschnitt 3.2.3 bereits beschrieben wurde. Auch bei den mittleren Spezifikationen bringt die Abstraktion eine markante Verminderung der Zustandszahl. Lediglich die letzte Spezifikation, deren Auswertung eine Konkretisierung aller Abstraktionen erfordert, ändert entsprechend nichts an der Größe des Zustandsraums.

Bei beiden Testprogrammen ist zu beachten, dass die Überprüfung ohne Abstraktion nicht mehr möglich wird, sobald man den Wertebereich der Eingangsvariablen vergrößert. Verwendet man z. B. 32-Bit Eingänge, dann kann jeder Eingang 2^{32} Werte annehmen, weshalb die Auflösung der nicht abstrahierten Ausdrücke zu einer Zustandsexplosion führt. Verwendet man allerdings die vorgestellten Abstraktionen, kann Folgendes beobachtet werden. In den Fällen in denen keine Abstraktion konkretisiert werden muss, wie es z. B. bei $\forall\Box true$ immer der Fall ist, ändert sich die Größe des Zustandsraums im Vergleich zu den festgehaltenen Werten nicht. Dies ist darin begründet, dass die Abstraktionen lediglich symbolischen Charakter haben und nicht vom Wertebereich der Variablen abhängen.

Das zuletzt überprüfte Programm zieht allerdings noch keinen Nutzen aus der Wertebereichsanalyse, da der Rückgabewert in der Summe-Funktion nicht eingeschränkt wird und dem gröbsten abstrakten Wert der Domäne entspricht. Im Folgenden wird ein Fall untersucht, der die Ergebnisse der Wertebereichsanalyse tatsächlich verwendet.

4.2 Verifikation einer beschränkten Funktion

Das in diesem Abschnitt schematisch dargestellte Listing 4.1 soll aufzeigen, welchen Einfluss die Wertebereichsanalyse bei der Modularen Abstraktion haben kann. Dazu wird die aus Listing 3.4 bekannte Funktion *clamp*, deren Wertebereichsanalyse gemäß der Beschreibung aus Abschnitt 3.2.3 den abstrakten Wert [0,127] liefert, im Programm mehrmals aufgerufen. Abhängig von der Auswertung von *clamp* werden die Variablen *RES* und *RES2* entweder mit dem Wert 0 oder 1 belegt. Die Variablen-

deklaration wurde auch in diesem Listing weggelassen. Die verwendeten Variablen sind alle vom Typ USINT und bis auf *RES* und *RES2* allesamt Eingangsvariablen.

```
1  FUNCTION_BLOCK DBLCLAMP
2  [...]
3  IF (clamp(A,B) < 100) THEN
4      RES:=1;
5  ELSE
6      RES:=0;
7  END_IF;
8
9  IF (clamp(C,D) < 128) THEN
10     RES2:=1;
11 ELSE
12     RES2:=0;
13 END_IF;
14 END_FUNCTION
```

Listing 4.1: Clamp-Funktion

Abstraktion	$\forall\Box true$	$\forall\Box(RES2 < 128)$	$\forall\bigcirc\forall\Box(RES < 128)$	$\forall\bigcirc\forall\Box(RES = 0)$
Ohne	184.990	184.990	184.990	184.990
Boolesche	137.446	137.446	137.446	137.446
Modulare	3	3	3	184.990

Tabelle 4.3: Größe des Zustandsraums bei Überprüfung von Listing 4.1

Wie man anhand der Daten aus Tabelle 4.3 erkennen kann, bringt die Boolesche Abstraktion bereits eine deutliche Verminderung der Zustandszahl. Die Modulare Abstraktion kann den Zustandsraum aufgrund der Information aus der Wertebereichsanalyse noch wirkungsvoller verkleinern, da z. B. die Bedingung $clamp(C, D) < 128$ stets erfüllt wird, da bekannt ist, dass die Rückgabe nur im Intervall [0,127] liegen kann und der entsprechende Pfad direkt ignoriert werden kann.

Die bisher überprüften Programme waren alle sehr kurz und im Hinblick auf die auszuwertende Problematik konstruiert, weshalb im Folgenden eine reale Anwendung überprüft wird.

