Martin–Luther–Universität Halle–Wittenberg
Naturwissenschaftliche Fakultät III
Institut für Informatik
Lehrstuhl für Software–Engineering und Programmiersprachen

Programmanalysen zur Verbesserung der Softwaremodellprüfung

Dissertation

eingereicht am :	07.11.2011
von :	Dirk Richter
	geboren am 7. Juni 1981
	in Halle (Saale)
1. Gutachter :	Prof. Dr. Wolf Zimmermann
2. Gutachter :	Univ.-Prof. Dr. Jens Knoop
Verteidigungsdatum :	31.01.2012

Stichworte:
symbolisches Kellersystem (SPDS), Modellreduktion, Modellanalyse, Modellprüfung

Kurzfassung

Kaum eine heutige Software ist fehlerfrei. Mit Software*modellprüfung*[1] können Eigenschaften der Software wie Korrektheit formal nachgewiesen werden. Ein solcher Korrektheitsnachweis ist beim Testen der Software nicht möglich, wenn die Software entsprechend komplex ist. Durch Testen kann dann nur die Anwesenheit von Fehlern festgestellt werden. Die *Modellprüfung* ist im Gegensatz zum Testen sehr viel aufwändiger, was den Einsatz der Modellprüfung in der Praxis erschwert. *Programme* einer turingmächtigen Programmiersprache zu verifizieren (Halteproblem), ist nicht möglich. Sie werden daher bei der *Modellprüfung* zu Modellen vereinfacht [220]. Als Modelle werden endliche Automaten, Kripkestrukturen, Petri-Netze oder Kellersysteme (engl. *Pushdown System*, *PDS*) verwendet. Kellersysteme z.B. sind derartig *präzise*, dass damit eine ISO-C kompatible Semantik (insb. Rekursion) von C-*Programmen* nachbildet werden kann [70]. Sie dienen darum als Modelle für die Untersuchungen in dieser Arbeit, da sie außerdem zu weniger Fehlalarmen für sequentielle *Programme* führen als andere Modelle. Präzise Modelle führen i.d.R. zu komplexeren Modellen. Verhaltensbeschreibungen bzw. Modelle zu vereinfachen, ohne dabei aber die Präzision des *Modellprüfung*sprozesses zu verlieren, ist eines der Ziele dieser Arbeit.
Die *Modellprüfung* wird durch die sehr große Anzahl möglicher Modellzustände erschwert (*Zustandsraumexplosion* [81, 183]). In dieser Arbeit wurden daher vorrangig Verfahren entwickelt, um die *Modellprüfung* von Kellersystemen durch eine a priori Zustandsraumverkleinerung zu beschleunigen. Der Speicherverbrauch und die Laufzeit von Modellprüfern werden dabei derart reduziert, dass Modelle, die wegen bisheriger Speicherüberläufe und Zeitüberschreitungen nicht überprüft werden konnten, teilweise nun prüfbar werden. Es wird in dieser Arbeit gezeigt, **dass** und **welche** *Modellanalyse*n dazu genutzt werden können, um die *Modellprüfung* symbolischer Kellersysteme in dieser Hinsicht zu verbessern.

[1] Im Folgenden auch kurz als *Modellprüfung* bezeichnet. Kursiv hervorgehobene Worte finden sich im **Schlagwortverzeichnis**.

Erklärung

Hiermit erkläre ich, dass ich diese Dissertation selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

gez. Dirk Richter

Halle (Saale), den 17. Februar 2012

Inhaltsverzeichnis

1. Einleitung

1.1. Zielsetzung und Motivation

Modellprüfung und deren Verwendung. Häufige Fehlerkorrekturen und Sicherheitswarnungen aktueller Softwareprojekte (z.B. Internet-Browser, Betriebssysteme) zeigen, dass Software meist nicht fehlerfrei ist. Um diesen Umstand zu verbessern, können verschiedene Methoden eingesetzt werden. Eine davon ist Software-*Modellprüfung*[1]. Bei der Software-*Modellprüfung* geht es allgemein um die Frage, ob ein gegebenes System M eine Menge von Eigenschaften E erfüllt (siehe Abbildung 1.1), d.h. dass es in keiner Ausführung zu einem definierten Fehlverhalten bezüglich E kommt. M ist üblicherweise als ein Zustands-Transitions-System und E als eine Menge von Formeln einer temporalen Logik gegeben. *Modellprüfung* für Hardware ist derart erfolgreich, dass alle großen Chiphersteller diese zum zentralen Bestandteil bei der Qualitätssicherung gemacht haben [156]. *Modellprüfung* von Software zur Steigerung der Softwarequalität ist ebenfalls bereits im Einsatz, wie die Projekte JavaPathFinder (NASA) [131], SMV [128] oder Moped [122] zeigen. So werden bereits z.B. Kfz-Steuerungen [187], *Programme* für Microkontroller [209], Gerätetreiber [171, 188], Dateisysteme [255, 7], Netzwerk Filterlogik [44], diverse Protokolle [132] oder Serveranwendungen [133] mittels *Modellprüfung* verifiziert. Allerdings sind die derzeit verfügbaren *Modellprüfer* noch nicht in der Lage, den Quelltext komplexer Softwareprojekte adäquat selbständig und automatisiert zu überprüfen. So wird der Validierungs- und Verifikationsphase bei der Entwicklung eingebetteter Systeme (wie in der Raumfahrt oder im medizinischen Bereich), 50%-70% der Projektressourcen zugeschrieben [206].

Gründe für Modellprüfung. Dennoch ist erwünscht, den *Quellcode* großer Projekte effizient und automatisiert auf ihr richtiges Verhalten hin zu überprüfen. Zu diesem Zweck bieten sich statische *Programmanalysen* an. Solche *Programmanalysen* sind zwar wegen der starken Vereinfachungen des Programmverhaltens schnell durchzuführen, aber auf wenige *Programmeigenschaften* beschränkt. Eine Zeigeranalyse (Point-To-Analyse) bestimmt z.B. für einen Zeiger eine Obermenge seiner möglichen Ziele. Wegen Vereinfachungen brauchen nicht alle dieser Ziele während der Laufzeit realisierbar sein, so dass der Zeiger möglicherweise als uninitialisert erkannt wird, obwohl er zur Laufzeit stets korrekt initialisert ist. Testen von Software durch Ausführen des *Programms* mit verschiedenen Startwerten (Testfälle) ist demgegenüber flexibler. Die Erstellung von Testfällen mit guter Abdeckung[2] des *Programmverhaltens* ist jedoch kompliziert und aufwändig. *Programmüberwachungen* (Monitoring) während der Laufzeit sind zwar *präzise*, aber erzeugen zusätzlichen Ballast (Overhead) und berücksichtigen nur ausgeführte *Programmabläufe* (analog zum Testen). Sie sind daher im Gegensatz zur *Modellprüfung* ebenso nicht in der Lage, sämtliches Fehlverhalten bezüglich E zu entdecken und die Abwesenheit von derart definiertem Fehlverhalten formal nachzuweisen. Z.B. können verschiedene Fehler erst beim Zusammenschluss von Komponenten auftreten (hierzu zählen u.a. sog. rekursive Callbacks [252]) und werden gelegentlich im frühzeitigen Normalbetrieb der Anwendung - wie auch beim Testen - gar nicht entdeckt.

Modelle als Grundlage der Modellprüfung. Man kann derzeit ausgehend vom *Quellcode* eines *Programms* automatisch ein Modell M generieren (siehe Abbildung 1.1) [124, 155, 3], welches das Verhalten des *Programms* z.B. als Zustands-Transitions-System beschreibt. Während der *Modellprüfung* werden für M vorgegebene Eigenschaften E überprüft (z.B. dass vor der Landung eines Flugzeugs stets das Fahrwerk ausgefahren ist). Intuitiv ist klar: je genauer ein solches Modell M das *Programmverhalten* nachbildet, desto *präzise*re Aussagen zur Gültigkeit der Eigenschaften E

[1] Im Folgenden auch kurz nur Modellprüfung genannt.
[2] möglichst wenige Testfälle mit möglichst großer Abdeckung

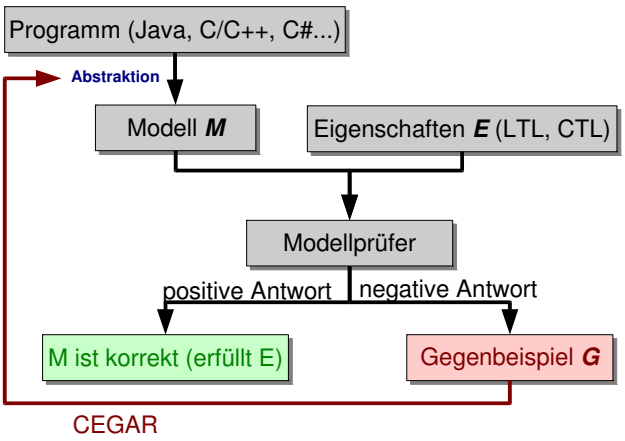

Abbildung 1.1.: Beispielhafte Vorgehensweise bei der Softwaremodellprüfung

können durch die *Modellprüfung* getroffen werden. Diese Aussagen zum Modell M müssen aber nicht für das ursprüngliche *Programm* zutreffen, was als *Fehlalarm* bezeichnet wird. Statische *Programmanalysen* weisen dabei sehr viel häufiger als *Modellprüfung* auf Fehler im Programm hin, welche gar keine Fehler sind [156].

Fehlalarme. Es werden zwei grundlegende Arten von Fehlalarmen unterschieden: *falsche Positiv-Meldungen* (engl. *False Positives*) sowie *falsche Negativ-Meldungen* (engl. *False Negatives*). Der *Modellprüfer* prüft die zu einem Modell M gegebenen Eigenschaften E (siehe Abbildung 1.1).
Antwortet der *Modellprüfer* **negativ**, so wurden während der *Modellprüfung* Verletzungen von Eigenschaften identifiziert. Der Modellprüfer konstruiert dann i.d.R. automatisch das zur verletzen Eigenschaft gehörige Gegenbeispiel. Es dient dem Modellprüfer als Beleg dafür, dass und wo das Modell Fehlverhalten aufweist und kann im weiteren *Modellprüfung*sverlauf für verschiedene Zwecke genutzt werden. Z.B. dienen Gegenbeispiele im Rahmen einer CEGAR-Schleife [77, 258] (Counterexample Guided Abstraction Refinement, siehe Abbildung 1.1) zur Modellverfeinerung, um die Präzision des Modells zu verbessern und erkannte Fehlalarme zu unterbinden. Solche Gegenbeispiele sind Zustandsfolgen der Modelle, die zu den Stellen führen, an denen die Eigenschaften E verletzt sind. Man bezeichnet sie als *unechte Gegenbeispiele* (engl. *Spurious Counterexamples*), falls das Verhalten im ursprünglichen *Programm* C nicht reproduziert werden kann. Unechte Gegenbeispiele stellen falsche Negativ-Meldungen dar und entstehen auf Grund der Vereinfachung (*Abstraktion*) des *Programmverhaltens* (Semantik) zu einem Modell.
Antwortet der *Modellprüfer* andererseits **positiv**, so sollten das Modell M sowie das ursprüngliche *Programm* C die gegebenen Eigenschaften E erfüllen. Der *Modellprüfer* heißt *konservativ*, wenn er bei Negativ-Antworten lediglich echte oder unechte Gegenbeispiele liefert und bei einer Positiv-Antwort das ursprüngliche *Programm* C die gegebenen Eigenschaften E erfüllt. Erfüllen das Modell M oder das *Programm* C die Eigenschaften E nicht, so liegt eine falsche Positiv-Meldung vor. Die *Abstraktion* von *Programm* C zu Modell M ist dann nicht *konservativ* bezüglich der Eigenschaften E (keine sichere *Abstraktion*) oder der *Modellprüfer* ist unzuverlässig (Fehlerhaft).

Beispiel 1.1.1 (Nicht-*konservative* Approximation)
Als Beispiel für falsche Positiv-Meldungen diene eine Abstraktion *von Variablenzuständen. Dabei wird der* Wertebereich *einer Variablen derartig abstrahiert, dass lediglich* 3 *Zustände unterschieden werden: NEG (Variable ist negativ), ZERO (Variable ist 0) und POS (Variable ist positiv). Die arithmetischen Operationen werden entsprechend interpretiert. Z.B. ist* $[\![POS + POS]\!] = POS$, $[\![NEG + ZERO]\!] = NEG$, $[\![ZERO < POS]\!] = true$, $[\![POS < NEG]\!] = false$ *und*

Quellcode 1.1: Beispiel einer Nicht-*konservativen* Approximation mit False-Positive.

```
    x = 1;                          x = POS;
    y = x+1;                        y = x+POS; //y=POS+POS => y=POS
l1: if (y==x) goto l1;              l1: if (y==x) goto l1;
l2: goto error;                     l2: goto error;
```

$[\![POS == POS]\!] = true$ *etc. Dazu sei das* Beispielprogramm *aus* Quellcode 1.1 *links betrachtet. Es ist klar, dass für ganzzahlige Variablen* x *und* y *die Bedingung in Zeile* 3 *nie erfüllt sein wird in der Semantik höherer* Programmiersprachen*. Die Programmmarken* $l2$ *und error sind daher stets* erreichbar*. Die* Abstraktion *ist in* Quellcode 1.1 *rechts abgebildet. Die Variablen* x *und* y *nehmen vor der 3. Zeile beide jeweils den* abstrakten *Wert* POS *an, da in Zeile* 2 *die Addition als* $[\![POS + POS]\!] = POS$ *interpretiert wird. Wegen* $[\![POS == POS]\!] = true$ *ist dann jedoch die Bedingung in Zeile* 3 *an der Marke* $l1$ *stets erfüllt, so dass die Marke* $l2$ *und damit auch error nicht mehr* erreichbar *sind. Ein* Modellprüfer *wird die Erreichbarkeit von error nicht erkennen und ein* False Positive *melden. Dies zeigt, dass diese Form der* Abstraktion *nicht* konservativ *ist und im Rahmen von* Modellprüfung *nicht eingesetzt werden sollte.* $[\![POS == POS]\!]$ *ist eben nicht immer true. Statt dessen sollte für eine sichere bzw.* konservative Abstraktion $[\![POS == POS]\!] = \bot \notin \{true, false\}$ *ergeben, was die Erweiterung der Logik zur dreiwertigen* Kleene Logik *[161, 224] nahe legt.*