4.3 Verifikation eines Funktionsbausteins nach PLCopen

Die PLCopen ist eine Organisation, deren Ziel die Standardisierung von SPSen ist. In [PLC06] beschreibt diese diverse Funktionsbausteine, die in sicherheitskritischen

Programmen Anwendung finden. Die Bausteine werden jedoch lediglich textuell und in Form von Zustandsautomaten beschrieben, so dass die Implementierung in einer der IEC 61131-3 Sprachen erst durchgeführt werden muss. Eine 188 Codezeilen umfassende Implementierung des PLCopen-Bausteins „ModeSelector“ wird im Folgenden der Überprüfung einiger Spezifikationen unterzogen.

Der ModeSelector ist ein Baustein, welcher 13 Boolesche Eingangsvariablen und einen Eingang für eine Zeitkonstante hat. Des Weiteren besitzt diese POE 11 Boolesche und einen 16-Bit Ausgang. Die Funktionalität, die durch diesen Baustein bereitgestellt wird, ist die zuverlässige Wahl und Änderung des Betriebszustands eines Systems wie z. B. manuell, halb-automatisch oder automatisch. Der Funktionsbaustein verwendet keine Aufrufe, weshalb hier lediglich die Auswertung der Booleschen Abstraktion in Frage kommt.

Tabelle 4.4 zeigt, dass durch die Boolesche Abstraktion einige Spezifikationen in deutlich kleineren Zustandsräumen überprüft werden können, deren Überprüfung zuvor aufgrund einer Zustandsexplosion sehr aufwendig war. Das Problem der Zustandsexplosion wurde im Allgemeinen zwar nicht beseitigt, aber Spezifikationen, die keine Auswertung der problematischen Ausdrücke erfordern, lassen sich nun mit signifikant weniger betrachteten Zuständen überprüfen.

Zusätzlich konnten die Spezifikationen aus Tabelle 4.4 mit der Booleschen Abstraktion jeweils in weniger als 10 Sekunden überprüft werden, während die Überprüfung der ersten beiden ohne entsprechende Abstraktion jeweils über 7 Minuten dauerte.

Abstraktion	$\forall\Box true$	$DiagCode = 0x8000 \Rightarrow [DiagCode = 0x8005, T_1.Q)$	$\exists\Diamond(Error = 1)$
Ohne	15.265	15.589	861
Boolesche	1603	1621	861

Tabelle 4.4: Größe des Zustandsraums bei Überprüfung von ModeSelector

Die Art und Weise, in der die Abstraktionen implementiert wurden, ermöglicht zudem die Überprüfung einer anderen Klasse von Spezifikationen, deren Verifikationsvorgang in [RBH+11] beschrieben und in ARCADE bereits implementiert wurde. Die temporale Logik *past time LTL* (ptLTL) ist eine Variante von LTL, welche Operatoren zum Beschreiben vergangener Zustände bereitstellt. Auf die genaue Definition wird hier verzichtet, da diese nicht von Relevanz ist und in [LPZ85] nachgelesen werden kann. Wesentlich ist, dass deren Verifikation stets lineare Gegenbeispiele liefert.

Da die in dieser Arbeit vorgestellten Abstraktionen nicht mit der Spezifikation selbst, sondern nur mit den Gegenbeispielen und der Programmbeschreibung arbeiten, werden auch Spezifikationen in ptLTL von den Abstraktionen unterstützt. Die mittlere, in Tabelle 4.4 überprüfte Spezifikation ist ein Beispiel für eine ptLTL-Spezifikation. Man sieht, dass die Boolesche Abstraktion auch deren Überprüfung in kleineren Zustandsräumen ermöglicht.

5 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt, die sich mit Model-Checking von SPS-Programmen beschäftigen, zur Modularen und Booleschen Abstraktion verwandte Abstraktionstechniken beschreiben oder die CEGAR-Technik verwenden.

5.1 Model-Checking speicherprogrammierbarer Steuerungen

Die Verifikation von Programmen für speicherprogrammierbare Steuerungen ist Thema einiger Arbeiten. Viele der dort elaborierten Ansätze setzen dazu auf die Übersetzung eines zu überprüfenden SPS-Programms aus einer standardisierten Programmiersprache in ein für einen Model-Checker geeignetes Modell.

So beschäftigt sich Pavlović in ihrer Dissertation [Pav09] mit der Verifikation von in *Funktionsplan* [Int03] vorliegenden SPS-Programmen durch Übersetzung dieser in für den Model-Checker NuSMV [CCG+02] geeignete Modelle. Analog verfährt auch Huuck, indem er in seiner Arbeit [Huu03] in *Ablaufsprache* oder *Anweisungsliste* [Int03] geschriebene SPS-Programme in geeignete Eingaben für den Model-Checker CaSMV [JM01] überführt.

Zhou et al. verfolgen in [ZHGS09] einen ähnlichen Ansatz, überführen SPS-Programme dafür allerdings in *Zeitautomaten* [AD94], welche im Anschluss dem Model-Checker Uppaal [LPY97] als Eingabe dienen.

Arcade.PLC geht konzeptionell gesehen ebenso vor wie die aufgeführten Arbeiten, verwendet allerdings einen eigenen und entsprechend an die Domäne angepassten Model-Checker. Die fundamentalen Ideen hinter Arcade.PLC werden von Schlich et al. in [SKW09] beschrieben und in [BBK10] um die *abstrakte Simulation* erweitert.

5.2 Abstraktionstechniken

Die in dieser Arbeit dargestellte Boolesche Abstraktion orientiert sich an dem von Ball und Rajamani vorgestellten Modell für Boolesche Programme [BR00b, BPR01, BR00a]. In dieser Arbeit wird das Programm jedoch nicht in ein Boolesches Programm übersetzt, sondern die Idee der Beschreibung des Kontrollflusses durch Boolesche Variablen und aggressive Abstraktion von den ursprünglichen Bedingungen von If-Statements übernommen. Einen ähnlicher Ansatz wird in [Sim08] mit der Verwendung Boolescher Variablen zur Beschreibung des Kontrollflusses verfolgt.

Die Modularität von Programmen wurde bereits in [AG04, HQR98, McM97] als wesentlicher Faktor der Skalierbarkeit von Model-Checking erkannt und ausgenutzt. In [BR00a] werden entsprechend auch Zusammenfassungen von Funktionsaufrufen beschrieben, welche insofern ähnlich zu der Idee hinter der Modularen Abstraktion sind, als dass sie die Ausführung der Funktion vermeiden, wenn eine Funktionszusammenfassung vorliegt. Eine Zusammenfassung wird dort allerdings als Tupel $\langle I, O \rangle$ beschrieben, das festhält, welche Ausgabe O eine Funktion für einen bestimmten Eingabewert I hatte. Wird die selbe Funktion später erneut aufgerufen, wird die Zusammenfassung statt der ursprünglichen Funktion aufgerufen.

Sehr ähnlich zur Modularen Abstraktion ist die Verwendung der Funktionszusammenfassung in [AGC12]. Die Zusammenfassung hat dort zwar einen gänzlich anderen Informationsgehalt, ersetzt aber wie bei der Modularen Abstraktion die Ausführung der entsprechenden Funktion und wird mit der Gegenbeispiel-geleiteten Abstraktionsverfeinerung gebündelt.

5.3 Gegenbeispiel-geleitete Abstraktionsverfeinerung

Die Beschreibung des CEGAR-Ansatzes geht auf die in [CGJ$^+$00a] beschriebene und auf [Kur94] aufbauende Arbeit von Clarke et al. am symbolischen Model-Checker NuSMV [CCG$^+$02] zurück. Im Gegensatz zu der in dieser Arbeit vorgestellten Methode wird in [CGKS02] beschrieben, wie die Überprüfung der Realisierbarkeit auf die Überprüfung der Erfüllbarkeit einer Booleschen Formel reduziert werden kann.

Weitere Verwendung findet CEGAR unter anderem in Microsofts Model-Checkern Bebop und Slam2 [BR00a, BBKL10] im Kontext der in [BR00b] eingeführten Booleschen Programme. Darüber hinaus wird in [AGC12] die Bündelung der Gegenbeispiel-geleiteten Abstraktionsverfeinerung mit einem zur Modularen Abstraktion ähnlichen Ansatz beschrieben.