Vermeidung von Fehlalarmen. Vermeintliche Fehler werden dem Nutzer eines *Modellprüfer*s als Gegenbeispiele gemeldet. Der Nutzer hat dann zu entscheiden, ob es sich um ein echtes oder ein unechtes Gegenbeispiel handelt, da er dem *Modellprüfer* i.d.R. nur ein vereinfachtes Modell des zu verifizierenden *Programms* gegeben hat. Eine Möglichkeit, dem Nutzer weniger unechte Gegenbeispiele zu melden, ist, dass gefundene Gegenbeispiele nachträglich (**a posteriori**) auf ihre Echtheit hin überprüft[3] und die generierten Modelle verfeinert werden. Im Falle einer automatischen CEGAR-Schleife (siehe Abschnitt 1.2 ab Seite 13 und Abbildung 1.1) wird z.B. zusätzlich vom gegebenen Modell M in ein neues internes weiter vereinfachtes Modell M' für den *Modellprüfer* abstrahiert. Der Modellprüfer ist dann in der Lage, unechte Gegenbeispiele zu erkennen, die durch die *Abstraktion* von M nach M' entstehen. Allerdings ist die Echtheitsprüfung von Gegenbeispielen, d.h. die Rückinterpretation des Gegenbeispiels aus dem Modell M in das ursprüngliche *Programm* C oft nicht trivial [206]. Nachteil dieser Vorgehensweise sind z.B. die mehrfach nötigen Durchläufe des *Modellprüfer*s für die verschiedenen *Abstraktion*sstufen (CEGAR-Durchläufe). Üblicherweise wird in der Praxis dann auf einfachere Modelle oder auf Modellvereinfachung von Hand zurück gegriffen (z.B. durch Markieren irrelevanter Modellbereiche). Je einfacher bzw. ungenauer die Modelle das *Programmverhalten* beschreiben, desto mehr Fehlalarme sind zu erwarten. Damit ist folgendes klar:

Präzisere Modelle liefern weniger Fehlalarme. Eine bessere Möglichkeit, die Fehlalarme zu reduzieren, ist die *Verwendung* von genaueren Modellen.

Bemerkung 1.1.1 (*Modellprüfung* mittels Lösen eines Erfüllbarkeitsproblems)
Bei Satisfiability (SAT) bzw. Satisfiability Modulo Theories (SMT [43]) basierter Modellprüfung *[150, 146, 260, 147, 83] wird das zu prüfende* Programm *oder Protokoll in ein* ***endliches*** *SAT- bzw. SMT-Problem überführt und schließlich mit einem SAT- bzw. SMT-Solver überprüft*[4]*. Vereinfachungen von SAT- bzw. SMT-Formeln [170, 37, 173, 8] vereinfachen und lösen dann das Modell. Zwar sind SAT- und SMT-Techniken (wie auch* automatic test pattern generation *(ATP) [162, 176]) bekannt für ihren geringeren Speicherverbrauch gegenüber BDD-basierter* Modellprüfung *[214, 110] (Binary Decision Diagramm). Jedoch benötigen diese Methoden deutlich mehr Laufzeit*

[3] z.B. durch Simulation via *Symbolisch*er Ausführung [232, 55, 208]
[4] YICES [41] ist ein solcher SMT-Solver. Er wird im Rahmen dieser Arbeit verwendet (siehe Abschnitt 4.3.1 ab Seite 45) und ging als Sieger hervor bei der Computer-Aided-Verification-Konferenz (CAV) 2007 in Berlin.