Die dieser Arbeit zugrunde liegende Einbettung von CEGAR in Arcade.PLC stellen Biallas et al. in [BBK10] dar.

6 Zusammenfassung und Ausblick

Im Kontext der Verifikation von SPS-Programmen wurden in dieser Arbeit Abstraktionen vorgestellt, welche zwei Ausprägungen des Problems der frühzeitigen Auswertung von Ausdrücken, die zu einer Zustandsexplosion führen können, lösen. Speziell wurden die Boolesche Abstraktion zur Abstraktion von If-Statements und die Modulare Abstraktion zur Abstraktion von Aufrufen eingeführt. Erstere beschreibt symbolisch welcher Pfad in einem If-Statement genommen wird, während letztere die Simulation von Callees unterbindet, indem sie den Call durch dessen Zusammenfassung ersetzt.

Es wurde jeweils beschrieben wie die initiale Abstraktion durchzuführen ist und die Abstraktionsverfeinerung im CEGAR-Verfahren realisiert werden kann. In diesem Kontext wurde darauf eingegangen wie mit den beiden Arten von Gegenbeispielen in diesem Prozess umgegangen werden kann und wie die Realisierbarkeit eines Gegenbeispiels in einem verfeinerten Zustandsraum unter Berücksichtigung der Struktur eines Gegenbeispiels überprüft werden kann.

In Kapitel 4 wurden die entwickelten Abstraktionen an den in der Motivation aufgeführten problematischen Programmen und einer realen Anwendung mit positivem Ergebnis ausgewertet. Insbesondere wurde festgestellt, dass die Abstraktionen eine Überprüfung bestimmter Spezifikationen an zuvor nicht verifizierbaren Programmen ermöglichen.

Aufbauend auf den Erkenntnissen dieser Arbeit wird im folgenden Abschnitt auf festgestellte Probleme sowie entsprechende Erweiterungs- und Optimierungsmöglichkeiten eingegangen.

6.1 Ausblick

Die vorgestellten Abstraktionen liefern bereits in vielen Fällen gute Ergebnisse – einige Aspekte haben jedoch noch Ausbaupotential. An anderer Stelle sind zudem durch einen alternativen Ansatz bessere Ergebnisse zu erwarten. Im Folgenden wird ein Ausblick auf die angedeuteten Erweiterungsmöglichkeiten gegeben.

6.1.1 Erweiterung der Heuristiken

Bei der Erzeugung der initialen Abstraktion wird in beiden präsentierten Fällen sichergestellt, dass der zu abstrahierende Ausdruck nicht in einer Programmschleife

vorkommt. Zusätzlich wird, wie in Abschnitt 3.1.3 dargestellt, bei der Booleschen Abstraktion berücksichtigt, dass nur If-Statements, deren Bedingung mindestens zwei Variablen beinhaltet, abstrahiert werden.

Diese einfache Heuristik kann aber in bestimmten Fällen dazu führen, dass der abstrakte Zustandsraum größer ist als der konkrete. Am einfachsten kann man sich dies an folgender Überlegung klarmachen. Angenommen ein Programm hat zwei Boolesche Eingangsvariablen, die beide in n Abstraktionen vorkommen. Dann kann dies darin resultieren, dass die Boolesche Abstraktion einen Zustandsraum mit über 2^n Zuständen erzeugt. Dies lässt sich daraus ableiten, dass eine Boolesche Abstraktion symbolisch beide Pfade eines If-Statements modelliert und so bei mehreren aufeinander folgenden If-Statements sich Anzahl möglicher Pfade und entsprechend die der Zustände verdoppelt.

Insofern können Programme mit wenigen Variablen bzw. solchen mit kleinem Wertebereich und vielen If-Statements die Verifikation durch Anwendung der Booleschen Abstraktion erschweren. Eine entsprechende Erweiterung der Heuristik, die über die Anwendung der Abstraktion eines If-Statement entscheidet, könnte die Typen der Variablen und deren Vorkommen in If-Statements entsprechend berücksichtigen, um das geschilderte Problem zu vermeiden.

Auch die Heuristik aus der Analyse der nichtlinearen Gegenbeispiele beider Abstraktionstechniken hat Verbesserungspotential. So kann dort viel Zeit mit der Überprüfung der Relevanz von Abstraktionen für ein Gegenbeispiel verbracht werden, welche bei Beachtung der konkreten Programmlogik aus der Betrachtung ausgeschlossen werden können.

Hier ist die Idee auf einen *SMT-Solver* zurückzugreifen, um die Gegenbeispiele zu überprüfen und einige sonst zu betrachtende Abstraktionen ausschließen zu können. Des Weiteren würde damit die explizite Überprüfung der Realisierbarkeit entfallen.

Ein SMT-Solver ist zunächst ein Programm zur Überprüfung der Erfüllbarkeit einer in *Prädikatenlogik* vorliegenden Formel, welche auf *Theorien* zurückgreifen kann – einer sogenannten SMT-Formel. Insbesondere ermöglichen Theorien der reellen oder natürlichen Zahlen und bestimmter Datenstrukturen, wie z. B. Arrays, Aussagen über die Erfüllbarkeit von Formeln zu machen, die sich auf solche Elemente beziehen. So könnte eine geeignete Überführung des konkreten Programms in eine SMT-Formel und entsprechende Zusammenführung mit der SMT-Formel eines Gegenbeispiels dazu genutzt werden die Realisierbarkeit eines Gegenbeispiels und die Ermittlung der für ein scheinbares Gegenbeispiel relevanten Abstraktionen in den SMT-Solver auszulagern. Dieser Ansatz ähnelt dem von Clarke et al. in [CGKS02] beschriebenen Verfahren, verwendet allerdings eine andere Formelklasse.

Eine mögliche Überprüfung wird im Folgenden exemplarisch an der Verifikation der Spezifikation $\Phi := \forall\Box(B > 127 \Rightarrow RES = 127)$ an dem aus Listing 3.1 bekannten Programm dargestellt. Bei der Überprüfung der Spezifikation würde sich bei Verwendung der Booleschen Abstraktion gerade der aus Abbildung 3.1 bekannte

abstrakte Zustandsraum ergeben, an dem man leicht erkennen kann, dass $s_1 \not\models \Phi$ gilt, da sowohl B als auch RES in diesem Zustand beliebig sein können. Eine entsprechende Kodierung des Gegenbeispiels und des Programms als SMT-Formel könnte im SMT-Solver zu folgender Auflösung führen. Da im Problemzustand s_1 $D = 1$ gilt, würde die Beweisführung damit beginnen.

$$\begin{aligned} D = 1 \quad &\Rightarrow \quad A + B + C < 127 \\ &\Rightarrow \quad B < 127 \\ &\Rightarrow \quad \neg\,(B > 127) \\ &\Rightarrow \quad B > 127 \Rightarrow RES = 127 \end{aligned}$$

Da bekannt ist, dass D den Ausdruck $A + B + C$ abstrahiert, könnte so mittels SMT-Solver darauf geschlossen werden, dass $D = 1$ impliziert, dass es sich um ein scheinbares Gegenbeispiel handelt, da die konkreten Zustände die gefragte Formel erfüllen.

6.1.2 Erweiterung der Überprüfung der Realisierbarkeit

Eine weitere Problematik geht aus der in Abschnitt 3.1.4 vorgestellten Methode zur Überprüfung der Realisierbarkeit eines abstrakten Gegenbeispiels in einem verfeinerten Zustandsraum hervor. Im Speziellen liegt das Problem in der Nachfolgerbildung im zweiten Schritt des Algorithmus. Wie in Abschnitt 2.4.2 beschrieben, ist diese in ARCADE so implementiert, dass alle Nachfolger eines Zustands in einem Schritt erzeugt werden und somit zu einem bestimmten Zeitpunkt alle gleichzeitig im Speicher liegen müssen. Dies ist ein Problem, da es dadurch zu einer Zustandsexplosion kommen kann, wenn ein problematischer Ausdruck im Rahmen der Überprüfung der Realisierbarkeit konkretisiert wird.