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
i--;
  if (i>0)
    b(i);
}
```

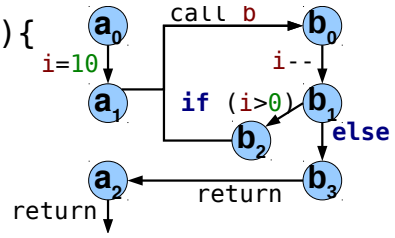

Abbildung 1.2.: Codebeispiel und zugehöriger schematischer *Programmablauf*

als BDD-basierte Modellprüfung *[214]. Insbesondere schlägt die SAT-basierte* Modellprüfung *fehl, wenn die Anzahl der Variablen groß ist [160], wie für Modelle typisch, die aus* Programmiersprachen *gewonnen wurden. Eine Reduzierung des Speicherverbrauchs und der Laufzeit* ***zugleich*** *wird durch die in dieser Arbeit untersuchten Techniken erreicht. Es verbessert sich so nicht nur die Skalierbarkeit, sondern es werden auch Speicherüberläufe sowie Zeitüberschreitungen deutlich reduziert (siehe experimentelle Ergebnisse aus Abschnitt 5 ab Seite 123).*

Wie folgendes Beispiel 1.1.2 zeigt, neigen einfache Modelle (wie für SAT/SMT) schon bei einfachen Algorithmen wie Quicksort zu Fehlalarmen.

Beispiel 1.1.2 (False-Negatives bei zu einfachen Modellen)
Ein Modell kann aus einem Quelltext wie jener aus Abbildung 1.2 links gewonnen werden. Rechts in dieser Abbildung ist der Programmablauf *zu den Methoden a und b schematisch dargestellt. Wird in diesem Beispiel bei der Modellbildung vom konkreten Wert der Variable i in Methode b abstrahiert, so kann in der If-Bedingung (Knoten b_1) nicht mehr unterschieden werden, ob die Methode b rekursiv aufzurufen ist oder nicht. In diesem Fall wird die Rekursion überapproximiert. D.h. es gibt im Modell mehr mögliche Ausführungspfade (hier $a_0a_1(b_0b_1b_2)^kb_0b_1b_3^{k+1}a_2$ mit $k \in \mathbb{N}$) als im ursprünglichen* Programm *(nur $k = 8$ möglich). Die Rekursionstiefe der Methode b hängt vom übergebenen Parameter ab. Ohne weitere Kenntnisse zu den Eigenschaften dieses Parameters ist die Größe des Zustandsraums unbeschränkt wegen eines unbeschränkten Kellers zur Speicherung der* Prozeduraufrufe*. Wird neben der* Abstraktion *von Daten auch nur ein einfaches reguläres statt kontextfreies Modell (Kellersystem) eingesetzt, so werden noch mehr Ausführungspfade möglich ($a_0a_1(b_0b_1b_2)^ib_0b_1b_3^ja_2$ mit $i, j \in \mathbb{N}$).*

Modellprüfer mit lediglich endlichem Zustandsraum (sog. *Finite State Checker* [183]) müssen das Verhalten des Beispiels 1.1.2 approximieren. So verdeutlicht Beispiel 1.1.2, wie durch zu einfache Modelle (keine adäquate Berücksichtigung von Rekursion oder keine Modellierung von Daten) vermeidbare Fehlalarme entstehen.

Bemerkung 1.1.2 (Ähnlichkeit von Zwischencode und Kellerautomaten)
Die meisten modernen Übersetzer generieren zudem heute einen Zwischencode, der ähnlich zu Kellerautomaten ist [157] und daher für die Kellersystem basierte Modellprüfung *gut genutzt werden kann. Dieser Zwischencode erlaubt es, von der Prozessorarchitektur der Zielplattform zu abstrahieren, so dass der* programmiersprachliche *Teil von der konkreten Prozessorarchitektur getrennt wird.*

Viele *Modellprüfer* approximieren Methodenaufrufe wie im Beispiel 1.1.2 und erzeugen auf diese Weise unnötige Fehlalarme[5]. Ansätze, Methodenaufrufe bis zu einer Tiefe k in ein endliches Modell zu modellieren, führen zu einer ungewollten Modellvergrößerung oder zu *False Negatives* [63]. Die *Verwendung* von Kellersystemen ermöglicht jedoch die Berücksichtigung von rekursiven und tief geschachtelten Methoden bzw. Funktionen als kompaktes Modell (ohne Modellvergrößerung).
Nicht zuletzt konnte gezeigt werden, dass Kellersysteme derartig *präzise* sind, dass man das Verhalten von C-*Programmen* (eine mit ISO-C kompatible Semantik) nachbilden kann [70]. Aus diesen

[5] siehe Anlage „Wahl einer geeigneten Modellprüfkette“ auf beiliegender CD

Gründen war es ein erklärtes Ziel dieser Arbeit, die praktische Anwendbarkeit von Kellersystem basierenden *Modellprüfung*sverfahren (engl. Model Checking Pushdown Systems) für Software zu untersuchen und zu verbessern. So können einerseits Modelle einfach gewonnen werden. Andererseits können sämtliche Methodenaufrufe und die gesamte Rekursion exakt im Modell nachgebildet werden, womit eine wichtige Quelle für Fehlalarme ausgeschlossen werden kann.

Zustandsraumexplosion. Die Modellierung einer Variable mit 32 Bit führt dazu, dass ein *Modellprüfer* schon mindestens 2^{32} Zustände zu berücksichtigen hat. Wird lediglich ein einzelnes Bit mehr an Speicherverbrauch modelliert, so verdoppelt sich diese Anzahl der Zustände. Ein solches rasches Anwachsen des sog. Zustandsraums wird als *Zustandsraumexplosion* bezeichnet [63, 81] und erschwert die *Modellprüfung*, da ein *Modellprüfer* nur über begrenzten Speicher verfügt und der *Modellprüfung*sprozess i.d.R. sehr viel Speicher und Laufzeit benötigt. Wegen Speichermangel oder Zeitüberschreitungen kann ein *Programm* dann oft nicht naiv durch Anwendung eines gängigen *Modellprüfer*s geprüft werden. Das Problem verschärft sich in der heutigen Softwarelandschaft immer mehr, da Softwaresysteme immer größer werden, aus vielen Komponenten bestehen und komplexe Daten behandeln. Eine *Abstraktion* von Hand ist dann aufwändig und fehlerträchtig.

Bekämpfung der Zustandsraumexplosion. Um dieses Problem abzumildern, werden in den verbreiteten *Modellprüfer*n Variablen gern nur mit kleinen Bitbreiten modelliert (z.B. nur logische/-boolesche Variablen beim *Modellprüfer* Bebop [237]), da sonst der Suchraum für den Modellprüfer „explodiert“. Desweiteren vermeiden *symbolische* Verfahren die explizite Konstruktion eines Zustands-Transitions-Systems durch implizite Beschreibung mittels eines Automaten. D.h. Transitionsregeln werden nicht mehr explizit notiert und gespeichert, sondern statt dessen werden eigene Modellierungssprachen zur kompakteren Beschreibung verwendet. Dies ermöglicht einem *Modellprüfer* unter *Verwendung* verschiedener Datenstrukturen wie binäre Entscheidungsdiagramme (BDDs) oder geordnete Boolesche Vektoren (ordered boolean functional vectors [97]) die Analyse deutlich größerer Modelle. SAT-[6] und SMT-[7] basierte *Modellprüfung* sind bekannt für ihren geringeren Speicherverbrauch gegenüber BDDs. Allerdings ist BDD-basierte *Modellprüfung* diesen Methoden überlegen, da die Laufzeit deutlich geringer ist. Ziel dieser Arbeit ist, den Zustandsraum der Modelle zu reduzieren und damit auch den Speicherverbrauch für die *Modellprüfung*. Dazu wird für diese Arbeit ein geeignetes etabliertes *Modellprüfung*sverfahren ausgewählt und in den Modellprüfungsprozess mit verschiedenen Methoden eingegriffen.

Verkleinerung des Zustandsraums. Präzisere Modelle bedeuten aber im Umkehrschluss auch eine stärkere Zustandsraumexplosion. Insbesondere dann, wenn Belegungen von Variablen wegen Rekursion mehrfach in einem Keller zu berücksichtigen sind. Um so wichtiger ist es, diese Zustandsraumexplosion bei mächtigeren Modellen zu verhindern. In dieser Arbeit wurde daher ein **a priori** Ansatz zur Verfeinerung solcher Kellersystem-Modelle verfolgt. Durch Eingriffe in den *Modellprüfung*sprozess sollten neu entstehende Fehlalarme (*False Positives* wie auch *False Negatives*) vermieden werden. Einige Forschungsgruppen haben diesbezüglich andere Prioritäten - insbesondere diejenigen, die ohnehin nur auf sehr *unpräzise* Modelle mit vielen Fehlalarmen zur Verhaltensbeschreibung setzen (siehe verwandte Arbeiten). Verhaltensbeschreibungen bzw. Modelle zu vereinfachen, dabei aber die Präzision des *Modellprüfung*sprozesses zu erhalten, ist eines der Ziele dieser Arbeit. Dazu werden *Modellanalyse*n eingesetzt, welche ähnlich zu *Programmanalysen* bei höheren Programmiersprachen sind. *Modellanalyse*n sind mit den zu prüfenden Eigenschaften der *Modellprüfung* verbessert, um genauere und zielgerichtete Analyseergebnisse zu bestimmen. *Modellanalyse*n werden mit *Modelltransformationen* kombiniert, analog zur Kombination von *Programmanalysen* und -transformationen bei höheren *Programmiersprachen*. Bei höheren *Programmiersprachen* können Werkzeuge eingesetzt werden wie z.B. CodeSurfer [100], optimierende Übersetzer oder andere vergleichbare Werkzeuge. Keine der betrachteten Arbeiten wendet diese für Kellersystem-Modelle so an, dass bei Zustandsraumverkleinerung die zu prüfenden Eigenschaften der Modellprüfung berücksichtigt werden. Dies erfolgt in der vorliegenden Arbeit.

[6] satisfiability
[7] satisfiability modulo theories

Eine andere Möglichkeit zur Zustandsraumbekämpfung ist ein CEGAR-Ansatz (siehe Abbildung 1.1 auf Seite 8). Dabei werden Modelle zunächst erst extrem stark abstrahiert, so dass es zu sehr vielen falschen Negativ-Meldungen (unechte Gegenbeispiele) kommt. Anhand dieser Gegenbeispiele wird das Modell dann immer weiter verfeinert und immer wieder der *Modellprüfung* unterworfen. Viele *Modellprüfer* verwenden bereits derartige Ansätze, denn sie erlauben es, schnell Fehler im Modell zu finden. Ist das Modell jedoch fehlerfrei, so benötigen CEGAR-Ansätze bereits für Hardware-Schaltungen mit deutlich geringerem Zustandsraum viele hunderte bis tausende Iterationen für die vollständige Verifikation [258]. Ziel sollte es demnach sein, viele dieser Iterationen a priori zu verhindern. Dies kann durch die in dieser Arbeit beschriebenen a priori Methoden erfolgen.
Eine dritte Möglichkeit zur Zustandsraumbekämpfung ist ein Eingriff in den *Modellprüfer* selbst. Dies hat jedoch den Nachteil, dass zunächst das *symbolische* Modell mit einem möglicherweise sehr großen Zustandsraum in die Datenstrukturen des *Modellprüfers* übertragen wird. Wie auch mit BDDs hat dies den Nachteil, dass bereits vor der Verarbeitung des Modells (quasi schon beim Konstruieren des Modells im Speicher aus der symbolischen Beschreibung) der Arbeitsspeicher nicht genügt und die *Modellprüfung* nicht erfolgen kann [42].
Eine vierte Möglichkeit besteht darin, bereits das *Ausgangsprogramm* in der Hochsprache zu optimieren. Hochsprachen erlauben allerdings keinen direkten Einfluss auf zielarchitekturspezifische Optimierungen. *Programmierer* können daher nur begrenzt indirekt Einfluss nehmen, wenn sie mit der Arbeitsweise des Übersetzers und der Zielarchitektur vertraut sind. Zudem kann ein *Programmierer* ein Programm i.d.R. auf Quelltextebene nicht *optimal* für jede Zielarchitektur formulieren, was auch im Sinn einer Hochsprache ist, um z.B. Portabilität zu gewährleisten.
Insgesamt lassen sich aber alle hier angeführten Ansätze (und sicher auch noch weitere) zur Zustandsraumverkleinerung miteinander kombinieren und führen so dann zu einem Synergieeffekt.

Ziele dieser Arbeit. Ziel dieser Arbeit ist zu zeigen, **dass, welche, wie** und **wie nicht** *Modellanalysen* genutzt werden können, um mittels Modelloptimierung Kellersysteme für die *Modellprüfung* zu verbessern. Dazu werden im Rahmen dieser Arbeit Methoden untersucht und weiterentwickelt, welche vorrangig durch Zustandsraumverkleinerung die *Modellprüfung* beschleunigen. Im Rahmen dieser Arbeit wurden aus dem Übersetzerbau bekannte optimierende Transformationen auf optimierende Transformationen für *symbolische* Kellersysteme übertragen unter Berücksichtigung der zu prüfenden Eigenschaften. Ist im Übersetzerbau der Erhalt des beobachtbaren Verhaltens eines *Programms* maßgeblich für die Korrektheit [84], so ist bei den Untersuchungen dieser Arbeit nur der Erhalt des *Wahrheitsgehalts temporaler Formeln* dafür maßgeblich. Dies ermöglicht andere Transformationen als im Übersetzerbau, welche den Zustandsraum des Modells stärker beeinflussen. Die Ziele sind:

Z1: Möglichst wenige Fehlalarme bei der *Modellprüfung* generieren und daher *präzises* Modell mit unbeschränkter Rekursion einsetzen (Kellersysteme).

Z2: Einen entscheidbaren und terminierenden *Modellprüfungsprozess* nutzen (d.h. bei unbeschränkter Rekursion ist nur beschränkte Parallelität möglich[8], damit ein *Modellprüfer* stets eine klare Ja oder Nein-Antwort liefern kann [227, 213]).

Z3: Direkt die kompakte symbolische Beschreibung des Modells analysieren und transformieren, so dass einer Modell- bzw. Zustandsraumexplosion frühzeitig vorgebeugt wird (Skalierbarkeit).

Dabei sind folgende Kriterien zu erfüllen. Der Eingriff in die *Modellprüfungskette* soll ..

K1: transparent gegenüber dem *Modellprüfungsprozess* (gleichgültig, ob SAT- oder BDD-basierte Modellprüfung) und damit auch in andere *Modellprüfungsprozesse* integrierbar sein[9].

K2: unter *Verwendung* der zu prüfenden Eigenschaften das Modell besser optimieren. können.

[8] siehe Anlage „Wahl einer geeigneten Modellprüfkette“ auf beiliegender CD

[9] Damit wird der Eingriff auch unabhängig bezüglich der eingesetzten Werkzeuge (invariant gegenüber Veränderungen des restlichen *Modellprüfungsprozesses* wie z.B. bei Updates/Fehlerkorrekturen des eingesetzten *Modellprüfers* oder Modellgenerators) oder der Reduktion von *Erreichbarkeit* auf Graphenprobleme [5].

Abbildung 1.3.: Modifikation von Kellersystemen als Eingriff in die Modellprüfkette

K3: sprachunabhängig sein und damit universell für beliebige Ausgangssprachen (C,Java,..) verwendet werden können.

K4: zusätzliche Fehlalarme vermeiden, d.h. die zu prüfenden Eigenschaften nicht beeinflussen[10]

1.2. Vorgehensweise

In dieser Arbeit wird ein **a priori** Ansatz zur Verbesserung von *symbolisch*en Kellersystemen verfolgt. Es wird beschrieben, wie in den allgemeinen *Modellprüfung*prozess eingegriffen und dabei *Programm*- bzw. *Modellanalyse*n genutzt werden können. Dazu wurden verschiedene *Modellprüfer* untersucht und miteinander verglichen. Anhand der Ziele und Kriterien der Arbeit wurde die Modellprüfkette mit dem *Modellprüfer* Moped beispielhaft für einen Eingriff ausgewählt[11], um mit Experimenten die Praxistauglichkeit der vorgestellten Methoden zu erproben. Dabei wurden u.A. die Modellprüfer Blast [229], SPIN [89], Bogor [156], JavaPathFinder (verschiedene Versionen) [249, 131], SMV und NuSMV [128, 54], CWB [196], CBMC [80], YASM [16], Magic [205], Mops [107], VIS [204], Alfred [58, 118], Zing [167, 168, 240], Moped [123, 220], Mucke [17], Bebop [237], F-Soft [259] sowie Murphi [61] berücksichtigt. Danach ist Moped der einzige echte *symbolische Modellprüfer* mit exakter Nachbildung von Rekursion (dank Kellersysteme als *präzise* Modelle weniger Fehlalarme), welcher auch bei unendlichem Zustands- bzw. *Konfigurationenraum* immer terminiert. Für sequenzielle *Programme* sind dann interprozedurale *Modellanalyse*n möglich im Gegensatz zur *Abstraktion* auf endliche Systeme. Zwar bietet Moped nicht wie SPIN direkte Unterstützung zur Beschreibung von asynchroner Parallelität. Man kann beschränkte Parallelität in nicht parallelen Kellersystemen ermöglichen[12]. So ist mittlerweile auch durch das Tool JMoped [119, 65, 64] beschränkte Parallelität für Java [117] Programme erlaubt, was mittels Kontextbeschränkung (context-bounded) realisiert ist [223]. Synchrone Parallelität (z.B. Zuweisung mehrerer Variablen gleichzeitig) wird hingegen bei Kellersystemen durch sog. simultanen Konfigurationsübergang ermöglicht[13].