Eine mögliche Lösung wäre hier z. B. die Implementierung der Nachfolgerblidung über einen *lazy iterator* zu realisieren, welcher Zustände erst erzeugt, wenn auf sie zugegriffen werden soll und man somit leicht über alle Nachfolger eines Zustands iterieren könnte ohne je alle Nachfolger gleichzeitig im Speicher haben zu müssen. Für die Überprüfung der Realisierbarkeit mit der vorgestellten Methode wäre dies ideal, da diese sich zu keinem Zeitpunkt für die Menge aller Nachfolger interessiert, sondern den verfeinerten Zustandsraum in einer Tiefensuche traversieren könnte.

6.1.3 Erweiterung des Zusammenspiels verschiedener Abstraktionen

Des Weiteren kann das Zusammenspiel der Modularen und Booleschen Abstraktion noch optimiert werden, da diese sich bei gleichzeitiger Anwendung gegenseitig behindern können. So kann es z. B. sein, dass die Modulare Abstraktion an einem

Programm zu einem größeren Zustandsraum führt als die Boolesche Abstraktion. Aufgrund der gegenseitigen Verdeckung von Abstraktionen liefert die gleichzeitige Anwendung jedoch nicht zwangsläufig einen Zustandsraum der maximal so groß ist wie der aus der Booleschen Abstraktion Resultierende.

Auch das Zusammenspiel der Modularen und Booleschen Abstraktion mit anderen Abstraktionen kann ausgebaut werden, um noch kleinere Zustandsräume zu erzeugen. So dürfte z. B. bei Spezifikationen, die sich nicht auf die Eingangsvariablen beziehen, die in Arcade.PLC bereits implementierte Option zum Weglassen der entsprechenden atomaren Aussagen eines Zustands, signifikant kleinere Zustandsräume erzeugen.

6.2 Fazit

Arcade.PLC war bereits vor der Bearbeitung in der Lage viele SPS-Programme mit guten Resultaten auf Einhaltung von Spezifikationen zu überprüfen. Bestimmte Programme widersetzten sich allerdings aufgrund von schwer zu abstrahierenden Ausdrücken jeglicher Untersuchung, selbst wenn die problematischen Ausdrücke nicht von Relevanz für die zu prüfende Spezifikation waren. Die Modulare und Boolesche Abstraktion haben sich als geeignete Ansätze herausgestellt, um solche Spezifikationen dennoch überprüfen zu können, stellen jedoch keine allgemeine Lösung für die Überprüfung sämtlicher Spezifikationen dar. Insgesamt zeigt sich, dass die Verwendung von Abstraktionen eine geeignete Methode zur Erweiterung der Menge der überprüfbaren Programme darstellt.

Literaturverzeichnis

[AD94] ALUR, R. ; DILL, D. L.: A Theory of Timed Automata. In: *Theoretical Computer Science* 126 (1994), S. 183–235

[AG04] ALUR, R. ; GROSU, R.: Modular refinement of hierarchic reactive machines. In: *ACM Trans. Program. Lang. Syst.* 26 (2004), März, Nr. 2, S. 339–369. – ISSN 0164–0925

[AGC12] ALBARGHOUTHI, A. ; GURFINKEL, A. ; CHECHIK, M.: Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In: KUNCAK, Viktor (Hrsg.) ; RYBALCHENKO, Andrey (Hrsg.): *VMCAI* Bd. 7148, Springer, 2012 (Lecture Notes in Computer Science). – ISBN 978–3–642–27939–3, 39-55

[BAMP81] BEN-ARI, M. ; MANNA, Z. ; PNUELI, A.: The temporal logic of branching time. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA : ACM, 1981 (POPL '81). – ISBN 0–89791–029–X, S. 164–176

[BBK10] BIALLAS, S. ; BRAUER, J. ; KOWALEWSKI, S.: Counterexample-guided abstraction refinement for PLCs. In: *Proceedings of the 5th international conference on Systems software verification.* Berkeley, CA, USA : USENIX Association, 2010 (SSV'10), S. 2–2

[BBKL10] BALL, T. ; BOUNIMOVA, E. ; KUMAR, R. ; LEVIN, V.: SLAM2: static driver verification with under 4In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design.* Austin, TX : FMCAD Inc, 2010 (FMCAD '10), S. 35–42

[BCM+90] BURCH, J. R. ; CLARKE, E. M. ; MCMILLAN, K. L. ; DILL, D. L. ; HWANG, L. J.: *Symbolic Model Checking: 1020 States and Beyond.* 1990

[BK08] BAIER, C. ; KATOEN, J. P.: *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. – ISBN 026202649X, 9780262026499

[BKS12] BIALLAS, S. ; KOWALEWSKI, S. ; SCHLICH, B.: Range and Value-Set Analysis for Programmable Logic Controllers. In: *Proceedings of the*

11th International Workshop on Discrete Event Systems. Guadalajara, Mexico, 2012. – To appear

[BPR01] Ball, T. ; Podelski, A. ; Rajamani, S. K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* London, UK, UK : Springer-Verlag, 2001 (TACAS 2001). – ISBN 3–540–41865–2, S. 268–283

[BR00a] Ball, T. ; Rajamani, S. K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, Klaus (Hrsg.) ; Penix, John (Hrsg.) ; Visser, Willem (Hrsg.): *SPIN* Bd. 1885, Springer, 2000 (Lecture Notes in Computer Science). – ISBN 3–540–41030–9, 113-130

[BR00b] Ball, T. ; Rajamani, S.K.: Boolean Programs: A Model and Process For Software Analysis. Microsoft Research, 2000. – Forschungsbericht

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 35 (1986), Nr. 8, S. 677–691. – ISSN 0018–9340

[CC77] Cousot, P. ; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* New York, NY, USA : ACM, 1977 (POPL '77), S. 238–252

[CCG+02] Cimatti, A. ; Clarke, E. M. ; Giunchiglia, E. ; Giunchiglia, F. ; Pistore, M. ; Roveri, M. ; Sebastiani, R. ; Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking, Springer, 2002, S. 359–364

[CES86] Clarke, E. M. ; Emerson, E. A. ; Sistla, A. P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. In: *ACM Trans. Program. Lang. Syst.* 8 (1986), April, Nr. 2, S. 244–263. – ISSN 0164–0925

[CFR+91] Cytron, R. ; Ferrante, J. ; Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), Oktober, Nr. 4, S. 451–490. – ISSN 0164–0925

[CGJ+00a] Clarke, E. M. ; Grumberg, O. ; Jha, S. ; Lu, Y. ; Veith, H.: Counterexample-Guided Abstraction Refinement. In: *Computer Ai-*

ded Verification (CAV 2000) Bd. 1855, Springer, 2000 (Lecture Notes in Computer Science), S. 154–169

[CGJ+00b] CLARKE, E. M. ; GRUMBERG, O. ; JHA, S. ; LU, Y. ; VEITH, H.: *Progress on the State Explosion Problem in Model Checking.* 2000

[CGKS02] CLARKE, E. M. ; GUPTA, A. ; KUKULA, J. H. ; SHRICHMAN, O.: SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In: *Proceedings of the 14th International Conference on Computer Aided Verification.* London, UK, UK : Springer-Verlag, 2002 (CAV '02). – ISBN 3–540–43997–8, S. 265–279

[CGP99] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking.* The MIT Press, 1999. – ISBN 0–262–03270–8

[Cla08] CLARKE, E. M.: *The Birth of Model Checking.* Berlin, Heidelberg : Springer-Verlag, 2008. – 1–26 S. – ISBN 978–3–540–69849–4

[CV03] CLARKE, E. M. ; VEITH, H.: Counterexamples Revisited: Principles, Algorithms, Applications. In: DERSHOWITZ, Nachum (Hrsg.): *Verification: Theory and Practice* Bd. 2772, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–21002–4, 208-224

[EC80] EMERSON, E. ; CLARKE, E. M.: Characterizing correctness properties of parallel programs using fixpoints. In: BAKKER, Jaco de (Hrsg.) ; LEEUWEN, Jan van (Hrsg.): *Automata, Languages and Programming* Bd. 85. Springer Berlin / Heidelberg, 1980. – ISBN 978–3–540–10003–4, S. 169–181

[HQR98] HENZINGER, T. A. ; QADEER, S. ; RAJAMANI, S. K.: You Assume, We Guarantee: Methodology and Case Studies. In: *Proceedings of the 10th International Conference on Computer Aided Verification.* London, UK, UK : Springer-Verlag, 1998 (CAV '98). – ISBN 3–540–64608–6, S. 440–451