Ausgehend von einem *Programm* in C oder Java werden dabei (ggf. durch *konservative Abstraktion*, z.B. JMoped) Kellersysteme erzeugt. Durch einen *Modellprüfer* (z.B. Moped) kann dieses dann auf Korrektheit (erfüllen aller vorgegebenen Eigenschaften E) geprüft werden (Abbildung 1.3). Der gewählte Eingriff hat dann sämtliche Kriterien der Zielsetzung zu erfüllen. Abbildung 1.3 veranschaulicht den Eingriff in den *Modellprüfung*sprozess, wo Kellersysteme analysiert und transformiert werden. So sind bestimmte Veränderungen des Modells transparent gegenüber dem *Modellprüfung*sprozess. Diese Änderungen erfolgen durch den Eingriff derart, dass die zu prüfenden Eigenschaften nicht beeinflusst werden. Eine generelle Zwischensprache wie JavaBytecode, CIL (Common Intermediate Language [79]), Assembler oder Jimple (SOOT-Projekt) ermöglicht Sprachunabhängigkeit. Kellersysteme bilden eine solche Zwischensprache. Ein Eingriff in die Modellprüfkette durch Manipulation von Kellersystemen ist daher in diesem Sinn sprachunanhängig. Durch die Konzentration auf Kernsprachelemente kann unter Ausnutzung der zu prüfenden Eigenschaften das Modell noch besser optimiert werden (auch dies ist bei Transformationen im Übersetzerbau eher unwichtig [10, 84]). Insbesondere können die Modellmodifikationen derart realisiert

[10] Invarianz *Erreichbarkeit*/temporale Formeln: es kommt zu keinen neuen Fehlalarmen (False Negative/Positive, z.B. bei *Abstraktion*).
[11] siehe Anlage „Wahl einer geeigneten Modellprüfkette“ auf beiliegender CD
[12] siehe Anlage „Modellierung von Parallelität“ auf beiliegender CD
[13] Mehr dazu im Anhang A ab Seite 133 bei *Parallelzuweisung*en ohne *Interpretation* und ohne zusätzliche *Konfigurationenraum*vergrößerung.

werden, dass diese invariant bezüglich *Erreichbarkeit* bzw. einer gegebenen *temporalen Formel* sind (Definitionen und Details siehe Abschnitt 3.3 ab Seite 32). Durch diese Invarianz einerseits und *präzise* Modelle (Kellersysteme) andererseits können Fehlalarme vermieden werden. Wegen der **Unabhängigkeit** von der Ausgangssprache ist der Eingriff universell auch für andere *Programmiersprachen* und andere *Modellprüfung*sprozesse einsetzbar. Weiter bietet ein solcher Ansatz auch praktische Vorteile. Denn um nicht in Gefahr zu geraten, immer und immer wieder eigene Implementationen in die aktuellen Versionen der Werkzeuge der eingesetzten Modellprüfkette einarbeiten zu müssen, ist die Wahl einer stabilen Zwischensprache ebenfalls vorteilhaft. Zusätzlich bleibt die Unabhängigkeit von Fehlern verschiedener Transformationswerkzeuge gewahrt, was in anderen Eingriffsvarianten nicht unbedingt der Fall ist. Nicht zuletzt ermöglicht die Wahl einer formalen Zwischensprache wie Kellersysteme aber auch die Anwendung der entwickelten Modellverbesserungen auf bereits bestehende Modelle, welche früher bereits erzeugt und ggf. mühevoll von Hand optimiert wurden.
Die Wahl des Eingriffs bedeutet, dass in dieser Arbeit neben der Korrektheit für die *Modellanalyse*n und -Transformationen nicht noch zusätzlich die Korrektheit einer Transformation aus der Quellsprache (wie C oder Java) in Modelle gezeigt werden muss. Wegen wie oben beschriebener Fehleranfälligkeit und Unskalierbarkeit ist im Rahmen von *Modellprüfung* immer noch viel Handarbeit bei der Selektion und *Abstraktion* geeigneter *Programmteile* erforderlich [124], was im praktischen Einsatz sicherlich zu einer Vielzahl von Hand erstellter bzw. angepasster Modelle führt, um z.B. verschiedene Randbedingungen eines *Modellprüfer*s besser zu berücksichtigen.
Möglichkeiten zur Form des gewählten Eingriffs und deren Auswirkungen auf die *Modellprüfung* werden im Abschnitt 4 ab Seite 43 genauer erörtert. Praktische Resultate des vorgestellten Eingriffs wurden in den Experimenten in Abschnitt 5 ab Seite 123 untersucht.

2. Verwandte Arbeiten

Da die *Modellprüfung* von Turing-mächtigen *Programmiersprachen* unentscheidbar ist[1], kann es kein Verifikationswerkzeug geben, das alle *Programme* auf Korrektheit prüft. Aus diesem Grund existieren Modellierungssprachen zur Beschreibung von Nicht-Turing-mächtigen Modellen. Im Idealfall ist so ein Modell (zumindest aus theoretischer Sicht) stets analysierbar und der *Modellprüfer* liefert nach endlicher Zeit ein Verifikationsergebnis (z.B. ein Gegenbeispiel). In der Praxis sind aus *Quellcode* gewonnene Modelle naturgemäß sehr groß. Bereits während der Modellgenerierung kann darauf geachtet werden, dass Modelle nicht zu groß und komplex werden für den weiteren *Modellprüfung*sprozess. Mit diesem Ziel werden derzeit bereits verschiedene Verfahren genutzt, um der Zustandsraumexplosion vorzubeugen. Eine Technik, um mit der Zustandsraumexplosion um zu gehen, ist die Delta-Ausführung (engl. Delta Execution) [2]. Sie operiert simultan auf mehreren Zuständen bei Zustandsübergängen. Weil viele Ausführungspfade während der Durchsuchung des Zustandsraums ähnlich sind, nutzt Delta-Ausführung bei der Analyse eines Ausführungspfades den gemeinsamen Teil anderer Ausführungspfade und führt lediglich die davon verschiedenen Teile (Deltas) erneut aus [2]. Genau betrachtet ist dies eine Art der *symbolisch*en Verarbeitung für explizite Zustandsraumdurchsuchungen [127]. Bei dem gewählten *Modellprüfung*sprozess mit Moped erfolgt bereits eine *symbolisch*e Verarbeitung, so dass diese Art der Techniken zur Verbesserung der *Modellprüfung* keines Einsatzes bedürfen. Dass diese Technik in expliziten *Modellprüfer*n wie JPF und BOX [2] Einsatz findet, zeigt die Bedeutung und Mächtigkeit *symbolisch*er Ansätze. Dies gilt ebenso für *symbolisch*e Ausführung (symbolic execution) [208, 248, 226], wo Zustandsübergänge von konkreten Zuständen mit *symbolisch*en Werten erfolgen. Aber auch andere Themengebiete fanden wegen der Zielsetzung und den Kriterien dieser Arbeit keine wesentliche Betrachtung. Folgende drei Themengebiete sollen daher zur Abgrenzung aufgeführt sein:

- Beschränkte Modellprüfung (Bounded Model Checking) können ein Modell für eine feste maximale Anzahl k an Schritten (Zuständsübergängen) prüfen. Dabei entsteht ein beschränktes Modell, welches z.B. in Form einer SAT-Formel formuliert sein kann. Diese kann durch einen SAT-Löser auf Erfüllbarkeit überprüft werden [150, 146, 260, 147, 83]. Bei beschränkter Modellprüfung sind allerdings die Kriterien K2 und K4 verletzt.

- Reduktionen mittels Partieller Ordnungen [72, 78, 217, 189] (Partial Order Reduction, *POR*) können die Anzahl unabhängiger Verschränkungen (Interleavings) paralleler Prozesse reduzieren. U.U. genügt es, nur wenige repräsentative Verschränkungen zu betrachten, um die Korrektheit eines Modells zu zeigen. In [67] haben wir den umgekehrten Zusammenhang für *SPDS* ausgenutzt (Dualität). Die darin vorgestellte Stotterreduktion ändert das Modell so, dass die Kellersystem-basierte *Modellprüfung* (mit Moped) weniger Anwendungen von Zuständsübergängen zu berechnen hat (Berechnung von Transitionen der Form $a \rightarrow c$ falls $\exists(a \rightarrow b \wedge b \rightarrow c)$). *POR* benötigen initiale Zustandsraumexplorationen, um angewendet werden zu können [129], die Stotterreduktion aus [67] aber nicht. Werden bei *POR* lediglich zu untersuchende Verschränkungen ausgewählt, so ist bei der Stotterreduktion [67] ein einziger Kontrollflusspfad vorgegeben (so gesehen eine einzige Verschränkung). Dieser Kontrollflusspfad wird zu möglichst wenigen Transitionsübergängen[2] zusammengefasst[3].Ähnlich zur Stotterreduktion [67] werden auch von Murphi Berechnungspfade mittels einer Pfadreduktion vereinfacht [129][4]. Da *SPDS* jedoch nicht (unbeschränkt) parallel, sondern rekursiv

[1] Bereits kontextsensitive Datenabhängigkeitsanalysen für *Programme* mit Rekursion sind unentscheidbar [235, 92].
[2] synchron parallel
[3] Deren asynchron aufgefasste Transitionenübergänge bedürfen nur genau eine repräsentative Verschränkung
[4] Die Pfadreduktion in [129] dupliziert Teile der Modellbeschreibung, wenn *Anweisungen* in zwei oder mehr elementaren Pfaden enthalten sind [104]. Die Stotterreduktion [67] erfordert eine solche Duplikation von Teilen der Modellbeschreibung nicht.

sind, entstehen bei *SPDS* wesentlich weniger mögliche Berechnungspfade. Dadurch werden deutlich mehr Reduktionen möglich als es bei Modellen mit unbeschränkter asynchroner Parallelität der Fall ist. Allerdings sind *POR*s bei Abwesenheit asynchroner Parallelität auch nicht einsetzbar für die gewählte Modellprüfkette mit Kellersystemen (Z1).

- Counter-Example guided Abstraction Refinement (CEGAR) beginnt mit einer sehr unpräzisen Abstraktion [71] und verfeinert diese iterativ durch mehrfach wiederholte Modellprüfung. Abstraktion dient dabei der Vereinfachung wesentlicher Teile der Verhaltensbeschreibung. Das vereinfachte System spiegelt dann allerdings nicht präzise die Eigenschaften des Originals wieder (siehe Beispiel 1.1.2 auf Seite 10). Dies führt zu potentiell neuen Fehlalarmen (K4). Dabei auftretende Gegenbeispiele sind ggf. unecht (Counter-Example) und nur durch die Abstraktion entstanden. Diese dienen dann der Modellverbesserung für weitere Durchläufe. Direkte Verbesserungen der CEGAR-Technik wurden in dieser Arbeit nicht untersucht, da der gewählte Modellprüfer Moped bereits einen solchen Prozess einsetzt und zu dessen Änderung ein direkter nicht erwünschter Eingriff in Moped erforderlich wäre (K1).

Neben dem gewählten Eingriff in die Modellprüfkette sind aber auch andere verwandte bzw. vergleichbare Eingriffe möglich. Der Eingriff in die gewählte Modellprüfkette wurde an den Kriterien der Zielsetzung begründet. Transparent gegenüber dem *Modellprüfung*sprozess sind entweder Veränderungen des *Programms* (Variante P) oder Veränderungen des Modells oder Veränderungen in der *Abstraktion* (vom *Programm* zum Modell, Variante A). In allen drei Fällen sind durch einen Eingriff die zu prüfenden Eigenschaften nicht zu beeinflussen, d.h. nur eingeschränkt dürfen Änderungen vorgenommen werden.

Variante P Zur *Programmverbesserung* können u.a. gewöhnliche Übersetzerbau-Techniken [10] eingesetzt werden, um letzlich bessere Modelle zu generieren. Optimierungen der optimierenden Übersetzer können unterschieden werden in maschinenabhängige, maschinenunabhängige und algebraische Optimierungen [195]. Maschinenunabhängige Optimierungen werden bei der Code-Erzeugung genutzt, um eine Zwischensprache (z.B. 3-Adress-Code) zu optimieren [195]. Maschinenabhängige Optimierungen optimieren schließlich den aus der Zwischensprache generierten Maschinencode [195]. Die algebraischen Optimierungen transformieren Programme direkt in der Quellsprache [195], wie es für den in dieser Arbeit gewählten Eingriff auch der Fall ist. Sie lassen sich als Baumtransformationen formulieren [247] und werden üblicherweise unter Beibehaltung des beobachtbaren Verhaltens durchgeführt [195]. Diese sind darum jedoch so allgemeingültig, das optimierende algebraische Transformationen i.d.R. unabhängig von einer zu prüfenden Eigenschaft sind (Widerspruch zu K4, K3, K2). Grundlage für die algebraische Optimierung ist eine mathemtische Struktur der zu optimierenden Sprache [195]. Eine solche Struktur existiert für imperative Sprachen meistens überhaupt nicht [195]. Deshalb wird die algebraische Optimierung im „klassischen“ Übersetzerbau so gut wie gar nicht thematisiert [195]. Andererseits verändern optimierende algebraische Optimierungen u.U. das *Programm* so stark, dass dabei das *Modellprüfung*sergebnis beeinflusst wird (z.B. wegoptimieren von Variablen, über die eine Temporale Formel Aussagen trifft). Dies kann zu Fehlalarmen führen (abhängig vom *Abstraktion*sprozess). Letztlich sind Transformationen auf *Programmebene* auch abhängig vom *Abstraktion*sprozess (Widerspruch zu K1).

Variante A und P Techniken werden für diese Fälle angewendet auf die Quellsprache und sind damit nicht sprachunabhängig (Widerspruch zu K3). Wegen der Unentscheidbarkeit des Halteproblem sind weniger mächtige Analysen möglich als bei einfacheren Modellen. So sind Analysen wie z.B. Slicing bei umso mächtigeren *Programmiersprachen* umso *unpräziser*, da weniger starke Schlussfolgerungen zum *Programmablauf* vorhergesagt werden können. Zudem erschwert ein komplizierterer *Abstraktion*sprozess die Korrektheit des gesamten *Modellprüfung*sprozesses sowie Ändererungen (neue Verbesserungen) im *Abstraktion*sprozess. Wie im F-Soft Projekt z.B. [147, 83, 6] können auf diese Weise mittels *Programmanalysen* wie *Slicing* (*Static Program Slicing*), *Konstanten-Propagation* (*Constant Propagation*), *Verschmelzung* (*Merging*) und *Intervallanalyse* (*Interval Analysis*) kleinere Modelle in Form von *EFSMs* [135] (*extended finite state maschine*) aus C-*Quellcode* generiert werden. Man unterscheidet zwei Arten von Merging.

a) a priori mehrere Modellzustände zu einem zusammenfassen und

b) a posteriori erst während der expliziten Exploration des Modells

F-Soft nutzt Methode a), da direkt *Programme* analysiert und Modelle (EFSMs) anhand der Analyseergebnisse generiert werden. Diese EFSMs werden bei F-Soft zusammen mit Eigenschaften in Form von LTL-Formeln in eine SAT-Formel überführt und durch einen SMT-Solver verifiziert. Jedoch beherrscht F-Soft lediglich beschränkte *Modellprüfung* (Bounded Model Checking).

Variante A Je besser von den Daten eines *Programms* abstrahiert wird, desto wahrscheinlicher gelingt eine erfolgreiche *Modellprüfung*. Ein Eingriff in den *Abstraktion*sprozess ist aber nicht unabhängig bezüglich der eingesetzten Werkzeuge. So sollte aber der vorzunehmende Eingriff möglichst invariant gegenüber Veränderungen des restlichen *Modellprüfung*sprozesses sein (z.B. bei Updates oder Fehlerkorrekturen des eingesetzten *Modellprüfer*s oder Modellgenerators). Bei der Modellgenerierung kann zu Gunsten eines kleineren Zustandsraums (wie bei *Prädikat*-Abstraktion [231, 237, 250], vgl. *abstrakte* Interpretation - engl. *Abstract Interpretation* [178])) komplett auf die Modellierung von Datenstrukturen verzichtet oder davon abstrahiert werden (*Symmetry-Reduction* [192, 191, 76]). Statische *Programmanalysen* wie Strukturanalysen (engl. *Shape Analysis*) werden in diesen Fällen eingesetzt, um Fehlalarme zu reduzieren oder zu vermeiden [199, 243, 193, 256, 115, 93]. Aber nur selten werden Fehlalarme bei der Abstraktion derart vermieden, dass die Abstraktion so präzise ist, dass keine Fehlalarme entstehen[5]. Bei Bebop z.B. werden trotz dieser Fehlalarme die Modelle auf lediglich Boolesche Variablen abstrahiert [237, 238]. Für explizite und endliche *Modellprüfer*, die den gesamten Zustandsraum eines Modells durchsuchen, haben sich Techniken etabliert, welche nicht direkt auf *symbolische* Zustandsräume übertragbar sind [154] (Widerspruch zu Z3). Hierzu gehören Hashing-Techniken (*Hash Compression* bei Murphi [241], *Bitstate Hashing* bei SPIN [89], etc), *Statement-Merging* [244, 85] (Zusammenlegung von Zuständen zu einem *abstrakten* Zustand), State bzw. State Vector Compression [95] (effiziente Zustandsrepräsentation, z.B. Beschränkung auf nur *erreichbare* Heap-Objekte oder Verweis auf Referenz-Zustand mit Änderungsinformationen) etc.. Alle verlangen eine konkrete Repräsentation (keine *symbolische*) des Zustandes, um angewendet werden zu können (Z3). Hashing wird in expliziten *Modellprüfer*n z.B. eingesetzt, um bei der Zustandsraumdurchsuchung bereits besuchte Zustände nicht erneut zu untersuchen. Dabei wird von bereits besuchten Zuständen ein Hashwert bestimmt und dieser in einer Hashtabelle eingetragen. Wie in Abschnitt 1.2 ab Seite 13 beschrieben, kann dies sogar zu False Positives führen, wenn der „*Modellprüfer*“ beim wiederholten Auftreten eines Hashwertes die weitere Exploration des Zustandsraums beendet und diese Entscheidung lediglich anhand des nicht eindeutigen Hashwertes trifft (Widerspruch zu Z2). All diese Techniken führen durch den Präzisionsverlust bei der Abstraktion zu potentiell zusätzlichen Fehlalarmen. Wegen Kriterium 4 (keine weiteren Fehlalarme erzeugen) wurde von der Untersuchung von *Abstraktion*s-Techniken in dieser Arbeit abgesehen, welche potentiell zusätzliche Fehlalarme erzeugen. Das von uns in [71] vorgestellte Verfahren besitzt nicht diesen Anspruch (d.h. zusätzliche Fehlalarme sind dort gestattet) und ist ähnlich zu [75]. Auch dort werden Modell und temporale Eigenschaften derart abstrahiert, dass bei Modellprüfung der Abstraktionen Rückschlüsse auf das Ursprungsmodell möglich sind. Allerdings werden in [75] nur endliche Modelle sowie LTL betrachtet und es wird die Abstraktion manuell vom Nutzer durch die Bandera Abstraction Specification Language (BASL) bestimmt. [71] ist auch für unendliche Strukturen (*SPDS*) nutzbar, nicht auf LTL beschränkt und abstrahiert vollautomatisch mittels eines frei wählbaren Abstraktionsgrads α. Anders als die Predikatabstraktion in [99] operieren wir in [71] direkt in der symbolischen Beschreibung und nicht auf dem zu Grunde liegendem ggf. sehr großen oder unendlichem Transitionssystem. In [160], [48] und [111] werden Beweise bzw. Gegenbeispiele für das Erfülltsein bzw. Unerfülltsein von SAT-Formeln verwendet, um relevante Modellteile zu identifizieren und zu abstrahieren. Wir hingegen lösen in [71] das aufgestellte SAT-Problem nicht (weil aufwändig) und untersuchen es lediglich mittels effizienter Heuristiken auf wichtige Modellteile. In [172] werden als Prädikatabstraktion wichtige Prädikate bezüglich der temporalen Spezifikation berechnet. Im Gegensatz zum Verfahren aus [71] terminiert

[5] Die Möglichkeit einer präzisen Abstraktion [233, 108] ist der Grund für die vergleichsweise ausführliche Betrachtung der verwandten Arbeiten bei Abstraktionstechniken.

deren Verfahren nicht immer, Die in [215] vorgestellte Deduktive Modellprüfung generiert wie [71] Abstraktionen basierend auf gegebenen LTL Formeln. Diese ist allerdings auch nur auf LTL und endliche Modelle beschränkt und erfordert signifikanten manuellen Eingriff im Gegensatz zu [71]. In [130] werden Techniken vorgestellt zur Abstraktion von Kontrollfluss und Daten. Auch diese ist beschränkt auf LTL, endliche Modelle und operieren auf diskreten Kripkestrukturen statt in der symbolischen Beschreibung eines *SPDS*. Zudem muss auch hier im Gegensatz zu [71] für die Abstraktion eine Zuordnung (mapping) von abstrakten auf konkrete Zustände manuell erfolgen.

Zwischen-Ergebnis Alle vorgestellten Varianten können mit der gewählten Variante für diese Arbeit kombiniert werden, so dass jeder Zwischenschritt zu einer kompakteren Darstellung führt und daher a priori der Modellexplosion vorgebeugt wird (Skalierbarkeit). Oft werden nur endliche Modelle (vom unendlichem Programm) betrachtet, weshalb viele unnötige Fehlalarme durch den Abstraktionsprozess entstehen können. Letzte sind wegen der ISO-C konformen Semantik für *SPDS* [70] bei dem Ansatz in dieser Arbeit deutlich reduziert. Methoden dieser Arbeit zur Modellverfeinerung können ergänzend zusätzlich angewendet werden.

Sehr nah verwandt zu den Methoden dieser Arbeit sind daher solche Arbeiten, die sich mit dem Optimieren von Modellen beschäftigen (Modellreduktion). Bezüglich der betrachteten Gliederung eigenen sich zu diesem Zweck Maschinenunabhängige Optimierungen aus dem Übersetzerbau auf der Zwischensprache *SPDS*. Arbeiten, welche sich gezielt mit dem Optimieren von Modellen beschäftigen sollen daher im Folgenden näher betrachtet werden.

2.1. Techniken zur Modelloptimierung

Durch Optimieren von Modellen kann deren Zustandsraums verkleinert werden. Im folgenden sei dieser Vorgang als *Modellreduktion* bezeichnet. Nicht zu verwechseln ist dieser mit Programm- bzw. Modell-Analysen, wo lediglich Informationen über ein Programm bzw. ein Modell gewonnen werden. Für Modellreduktionen sind in einem ersten Schritt Informationen über ein Modell zu gewinnen (durch eine Modellanalyse). Diese werden genutzt, um das Modell zu optimieren. Bei der späteren Betrachtung verschiedener Modellreduktionen werden für die jeweilige Modellreduktion verschiedene einsetzbare Programmanalysen im entsprechenden Abschnitt untersucht (Abschnitte „Einsetzbare Programmanalysen“). Allerdings sind uns derzeit außer [220] keine Arbeiten bekannt, welche gezielt zum Zweck der Modellprüfung *symbolische* Kellersysteme (*SPDS*) analysieren und verkleinern. In den untersuchten Veröffentlichungen wurden bekannte Programmanalysen aus dem Übersetzerbau bisher **nicht** für *SPDS* eingesetzt, um damit die Modellprüfung zu beschleunigen. Techniken zur Modellreduktion mittels *Modellanalyse*n gibt es bereits bei anderen *Modellprüfer*n. Im Rahmen dieser Arbeit wurden daher verschiedene Techniken zur Modellreduktion untersucht, für *SPDS* angepasst und verbessert. Eine Übersicht der untersuchten bestehenden Techniken zu den einzelnen Modellprüfern befindet sich auf der beiliegenden CD in Anlage „Übersicht zu den Techniken zur Modellreduktion“. Oft sind dies einfache Analysen, angewandt auf lediglich einfache bzw. endliche Modelle (Z1). Techniken, welche erst während des *Modellprüfung*sprozesses eingreifen, bedürfen erst die Generierung des *symbolisch* beschriebenen Modells im Speicher (z.B. mittels BDDs), um angewendet werden zu können (Z3). Hierzu gehört z.B. Macro Expansion [257], wo BDD-Variablen durch äquivalente *Ausdrücke* ersetzt werden, um BDD-Variablen einzusparen. Dieser Ansatz ist (im Gegensatz zum gewählten) nicht transparent gegenüber dem Modellprüfungsprozess (K1) und operiert nicht in der *symbolisch*en Beschreibung des Modells (Z3), so dass ein potentiell sehr großes Modell erst explizit erzeugt werden muss um es dann zu vereinfachen. Aus der Literatur bekannte Techniken zur Modellreduktion sind nicht immer direkt miteinander und mit den Verfahren dieser Arbeit vergleichbar und unterscheiden sich trotz ähnlichen oder gleichen Namens oft in vielen Details, da unterschiedliche zu Grunde liegende Modelle oder Protokolle i.d.R. zu verschiedenen Verfahren führen.

2.1.1. Vereinfachung von Ausdrücken

Im Übersetzerbau werden verschiedene Techniken eingesetzt, um das Auswerten von Ausdrücken zu vereinfachen (z.B. Peephole Optimization [225, 159, 10] oder Finden einer optimalen Auswertungsordnung [247] bzw. Anweisungsreihenfolge [195]). Allerdings liegt der Fokus dort vorrangig auf Codeerzeugung von effizientem Zwischencode mit dem Ziel einer möglichst schnellen Ausführung des Programms [247]. Dabei werden zusätzliche Hilfs-Variablen bzw. Register der CPU verwendet, um Teilausdrücke auszuwerten. Sollen Ausdrücke aber lediglich symbolisch vereinfacht werden ohne dabei neue Variablen bzw. Register zu verwenden, können diese Techniken nicht universell genutzt werden. Verwand zum Vereinfachen von Ausdrücken sind daher eher Techniken wie Konstanten-Propagation und Konstanten-Faltung [169, 59, 221, 57]. Bodik, Gupta und Sarkar [198] berücksichtigen dabei z.B. nur Zuweisung einer Konstante und Addition einer Konstanten. Dies schränkt die Verwendbarkeit anderer Operatoren wie Multiplikation, Division und Modulo ein. Knoop und Rüthing erklären in [125] eine spezielle Konstanten-Propagation für *Quellcode* mit Prädikaten (engl. predicated code). Im Gegensatz zu gewöhnlicher Konstanten-Propagation und Konstanten-Faltung ist ihre Technik zudem in der Lage, Terme als Konstant zu identifizieren, auch wenn ihre Argumente es nicht sind. Sie ist weiter optimal für azyklische Programme und sog. Hyperblocks [125]. Dennoch ist es mit diesen Techniken z.B. nicht möglich, Ausdrücke der Form „$x + 1 < y + 1$“ auf „$x < y$“ zu vereinfachen, wenn x und y nicht konstant sind. Zu diesem Zweck sind weitere (algebraische) Umformungen nötig, welche im Rahmen dieser Arbeit erörtert werden sollen.

2.1.2. Slicing und Lebendigkeit

Slicing-Techniken [88] werden bei SPIN für Promela [136, 137] sowie bei Bogor [153] genutzt, um effizient unwichtige Teile der Modellbeschreibung zu identifizieren. Der JavaPathFinder (JPF) nutzt u.a. das Slicing-Tool aus dem Bandera-Projekt [102], um Modelle zu verkleinern. Durch einen Slice werden *lebendige* Teile einer Modellbeschreibung erkannt und dann überflüssige Teile eliminiert. Derartige überflüssige Teile können z.B. eine niemals aufgerufene Methode oder eine nirgends benutzte Variable sein. Sie wird als *tot* bezeichnet und leistet keinen Beitrag zum modellierten Verhalten. Im Gegensatz dazu sind die anderen Teile einer Modellbeschreibung *lebendig* und damit wichtig für das modellierte Verhalten. Ein genereller Ansatz zum Finden von totem Code mittels *temporaler Formel*n findet sich in [140]. Dort wird für jede Variable x zu jedem Zeitpunkt die Gültigkeit einer *temporalen Formel* geprüft. Dazu müssen aber temporale Formeln auf Gültigkeit überprüft werden, was eine einzelne Modellprüfung nicht unbedingt beschleunigt (Zielstellung dieser Arbeit). Wie lebendige Variablen bei Abwesenheit von Rekursion, dafür aber im geteiltem Speicher und in Kommunikationskanälen für Prozess-Algebras identifiziert werden können, wird in [151] beschrieben. Slicing kann auch bei der Bestimmung von maximalen Ausführungszeiten (*Worst-Case Execution Times*, WCET) [50] eingesetzt werden. Letzendlich findet Slicing auch *Verwendung* bei der Modellreduktion für Nebenläufigkeit bezüglich (in zu prüfenden Eigenschaften verwendeter) Zustandsprädikate [154]. Allerdings sind uns keine anderen Arbeiten bekannt (außer den eigenen), welche Slicing für *SPDS* einsetzen, um ein Modell zu verkleinern bezüglich der im Rahmen der Modellprüfung zu prüfenden Eigenschaften. Dies ist Gegenstand von Untersuchungen dieser Arbeit.

2.1.3. Elimination unwichtiger Variablen

Im *Modellprüfer* SPIN (Promela) werden mittels Kontrollflussanalysen nichtlesend verwendete Variablen aus dem endlichen Modell (ohne Berücksichtigung von Rekursion eliminiert) [96]. Diese Techniken sind auf *SPDS* übertragbar und wurden in der vorliegenden Arbeit entsprechend weiterentwickelt. Der Modellprüfer Moped (Remopla) z.B. identifiziert vereinfachend nur syntaktisch nicht verwendete globale Variablen innerhalb eines Prozedurablaufs [220]. Derartige Variablen sind zumindest für das *Erreichbarkeitsproblem redundant*. Moped versteckt diese vor den OBDDs für Variablenzustände, um den *Konfigurationenraum* synthetisch zu verkleinern [220]. Durch Kopier-Propagation aus dem Übersetzerbau [195] können durch Propagieren von einfachen Variablenzu-

weisungen der Form „x=y“ dahinter liegende Ausdrücke der Form „a*x“ zu „a*y“ vereinfacht werden. Dann wird u.U. die Zuweisung „x=y“ und ggf. auch die Variable „x“ überflüssig für das Programm. Die Techniken aus dem Übersetzerbau berücksichtigen allerdings keine temporalen Eigenschaften, welche für die Modellprüfung entscheidend sind. Ziel bestehender Techniken ist häufig nur die Erkennung und das Entfernen unwichtiger (z.B. toter) Variablen [96, 195], nicht jedoch die Minimierung des Zustandsraums.

2.1.4. Verkleinerung von Variablentypen

Während der maschinenabhängigen Codeerzeugung im Übersetzerbau werden Variablen oft auf 32 oder 64 Bit Register abgebildet. Daher haben Variablen in höheren *Programmiersprachen* oft feste Wortbreiten und die Notwendigkeit der Typminimierung ist i.d.R. nicht gegeben. Dies ist ein möglicher Grund, warum es hierzu wenig vergleichbare Arbeiten aus dem Übersetzerbau gibt. Lediglich Intervallanalysen liefern brauchbare *Modellanalyse*n (aber keine Transformationen). Bei der automatischen Testdatengenerierung z.B. wird eine Reduktionstechnik namens Dynamische *Wertebereich*reduktion (engl. *Dynamic Domain Reduction*, *DDR*) [185, 4] eingesetzt. Das Ziel von *DDR* ist es, vorgegebene *Wertebereiche* von Variablen aufzuteilen. Durch Hinzunahme immer weiterer Ungleichungen zu einem linearen Optimierungsproblem (Constraints) und mittels *symbolisch*er Ausführung werden dabei Eingabedaten zum Softwaretesten konstruiert, welche möglichst kompakt und dennoch repräsentativ sind (Äquivalenzklassen). Da Methoden wie DDR oft Heuristiken sind, können diese zwar hervorragend zum Testen, nicht jedoch zur *Modellprüfung* verwendet werden. Eine konservative Variante für Modellprüfung entspricht dann einer Abstraktion des Modells von konkreten Variablenwerten auf abstrakte [71] (siehe verwandte Arbeiten zur Abstraktion früher). Daher kommen zur Verkleinerung von Variablentypen auch prinzipiell Abstraktionstechniken in Betracht, wenn sie derart präzise abstrahieren, dass keine Fehlalarme entstehen. Im F-Soft-Projekt gibt es einen Ansatz, Wertebereiche in endlichen Modellen zu verkleinern [6]. Dieses Projekt operiert jedoch auf *Quellcode*ebene und nicht im Modell (K1) und erlaubt nur die Analyse von einfachen Intervallgrenzen (Minimum sowie Maximum). Für die Verkleinerung von Variablentypen in *SPDS* unter Berücksichtigung zu prüfender Eigenschaften, wie temporale Formeln, sind uns bisher keine Arbeiten bekannt (außer den eigenen).