[Huu03] HUUCK, R.: *Software Verification for Programmable Logic Controllers.* Kiel, Germany, University of Kiel, Dissertation, April 2003

[Int98] INTERNATIONAL ELECTROTECHNICAL COMMISSION: *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems.* Geneva, Switzerland : International Electrotechnical Commission, 1998

[Int03] International Electrotechnical Commission: *IEC 61131: Programmable controllers.* Geneva, Switzerland : International Electrotechnical Commission, 2003

[JM01] Jhala, R. ; McMillan, K. L.: Microarchitecture Verification by Compositional Model Checking. In: *CAV*, 2001, S. 396–410

[JT09] John, K. H. ; Tiegelkamp, M.: *SPS-Programmierung mit IEC 61131-3.* Springer, 2009. – ISBN 978–3–642–00268–7

[Kur94] Kurshan, R. P.: *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton, NJ, USA : Princeton University Press, 1994. – ISBN 0–691–03436–2

[LPY97] Larsen, K. G. ; Pettersson, P. ; Yi, W.: Uppaal in a Nutshell. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1997), October, Nr. 1–2, S. 134–152

[LPZ85] Lichtenstein, O. ; Pnueli, A. ; Zuck, L. D.: The Glory of the Past. In: *Proceedings of the Conference on Logic of Programs.* London, UK, UK : Springer-Verlag, 1985. – ISBN 3–540–15648–8, S. 196–218

[McM97] McMillan, K.: A compositional rule for hardware design refinement. In: Grumberg, Orna (Hrsg.): *Computer Aided Verification* Bd. 1254. Springer Berlin / Heidelberg, 1997. – ISBN 978–3–540–63166–8, S. 24–35

[MNS01] Miné, A. ; Normale Supérieure École: The octagon abstract domain. In: *In AST 2001 in WCRE 2001, IEEE*, IEEE CS Press, 2001, S. 310–319

[Pav09] Pavlović, O.: *Formale Verifikation von Software für speicherprogrammierbare Steuerungen mittels Model Checking.* Braunschweig, Germany, Technische Universität Braunschweig, Dissertation, Dezember 2009

[PLC06] PLCopen TC5: *Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks.* Germany : PLCopen, 2006

[Pnu77] Pnueli, A.: The temporal logic of programs. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* Washington, DC, USA : IEEE Computer Society, 1977 (SFCS ’77), S. 46–57

[RBH+11] Reinbacher, T. ; Brauer, J. ; Horauer, M. ; Steininger, A. ; Kowalewski, S.: Past time LTL runtime verification for microcontroller binary code. In: *Proceedings of the 16th international conference on Formal methods for industrial critical systems.* Berlin, Heidelberg : Springer-Verlag, 2011 (FMICS’11). – ISBN 978–3–642–24430–8, S. 37–51

[Ryd79] RYDER, B. G.: Constructing the Call Graph of a Program. In: *IEEE Trans. Softw. Eng.* 5 (1979), Mai, Nr. 3, S. 216–226. – ISSN 0098–5589

[Sim08] SIMON, A.: Splitting the Control Flow with Boolean Flags. In: ALPUENTE, M. (Hrsg.) ; VIDAL, G. (Hrsg.): *Static Analysis Symposium* Bd. 5079. Valencia, Spain : Springer, July 2008 (LNCS), S. 315–331

[SKW09] SCHLICH, B. ; KOWALEWSKI, S. ; WERNERUS, J.: Verifikation von SPS-Programmen in AWL mit Hilfe von direktem Model-Checking. In: *AUTOMATION 2009*. Düsseldorf : VDI Verlag, 2009 (VDI-Berichte 2067). – ISBN 978–3–18–092067–2

[Wei81] WEISER, M.: Program slicing. In: *Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA : IEEE Press, 1981 (ICSE '81). – ISBN 0–89791–146–6, S. 439–449

[ZHGS09] ZHOU, M. ; HE, F. ; GU, M. ; SONG, X.: Translation-Based Model Checking for PLC Programs. In: *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*. Washington, DC, USA : IEEE Computer Society, 2009 (COMPSAC '09). – ISBN 978–0–7695–3726–9, S. 553–562