2.2. Fazit

Viele Arbeiten zur Modelloptimierung betrachten oft nur endliche Modelle, was zu mehr Fehlalarmen führt als im gewählten Ansatz. Die dort eingesetzten Techniken wurden in der betrachteten Literatur nicht für *SPDS* untersucht (Ausnahme [220] mit einfachen Untersuchungen zu OBDD-Variablenordnungen sowie für die interne Repräsentation von Variablenbelegungen im Modellprüfer). Entsprechend müssen diese Techniken für den Einsatz mit *SPDS* entsprechend angepasst und weiter entwickelt werden. Programmanalysen aus dem Übersetzerbau hingegen können einfacher auf *SPDS* angewendet werden, da Programmiersprachen üblicherweise sehr mächtig sind und *SPDS* sich darin Ausdrücken lassen. Allerdings liegt der Fokus dort eher auf effiziente Codeerzeugung statt einem kleinen Zustandsraum, was teilweise kontraproduktiv sein kann. So vergrößert sich der Zustandsraum zu Gunsten einer effizienteren Programmausführung, wenn z.B. für Schleifenoptimierung die Schleifeninvarianten verlagert [195] und damit zusätzliche Hilfsvariablen eingesetzt werden. Dies kann dann die Modellprüfung erschweren oder gar ihre Durchführbarkeit verhindern (Speicherüberlauf). Andererseits berücksichtigen die Techniken aus dem Übersetzerbau nicht die für die Modellprüfung wichtigen zu prüfenden Eigenschaften. Auch in diesem Themengebiet sind uns keine Arbeiten bekannt (außer den eigenen), welche *SPDS* (unendliche Modelle mit Rekursion) optimieren unter Beachtung der zu prüfenden Eigenschaften in Form von temporalen Formeln. Daher sind auch diese Techniken entsprechend für den Einsatz auf *SPDS* für den Zweck der Modellprüfung entsprechend anzupassen und weiter zu entwickeln.

3. Modellprüfung

> All non-trivial semantic questions about programs from a universal *programming* language are undecidable (Rice's Theorem 1953 [106]).

In diesem Abschnitt wird definiert, was formal unter einem Kellersystem zu verstehen ist, wie diese kompakt (*symbolisch*) beschrieben werden können und was *Modellprüfung* von Kellersystemen bedeutet. Für die Korrekheitsnachweise sind eine *präzise* Definition von Syntax und Semantik für ***symbolische*** Kellersysteme wichtig. Bislang sind uns dazu keine Arbeiten bekannt[1]. Dazu wird die in dieser Arbeit verwendete Notation zur Beschreibung für *symbolische* Kellersysteme formal erklärt und erläutert. Nicht zuletzt wird auf grundlegende Konzepte von *Programmanalysen* eingegangen, welche im Kontext von *SPDS* entsprechend modifiziert als *Modellanalysen Verwendung* finden. Sie gewinnen später die nötigen Informationen (Analyseergebnisse) für die Modellverbesserung.

3.1. Kellersystem

Definition 3.1.1 (*Kripkestruktur*)
Es sei S eine endliche oder unendliche Menge von Zuständen und $I \subseteq S$ eine gegebene Menge von Initial- bzw. Startzuständen. Die Zustände S seien mit Merkmalen[2]. aus einer endlichen oder unendlichen Menge A annotiert. Hierzu dient die Annotationsfunktion $L : S \to 2^A$. $M = (S, \to, I, L)$ wird als Kripkestruktur *bezeichnet, falls $\to \subseteq S \times S$ eine Überführungsrelation der Zustände ist. Die Menge aller Kripkestrukturen sei* KS. *$\to$ wird als* Transitionsrelation *bezeichnet. Eine Folge von Zuständen $s_1 \to s_2 \to s_3 \ldots$ heißt* Lauf *der Kripkestruktur. s_{i+1} heißt* Nachfolgezustand *von s_i. Ein Lauf $s_1 \cdots \to s_n$ heißt* terminiert, *falls es keinen Nachfolgezustand $b \in S$ derart gibt, dass gilt $s_n \to b$. Eine Menge von Zuständen $B \subseteq S$ heißt* erreichbar[3] *aus $B' \subseteq S$ (in Zeichen $B' \rightsquigarrow B$), falls es Zustände $b \in B$ und $b' \in B'$ gibt, so dass ein Lauf $b' \to \cdots \to b$ existiert. Existiert kein solcher Lauf, so ist B* nicht *erreichbar aus B' (in Zeichen $B' \not\rightsquigarrow B$). Eine Menge an Zuständen $B \subseteq S$ heißt in M* erreichbar *bzw. nicht erreichbar, falls $I \rightsquigarrow B$ bzw. $I \not\rightsquigarrow B$. Für $B = \{b\}$ bzw. $B' = \{b'\}$, wird auch kurz $b \rightsquigarrow B'$ bzw. $B \rightsquigarrow b'$ bzw. $b \rightsquigarrow b'$ geschrieben. Die Menge aller in M* erreichbaren *Zustände sei mit $Post^*(M)$ bezeichnet: $Post^*(M) := \{s \mid s \in S, I \rightsquigarrow s\}$. Ein Lauf $s_1 \to s_2 \cdots \to s_n$ heißt* essentiell *für $B \subseteq S$, wenn gilt $s_n \rightsquigarrow B$.*

[1] Zwar gibt es Beschreibungen zur Eingabesprache Remopla für den *Modellprüfer* Moped, diese sind jedoch *unpräzise* und weitgehend informal. Sie eignen sich daher nicht als Beweisgrundlage.
[2] Merkmale werden auch als Atome bezeichnet und sind primitive Eigenschaften wie z.B. dass eine Variable x einen Wert größer 5 hat oder dass eine Variable gerade ist.
[3] Die Prüfung auf *Erreichbarkeit* von *Konfigurationen* ist eine der einfachsten Formen der *Modellprüfung*.

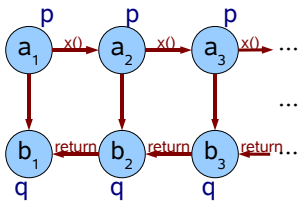

Abbildung 3.1.: Beispiel einer Kripkestruktur

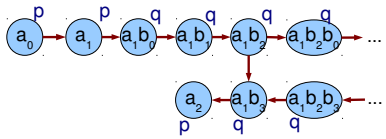

Abbildung 3.2.: Kripkestruktur für Beispiel 1.1.2

Beispiel 3.1.1 (Beispiel einer Kripkestruktur) *In Abbildung 3.1 ist ein Beispiel einer Kripkestruktur $M = (S, \to, I, L)$ zu sehen. $S = \{a_1, b_1, a_2, b_2 \dots\}$ ist darin die Menge der Zustände, $\to := \{a_i \to a_{i+1}, a_i \to b_i, b_{i+1} \to b_i \mid i \in \mathbb{N}_{>0}\}$ die Transitionsfunktion und $I = \{a_1\}$. $A = \{p, q\}$ ist die Menge der Merkmale und die Annotationsfunktion L gibt den Zuständen a_i das Merkmal p ($L(a_i) = \{p\}$) und den Zuständen b_i das Merkmal q ($L(b_i) = \{q\}$). In diesem Beispiel ist $a_1 \to a_2 \to a_3$ ein nicht-terminierter Lauf für die Kripkestruktur und $a_1 \to a_2 \to a_3 \to b_3 \to b_2 \to b_1$ ein terminierter Lauf. Wegen $I = \{a_1\}$ ist $a_3 \rightsquigarrow b_1, b_3 \not\rightsquigarrow a_2$ und $Post^*(M) = \{a_i, b_i \mid i \in \mathbb{N}\}$. Im Beispiel wurden zudem die Transitionskanten mit dem Methodennamen einer rekursiven Methode $x()$ und dem zugehörigen Methodenende return annotiert. Die Methode $x()$ kann sich selbst nichtdeterministisch potentiell unendlich oft aufrufen und führt daher konzeptionell zu einer unvorhersagbaren Rekursionstiefe.*

In dieser Arbeit werden Kontrollflussgraphen gemäß [163] verwendet. Grundblöcke haben dort die Größe 1 (d.h. sie bestehen aus einer Anweisung).

Definition 3.1.2 (*Kontrollflussgraph*)
Der interprozedurale Kontrollflussgraph $G = (V, E)$ eines Programms *besteht aus den* Anweisungen *als Knoten $p_i \in V$ und der Nachfolgebeziehung $(p_1, p_2) \in E \subseteq V \times V$ als Kanten, wenn bei Ausführung **der Anweisung** am Knoten p_1 die* Nachfolgeanweisung *p_2 (in einem Schritt) erreicht werden kann (kurz $p_1 \xrightarrow{s}_E p_2$). $G' = (V, E')$ heißt* intraprozedural*, wenn E' die größte intraprozedurale Teilmenge der Kanten E von G ist, d.h. jedes $(p_1, p_2) \in E'$ weder ein Methodenaufruf (call) noch ein Methodenende (return) ist.*

Beispiel 3.1.2 (Kripkestruktur für Beispiel 1.1.2 auf Seite 10)
In Abbildung 3.2 ist die Kripkestruktur $M = (S, \to, I, L)$ zu sehen für Beispiel 1.1.2 auf Seite 10. Sie entsteht bei Abstraktion *von den Daten und berücksichtigt lediglich den Kontrollflussgraphen. $S = \{a_0, a_1, a_2, a_1b_0, a_1b_2, a_1b_3, a_1b_2b_0, a_1b_2b_2, a_1b_2b_3 \dots\}$ ist darin die unendliche Menge der Zustände mit der abgebildeten Transitionsfunktion $\to$ und $I = \{a_0\}$. $A = \{p, q\}$ ist die Menge der Merkmale und die Annotationsfunktion L gibt den Zuständen, die mit einem a_i enden, das Merkmal p ($L(...a_i) = \{p\}$) und den Zuständen, die mit b_i enden, das Merkmal q ($L(...b_i) = \{q\}$). In diesem Beispiel ist $a_0 \to a_1 \to a_1b_0$ ein nicht-terminierter Lauf für die Kripkestruktur und $a_0 \to a_1 \to a_1b_0 \to a_1b_1 \to a_1b_2 \to a_1b_3 \to a_2$ ein terminierter Lauf. Wegen $I = \{a_0\}$ ist $Post^*(M) = \bigcup_{0 \le i < 3} a_i \cup \bigcup_{0 \le i \le 3} a_1 b_2^* b_i$. Die Methode $b()$ kann sich selbst nichtdeterministisch potentiell unendlich oft aufrufen und führt daher konzeptionell zu einer unvorhersagbaren Rekursionstiefe.*

Wenn die Rekursionstiefe nichtdeterministisch ist wie im Beispiel oder (wie es auch z.B. typisch für Sortieralgorithmen ist) von Benutzereingaben abhängt, ist oft unklar, bis zu welcher Tiefe Methodenaufrufe in die Kripkestruktur modelliert werden sollten, um im Modell das reale Verhalten möglichst *präzise* abzubilden. Eine unendliche Kripkestruktur kann nicht explizit, d.h. durch Aufzählung der Zustände, beschrieben werden. So genannte Kellersysteme besitzen eine endliche Beschreibung und erklären *symbolisch* und damit kompakt große und *präzise* (unendliche) Kripkestrukturen wie diejenige aus dem Beispiel. Ein Kellersystem beschreibt den Graphen von *Konfigurationen* eines Kellerautomaten [245].

Definition 3.1.3 (*Kellersystem*)
$\mathcal{P} = (P, \Gamma, \hookrightarrow, I, L)$ heißt Kellersystem *(PDS), falls P eine endl. Menge von Zuständen, Γ eine endl. Menge (das Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen und $I \subseteq$*

$P \times \Gamma^*$ *eine Menge von* Initialkonfigurationen *sowie* $L : P \times \Gamma^* \to 2^A$ *eine Annotationsfunktion für* Konfigurationen *sind, wobei eine Menge an Merkmalen* A *gegebenen ist.* (p, w) *heißt* Konfiguration, *falls* $p \in P, w \in \Gamma^*$. (p, v) *heißt* Kopf *der* Konfiguration (p, vw) *(in Zeichen:* $Kopf((p, vw)) = (p, v)$*).* Konfigurationen *werden mittels* L *mit Merkmalen annotiert. Auf* Konfigurationen *wird die Transitionsrelation* $\hookrightarrow$ *erweitert zu* $\to \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ *mit* $(p, aw) \to (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$*. Die Gesamtheit an* Konfigurationen *heißt* Konfigurationenraum *und wird mit* $Konf(\mathcal{P})$ *bezeichnet.*

Informal ist ein Kellersystem ein Kellerautomat ohne Eingabe. Zur Unterscheidung der Zustände P des *PDS* M von den Zuständen der durch M beschriebenen Kripkestruktur dient der Begriff *Konfiguration.*

Lemma 3.1.1 (Kellersysteme beschreiben Kripestrukturen)
Sei $\mathcal{P} = (P, \Gamma, \hookrightarrow, I, L) \in$ PDS *ein Kellersystem. Dann ist* $\mathcal{S} = (P \times \Gamma^*, \to, I, L)$ *mit* $(p, aw) \to (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$ *eine Kripkestruktur (* $\mathcal{S} \in$ KS*).*

Beweis (Skizze)
Konfigurationen des *PDS* P werden wie im Beipsiel 3.1.2 als Zustände der durch P definierten Kripkestruktur interpretiert. Transitionenübergänge der Kripkestruktur ergeben sich aus der Transitionsfunktion auf *Konfigurationen* von P. Insgesamt stimmen die Typen der Definition, daher ist $\mathcal{S} \in KS$.

□

Bemerkung 3.1.1 *Begriffe wie* Lauf, Nachfolgezustand, Erreichbarkeit *und* Terminierung *für Kripkestrukturen übertragen sich damit entsprechend auf Kellersysteme. Eine Folge von Konfigurationen* $a_1 \to a_2 \to a_3 \to \cdots \to a_n$ *beginnend aus einer gegebenen* Initialkonfiguration $a_1 \in P \times \Gamma^*$*, heißt* Lauf. Konfiguration a_{i+1} *heißt dabei* Nachfolgekonfiguration *von* a_i*. Ein Lauf heißt* terminiert*, falls es keine* Nachfolgekonfiguration $b \in P \times \Gamma^*$ *dergestalt gibt, dass gilt* $a_1 \cdots \to a_n \to b$*. Ist eine Menge* I *von Initial- bzw.* Startkonfigurationen *gegeben, so heißt eine* Konfiguration b erreichbar *(in Zeichen* $I \rightsquigarrow b$ *bzw.* $\mathcal{P} \rightsquigarrow b$*), falls es eine* Initialkonfiguration $i \in I$ *derart gibt, dass ein Lauf* $i \to \cdots \to b$ *existiert. Existiert kein solcher Lauf, so ist* b nicht *erreichbar (in Zeichen* $I \not\rightsquigarrow b$*). Analog sind die Mengenschreibweisen* $C \rightsquigarrow B :\Leftrightarrow \exists c \in C : \exists b \in B : c \rightsquigarrow b$ *sowie* $c \rightsquigarrow B :\Leftrightarrow \exists b \in B : c \rightsquigarrow b$ *erlaubt. Ein Lauf* $a_1 \to a_2 \cdots \to a_n$ *heißt analog* essentiell *für* $B \subseteq P \times \Gamma^*$*, wenn gilt* $a_n \rightsquigarrow B$*. Die Menge aller in* $\mathcal{P}$ erreichbaren Konfigurationen *sei mit* $Post^*(\mathcal{P})$ *bezeichnet. Dabei ist* $Post^*(C) = \{k \mid k \in P \times \Gamma^*, C \rightsquigarrow k\}$ *und* $Pre^*(C) = \{k \mid k \in P \times \Gamma^*, k \rightsquigarrow C\}$ *für* Konfigurationenmengen $C \subseteq P \times \Gamma^*$*.*

Nun wird es mit der Einführung von *symbolisch*en Kellersystemen ermöglicht, auch Kellersysteme symbolisch und damit die zu Grunde liegende Kripkestruktur noch kompakter zu beschreiben.

3.2. Symbolisches Kellersystem

Ein *symbolisch*es Kellersystem S (SPDS) besteht aus einer Menge von globalen Variablen *Vgbl* und einer Menge von Prozeduren *Prz* (*Modellbestandteile*) mit Wertparamter (*Wertaufruf / Call-By-Value*). Jede Prozedur m besteht aus einer Menge lokaler sowie Parameter-Variablen ($Vlcl_m$ bzw. $Param_m$) und einer Liste von *Anweisungen* (Prozedurrumpf). Jede Variable hat - wie auch üblich bei höheren *Programmiersprachen* - einen endlichen *Wertebereich*[4].
SPDS ermöglichen eine kompakte, implizite Beschreibung von Kellersystemen. Transitionen werden nicht explizit als Menge von Kanten dargestellt, sondern ergeben sich anhand einer klar definierten Beschreibung. Die Begriffe und Definitionen übertragen sich von *PDS* auf *SPDS* in natürlicher Weise, da sich lediglich die Beschreibungsform des *PDS* ändert. In diesem Abschnitt wird eine operationale Sematik für *SPDS* definiert. Dazu wird beschrieben, wie die in dieser Arbeit verwendeten syntaktischen Konstrukte *Konfigurationenübergänge* (Zustandsübergänge der zu Grunde liegenden Kripkestruktur) erzeugen. Es wird erklärt, was unter einer *Konfiguration* für *SPDS* verstanden wird und wie Konfigurationenübergänge mittels Transitionsregeln des *PDS* zu interpretieren sind.

[4] D.h. der Typ ist beschränkt.

Quellcode 3.1: Javabeispiel 1.1.2 als *SPDS*.

```
void a() {                    void b(int i(4)) {
    int i(4);                     b0: i = i - 1;
a0: i = 10;                       b1: if (i > 0) goto b2;
a1: b(i);                         b3: return;
a2: return; }                     b2: b(i); goto b3; }
```

Um für ein *Programm* (gegeben als Quelltext) ein Kellersystem zu modellieren, wird jeder Programmzustand einschließlich potentiell unendlich vieler (rekursiver) Methodenaufrufe (die sog. **Aufrufhierarchie**) als *Konfiguration* eines Kellersystems kodiert. Es wird der Kontrollfluss eines *Programms* in Transitionen des *PDS* überführt (*symbolisch* in Form eines *SPDS* beschrieben). Die meisten typischen *Anweisungen* aus Hochsprachen lassen sich als SPDS-*Anweisungen* umsetzen[5]. Die Werte von *SPDS*-Variablen beschreiben dann Zustände bzw. Konfigurationen. Variablenwerte globaler Variablen werden in den Zuständen des *PDS* repräsentiert. Lokale Variablen zusammen mit dem aktuellen Ausführungspunkt hingegen werden durch Kellersymbole des *PDS* repräsentiert. Dies ermöglicht auf natürliche Weise Methodenaufrufe, indem Kellertupel (engl. stack frames) bei einem Methodenaufruf neu im Keller abgelegt und beim Methodenende vom Keller wieder entfernt werden. Dadurch bleibt der Wert lokaler Variablen und der fortzusetzende *Programmpunkt* im tiefen Keller unverändert erhalten. Zwar sind die Typen von Variablen in Hochsprachen i.d.R. ebenfalls beschränkt[6] (endlicher *Wertebereich*), jedoch führt eine Modellierung von vergleichsweise großen Typen wie 32 oder 64 Bit zu einem in der Praxis unhandlichem *SPDS*-Modell mit sehr großem *Konfigurationenraum*.
Folgendes Beispiel demonstriere die Syntax *symbolisch*er Kellersysteme.

Beispiel 3.2.1 (Beispiel 1.1.2 auf Seite 10 als *SPDS*)
In Quellcode *3.1 ist ein Beispiel eines* SPDS *abgebildet, welches mittels der* Anweisungen *das* Programmverhalten *aus dem Java*-Programm *aus Beispiel 1.1.2 auf Seite 10 modelliert (Abbildung 1.2 auf Seite 10). Optisch unterscheidet sich die Beschreibung kaum vom zugrunde liegendem Java*-Programm. *Die Java*-Anweisungen *werden auf SPDS*-Anweisungen *zurückgeführt, welche sehr ähnlich zu Java aussehen. Der wesentliche Unterschied liegt in der* Interpretation. *Prozeduren aus der Hochsprache werden direkt als Prozeduren des* SPDS *realisiert. Die Kontrollflusspunkte* $a0, a1, a2, b0, b1, b2$ *und* $b3$ *entsprechen dabei den Marken (Labels) im* SPDS. *Als Typ für die Variablen wurde* $bits(i) = 4$ *gewählt. Integer-Variablen haben in Java typischerweise 32 Bit. 4 Bit für die Variablen im* Beispielprogramm *genügen, um die Semantik exakt im* SPDS*-Modell nach zu bilden. Der Typ, d.h. die Bitgröße, einer Variablen wird bei* SPDS *in Klammern hinter die Typdefinition gesetzt.*

3.2.1. Syntax

Definition 3.2.1 (*symbolisches Kellersystem*)
Ein symbolisch*es Kellersystem (SPDS)* $S = (Vgbl, PrzDesc, init)$ *besteht aus einer* Definitionsliste *von globalen Variablen* $Vgbl$*, einer Menge an Prozedurbeschreibungen* $PrzDesc$*, welche eine Menge an Prozeduren* Prz *beschreibt sowie einer* Hauptprozedur $init \in Prz$*. Eine* Definitionsliste $D = [(\alpha_1, b_1), (\alpha_2, b_2), \dots (\alpha_n, b_n)]$ *ist eine Liste von disjunkten Variablennamen* α_i *mit Typdefinitionen* b_i*.* SPDS*-Variablen* $v \in Vgbl$ *haben (wie auch in gewöhnlichen Hochsprachen üblich) einen beschränkten Typ* $bits((\alpha_i, b_i)) := b_i \in \mathbb{N}_{>0}$*, welcher angibt, wie viele Bits zur Speicherung eines Wertes verwendet werden. Der Typ* $b_1 \in \mathbb{N}_{>0}$ *heißt* anpassbar *an einen Typ* $b_2 \in \mathbb{N}_{>0}$*, falls* $b_1 \leq b_2$*. Sind die Typen im Kontext unwichtig, so wird auch die Schreibweise* $\alpha_i \in D$ *verwendet. Eine Variable* v *kann die ganzzahligen Werte* 0 *bis* $2^{bits(v)} - 1$ *annehmen. Der Wert einer*

[5] Dabei sind allerdings wegen der unterschiedlichen Semantik von *Ausdrücken* teilweise Anpassungen bei der Übertragung in SPDS-*Anweisungen* nötig.
[6] Dies gilt auch für zusammengesetzte Datentypen wie Strukturen oder Klassen: z.B. BigINT in Java (wird vor dem Nutzer durch „Tricks“ verschleiert).

Variablen v wird durch die Interpretation $[\![v]\!]$ *bestimmt. Im Folgendem wird für den* Wertebereich $range(v) := \mathbb{N}_{\geq 0} \cap [0, 2^{bits(v)} - 1]$ *verwendet. Eine* Prozedurbeschreibung *besteht aus einer Signatur und einem Prozedurrumpf. Ein* Prozedurrumpf $Body(p)$ *einer Prozedur p besteht aus einer Liste von* Anweisungen $Body(p) = [s_1, s_2, \ldots, s_m]$, *wobei $Stats(S)$ die Menge aller* Anweisungen *von S beschreibt. Jede* Anweisung s *ist mit einer eindeutigen Marke l versehen ($l : s \in Stats(S)$). Es wird auch kurz „$l : s$“ $\in S$ geschrieben. Die Hauptprozedur charakterisiert die* Startkonfigurationen *des* PDS *mit unbelegten* SPDS-*Variablen. Eine* Signatur *sig ist ein Tupel* $sig = (p, Param_p, Vlcl_p)$ *mit einem* Prozedurnamen $p \in Prz$, *einer Definitionsliste von* Parametervariablen $Param_p$ *und einer Definitionsliste* lokaler Variablen $Vlcl_p$. *Die Variablennamen aus* $Vgbl, Vlcl_p$ *und* $Param_p$ *seien disjunkt und* $Vars(S) := Vgbl \cup (\bigcup_{p \in Prz} Vlcl_p) \cup (\bigcup_{p \in Prz} Param_p)$ *die Menge der globalen sowie lokalen Variablendefinitionen und der Prozedurparameterdefinitionen. Statt $Vars(S)$ wird auch kurz $Vars$ geschrieben, wenn S sich aus dem Kontext ergibt. Kommentare werden mit dem Zeichen # eingeleitet und gelten bis zum Zeilenende*[7].

Bemerkung 3.2.1 (Formalisierung von Beispiel 3.2.1)
In Beispiel 3.2.1 sind 2 Prozeduren a und b als SPDS $S = (Vgbl, PrzDesc, a)$ *beschrieben, wobei die Prozedur a als Hauptprozedur gewählt wurde,* $Vgbl = \emptyset$, $init = a$, $PrzDesc = \{(sig_a, rumpf_a), (sig_b, rumpf_b)\}$, $sig_a = (a, Param_a, Vlcl_a)$, $Param_a = \emptyset$, $Vlcl_a = \{i\}$ *mit* $bits(i) = 4$, $sig_b = (b,\ Param_b,\ Vlcl_b)$, $Param_b = \{i\}$ *mit* $bits(i) = 4$ *sowie* $Vlcl_b = \emptyset$ *ist. Die Prozedurrümpfe sind in* Quellcode 3.1 *abgebildet.*

Bemerkung 3.2.2 (Markenfreie SPDS-*Anweisungen*)
Wird eine Marke $l \in Marken(P)$ nicht als Sprungziel oder anderweitig benötigt (z.B. für den Erreichbarkeitstest *bei der* Modellprüfung*), so darf zur übersichtlicheren Beschreibung der SPDS-*Anweisung „$l : s$;“ *in dieser Arbeit die Marke l weg gelassen und statt dessen nur „s;“' geschrieben werden. Es wird dann implizit eine eindeutige Marke vergeben.*

Um SPDS-*Anweisungen* zu definieren werden *SPDS-Ausdrücke* verwendet. Daher werden nun zunächst *SPDS-Variablenbelegung*en und *SPDS-Ausdrücke* erklärt.

Definition 3.2.2 (SPDS-Variablenbelegung)
Eine Belegung env_V *von Variablen $v \in V$ mit Typ $bits(v)$ ist eine Abbildung von Variablen auf natürliche Zahlen* $env_V : V \to \mathbb{N}_{\geq 0}$, *so dass* $env_V(v) = [\![v]\!]_{env_V} \in range(v)$. *Die ausgezeichnete Belegung* $env_V = \bot$ *gibt an, dass die* Variablenbelegung *ungültig ist* [8]. *Weiter ist es möglich, für eine gegebene Belegung* $env_V \neq \bot$ *den Variablenwert einer Variable $x \in V$ durch einen neuen Wert $a \in range(x) \subset \mathbb{N}_{\geq 0}$ zu ersetzen. In Zeichen:*

$$env_V|_x^a(v) := \begin{cases} a & falls\ v = x \\ env_V(v) & falls\ v \neq x \end{cases}$$

Ist $a \notin range(x)$, so ist $env_V|_x^a = \bot$. Ist $x \notin V$, so ist $env_V|_x^a = env_V$. Variablenbelegung*en env_A, env_B (auch $A \cap B \neq \emptyset$ zulässig) können zu einer größeren* Variablenbelegung

$$env_{A,B}(v) := \begin{cases} env_B(v) & falls\ v \in B \\ env_A(v) & sonst\ (insb.\ v \in A \setminus B) \end{cases}$$

verbunden werden. Jeder Kopf einer Konfiguration z *eines* PDS $\mathcal{P}$ *(in Zeichen $Kopf(z)$) beschreibt eine* Variablenbelegung[9] env_V^z. *Die Menge aller* Variablenbelegung*en sei mit ENV bezeichnet.*

Für eine Prozedur m mit lokalen Variablen $Vlcl_m$ und Parametervariablen $Param_m$ wird im Folgenden auch statt $env_{Vlcl_m, Param_m}$ abkürzend env_{loc_m} mit $loc_m := Vlcl_m \cup Param_m$ geschrieben.

[7] Kommentare werden in dieser Arbeit genutzt, um darin Analyseergebnisse der *Modellanalyse*n übersichtlich darzustellen.
[8] z.B. nach Division mit 0
[9] Speicherung der Variableninhalte in den Kellersystemzuständen für globale Variablen bzw. in obersten Kellersymbolen für lokale Variablen.

Definition 3.2.3 (SPDS-Ausdrücke)
Es können mittels Konstanten $c \in \mathbb{N}_{\geq 0}$ *und den Operatoren* $Op = Op_1 \cup Op_2$, *wobei* $Op_1 = \{!\}$ *und* $Op_2 = \{-, *, /, \&, <\}$ *sowie den Variablen Vars eines* SPDS *arithmetische und logische (boolesche)* Ausdrücke *Expr gebildet werden. Dabei ist Expr die kleinste Menge mit folgenden Eigenschaften:*

- $\mathbb{Z} \subset Expr$
- $Vars \subset Expr$
- $(e \in Expr \wedge (\oplus \in Op_1)) \Rightarrow \oplus e \in Expr$
- $(e_1, e_2 \in Expr \wedge (\oplus \in Op_2)) \Rightarrow e_1 \oplus e_2 \in Expr$.

Tabelle 3.1.: Rang, Bedeutung und Assoziativität von *SPDS*-Operatoren ($a, b \in Expr$).

Operator	Rang	Assoziativität	Bedeutung bzw. *Interpretation*
$!a$	3	rechts	bitweise Negation von a (positive Dualzahlen)
$a - b$	6	rechts	Subtraktion (mathematisch)
$a * b$	5	links	Multiplikation (mathematisch)
a/b	5	links	Division ohne Rest, $[\![b]\!] = 0$ ergibt $\perp$ (undef.)
$a\&b$	10	links	bitweises Und (positive Dualzahlen)
$a < b$	8	links	Vergleich auf kleiner (Resultattyp hat 1 Bit)
$-a$	3	rechts	$0 - a$ (links: unär, rechts: binärer Operator)
$a + b$	6	rechts	$a - (0 - b)$
$a\%b$	5	links	$a - ((a/b) * b)$
$'a$	3	rechts	$\underbrace{2 * 2 * \cdots * 2}_{a \text{ mal}}$
$a << b$	7	links	$a * ('b)$
$a >> b$	7	links	$a/('b)$
$a\|b$	12	links	$!(!a\&b)$
$a\hat{}b$	11	links	$(a\&!b)\|(!a\&b)$
$a\&\&b$	13	links	$a\&b$ (boolesch)
$a\|\|b$	14	links	$a\|b$ (boolesch)
$a == b$	9	links	$!((a < b)\|(b < a))$
$a! = b$	9	links	$!(a == b)$
$a <= b$	8	links	$(a == b)\|(a < b)$
$a >= b$	8	links	$!(a < b)$
$a > b$	8	links	$!(a <= b)$

Ausdrücke *werden interpretiert durch eine* Interpretation*sfunktion* $[\![\cdot]\!]_{env_V} : Expr \to \mathbb{Z}$. *Die* Interpretation $[\![e]\!]_{env_V}$ *eines Ausdrucks* $e \in Expr$ *ist abhängig von einer* Variablenbelegung $env_V \in ENV$. *Die* Variablenbelegung env_V *kann weg gelassen werden, wenn die Variablenbelegung unwichtig oder aus dem Zusammenhang klar ist. Der Typ eines Ausdrucks* $e \in Expr$ *ist* $bits(e) = \lceil log_2(1 + [\![e]\!]_{env_V}) \rceil$ *und besteht aus den minimal nötigen Bits für die Dualzahlendarstellung, wobei* $bits(\perp) = \infty$. *Er ist abhängig von der Belegung der Variablen, wenn e keine Konstante darstellt. Nicht-Basis-Operatoren werden auf die Basisoperatoren Op zurückgeführt. Alle verwendbaren Operatoren, deren Ränge, Assoziativitäten und Bedeutung bzw.* Interpretation *mittels Basisoperatoren sind in Tabelle 3.1 zusammengefasst. Ein Operator bindet stärker, wenn sein Rang kleiner ist (d.h. es ist* $a + b * c = a + (b * c)$*). Ist ein Operand eines Operators* $\perp$ *(z.B. bei Divison mit* 0*), so ist das Resultat ebenso* $\perp$. *Es sind in dieser Arbeit alle Operatoren strikt und es gibt keine Ringarithmetik. Mittels einer* Variablenbelegung $env_V \in ENV$ *lassen sich* Ausdrücke $e \in Expr$ *über Variablen*

$V \subseteq Vars$ auswerten *zu (bzw. interpretieren als):*

$$[\![e]\!]_{env_V} := \begin{cases} e & \text{falls } e \in \mathbb{Z} \cup \{\bot\} \\ env_V(e) & \text{falls } e \in V \\ [\![e_1]\!]_{env_V} \oplus [\![e_2]\!]_{env_V} & \text{falls } e = e_1 \oplus e_2 \\ \oplus [\![e_1]\!]_{env_V} & \text{falls } e = \oplus e_1 \\ \bot & \text{sonst.} \end{cases}$$

Wird eine Variable $v \in Vars$ innerhalb eines Ausdrucks e verwendet, so ist $v \in e$. Analog sei für Teilausdrücke $e' \in Expr$ die Schreibweise $e' \in e$ definiert, falls e' in e vorkommt[10].

Bemerkung 3.2.3 *Man beachte, dass lediglich nichtnegative Variablenwerte berücksichtigt werden und dennoch* Ausdrücke *und deren Teilausdrücke einen negativen Wert annehmen dürfen. Der Operator-Rang und die Assoziativitäten entsprechen denen der* Programmiersprache *C.*

Definition 3.2.4 (SPDS-*Anweisungen*)
Sei $Sigs = \bigcup_{p \in Prz}(p, Param_p, Vlcl_p)$ eine Menge von Signaturen *mit paarweise verschiedenen Prozedurnamen und $(p, Param_p, Vlcl_p) \in Sigs$ eine gegebene Signatur. Ein* Prozedurrumpf *ist eine Liste von* Anweisungen *$[s_1, s_2, \ldots, s_m]$, die mittels eindeutiger Marken $l_i \in Marken$ (Sprungziele, auch mit $Marken(S)$ bezeichnet) markiert sind ($l_1 : s_1; l_2 : s_2; \ldots l_m : s_m;$). Für $e \in Expr, x, i_j \in Vars$ und $l \in Marken$ hat eine* SPDS-Anweisung *eine der folgenden Formen:*

(Z)	$x = e;$	Zuweisung
(C)	$p(x_1, x_2, \ldots, x_n);$	Prozeduraufruf
(R)	$return;$	Prozedurende
(G)	$if\ (e)\ goto\ l;$	Verzweigung

Zudem gibt es expliziten Nichtdeterminismus durch eine spezielle Funktion $choose : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, welche einen zufälligen ganzzahligen Wert im angegebenen Intervall $[a, b]$ liefert: $choose(a, b) \in [a, b] \cap \mathbb{Z}$. choose darf nicht in Ausdrücken *verwendet werden und ist nur als* Zuweisung *der Form $x = choose(a, b)$; an eine globale Variable x zugelassen und wird in der weiteren Arbeit als* Prozeduraufruf *mit* Seiteneffekt *auf die globale Variable x aufgefasst.*

Kurz wird nun die informelle Bedeutung der SPDS-*Anweisungen* erläutert, um danach die formale Semantik im anschließenden Abschnitt zu definieren.
Die Zuweisung (Z) weist der Variablen x den Wert $[\![e]\!]$ zu, falls der Typ vom Ausdruck e an den Typ von x angepasst werden kann, d.h. $[\![e]\!] \in range(x)$. Andernfalls (wenn $[\![e]\!] \notin range(x)$) gibt es keine *Nachfolgekonfiguration* im *SPDS*. Dies trifft für Ausnahmen wie Division mit 0 und Typinkompatibilitäten zu. Auch dann ist $[\![e]\!] \notin range(x)$ und es gibt keine *Nachfolgekonfiguration* im *SPDS*.
Der Prozeduraufruf (C) ruft eine Prozedur p auf und übergibt per Wertaufruf (Call-By-Value) die Prozedurparameter $Param_p = [p_1, p_2, \ldots, p_n]$ mit $[\![p_j]\!] = [\![x_j]\!]$, falls die Typen anpassbar sind, d.h. $\forall j : range(x_j) \leq range(p_j)$ und $n = |Param_p|$. Andernfalls handelt es sich nicht um ein gültiges *SPDS* (Typfehler). Lokale Variablen $v \in Vlcl_p$ einer Prozedur p sind bei Prozedureintritt uninitialisiert, d.h. sie besitzen einen willkürlichen Wert innerhalb ihres *Wertebereiches*: $v = choose(0, 2^{bits(v)} - 1)$. Analog sind auch globale Variablen bei Eintritt in die Hauptprozedur uninitialisiert.
Die *Anweisung Prozedurende* (R) beendet die aktuelle Prozedur und kehrt zum Aufrufer zurück (falls vorhanden, d.h. nicht erster Aufruf der Hauptprozedur).
Die *Verzweigung* (G) setzt die Ausführung ab der Marke l (Sprungziel) fort, falls $[\![e]\!] = 1 \in \{0, 1\}$. Das Sprungziel l muss für ein gültiges *SPDS* innerhalb der gleichen Prozedur q liegen. D.h. für diese Arbeit sind nur intraprozedurale Verzweigungen erlaubt[11]. Alle Variablen behalten bei der Verzweigung ihren Wert.

[10] Hier ist nicht das syntaktische Auftreten gemeint, sondern vielmehr das Muster in einem abstrakten Syntaxbaum, so dass durch Klammern die Prioritäten sowie Assoziativitäten berücksichtigt sind.
[11] Eine Erweiterung auf interprozedurale Verzweigungen ist in Anlage „Verallgemeinerung auf Interprozedurale Verzweigungen“ auf beiliegender CD beschrieben.

Ist bei der Verzweigung (G) hingegen $[\![e]\!] = 0$, so wird wie im Fall der *Zuweisung* (Z) mit der nächsten *Anweisung* in der Liste des Prozedurrumpfes fortgesetzt. Ein Wert $[\![e]\!] \notin \{0,1\}$ ist für die Bedingung in der Verzweigung (G) nicht erlaubt.
Bei einem *Prozeduraufruf* (C) an einer Marke $l \in$ *Marken* wird mit der ersten *Anweisung* der aufgerufenen Prozedur fortgesetzt. Wird diese Prozedur beendet (R), so setzt die Ausführung mit der *Anweisung* in der Liste nach l fort. Die letzte *Anweisung* s_m eines Prozedurrumpfes ist stets *return*. SPDS-*Anweisungen* sind immer mit einer Marke $l \in$ *Marken* markiert. Werden sie in der Beschreibung zur besseren Lesbarkeit weggelassen, so ergeben sich implizit definierte Marken für diese Anweisungen.
Hiermit wurden die grundlegenden *Anweisungen* und deren Bedeutung für *SPDS* erklärt. Nun wird spezifiziert, wie diese *Anweisungen* die Transitionsregeln eines *PDS* bestimmen. Darauf folgen dann aufbauend abkürzende Schreibweisen, welche die Beschreibung von *SPDS* erleichtern.

3.2.2. Semantik

Im Folgendem wird formal die Semantik von *SPDS*-Beschreibungen errläutert. Dazu wird erklärt, wie die *SPDS*-Anweisungen mittels *PDS*-Konfigurationen zu interpretieren sind. Hierzu wird eine formale *Interpretation* $\gamma[\![\cdot]\!] : SPDS \to PDS$ definiert. Zur Veranschaulichung diene zunächst wieder Beispiel 3.2.1 auf Seite 24.

Beispiel 3.2.2 (*Interpretation* von Beispiel 3.2.1 auf Seite 24 als *PDS*)
Das modellierte SPDS *aus Beispiel 3.2.1 auf Seite 24* $S = (Vgbl, PrzDesc, a)$ *wird durch* Interpretation *der SPDS-*Anweisungen *zu dem* PDS $\gamma[\![S]\!] = (P, \Gamma, \hookrightarrow, I, L)$ *mit* $P = \{p\}$, $\Gamma = \{a_0, a_1, a_2, b_0, b_1, b_2, b_3\} \times \{0,1\}^4$ *und* $I = \{(p, (a_0, x_1x_2x_3x_4)) \mid x_i \in \{0,1\}\}$. *Die Belegung env lokaler Variablen von Methoden (einschließlich Parametervariablen) werden als Bitvektoren im Kelleralphabet zusammen mit den Marken bzw. Kontrollpunkten realisiert. Im Beispiel werden die lokale Variable i aus der Methode a sowie der Parameter i aus der Methode b derartig als Bitvektoren der Länge* 4 *(*$bits(i) = 4$*) realisiert.*
Die Transitionen ergeben sich dann als:

$$
\begin{aligned}
\{(p,(a_0,x_1x_2x_3x_4)) &\hookrightarrow (p,(a_1,0101)) \mid x_i \in \{0,1\}\} \cup \\
\{(p,(a_1,x_1x_2x_3x_4)) &\hookrightarrow (p,(b_0,x_1x_2x_3x_4)(a_2,x_1x_2x_3x_4)) \mid x_i \in \{0,1\}\} \cup \\
\{(p,(a_2,x_1x_2x_3x_4)) &\hookrightarrow (p,\varepsilon) \mid x_i \in \{0,1\}\} \cup \\
\\
\{(p,(b_0,x_1x_2x_3x_4)) &\hookrightarrow (p,(b_1,y_1y_2y_3y_4)) \mid x_i, y_i \in \{0,1\}, \\
&\qquad \left(\sum_{i=0}^{3} x_{i+1}2^i\right) - 1 = \sum_{i=0}^{3} y_{i+1}2^i\} \cup \\
\{(p,(b_1,x_1x_2x_3x_4)) &\hookrightarrow (p,(b_2,x_1x_2x_3x_4)) \mid x_i \in \{0,1\}, \sum_{i=0}^{3} x_{i+1}2^i > 0\} \cup \\
\{(p,(b_1,0000)) &\hookrightarrow (p,(b_3,0000))\} \cup \\
\{(p,(b_2,x_1x_2x_3x_4)) &\hookrightarrow (p,(b_0,x_1x_2x_3x_4)(b_3,x_1x_2x_3x_4)) \mid x_i \in \{0,1\}\} \cup \\
\{(p,(b_3,x_1x_2x_3x_4)) &\hookrightarrow (p,\varepsilon) \mid x_i \in \{0,1\}\}.
\end{aligned}
$$

So gibt z.B. die erste Menge an Transitionen $\{(p,(a_0,x_1x_2x_3x_4)) \hookrightarrow (p,(a_1,0101)) \mid x_i \in \{0,1\}\}$ *an, dass am Kontrollpunkt* a_0 *sämtliche zuvor gültigen* Variablenbelegung*en durch die* Variablenbelegung $i = 10$ *(dezimal) d.h.* 1010 *(dual) ersetzt wird. Dies entspricht der* Zuweisung *an die Variable i in der Prozedur a mit dem Wert* 10 *an Marke* a_0 *im zugehörigem* SPDS *(Beispiel 3.2.1 auf Seite 24).*
Bei Methodenaufrufen wird ein neues oberstes Kellersymbol auf den Keller gelegt. Die lokalen Variablen werden entsprechend der Parameteraudrücke vom Aufruf belegt. Die Marken zum Fortsetzten des Kontrollflusses werden entsprechend angepasst. In der aufrufenden Prozedur ist dies die nachfolgende Marke a_2 *und in der aufgerufenen Prozedur die Anfangsmarke* b_0*. Dies führt im Beispiel*

zu der Transitionenmenge $\{(p,(a_1,x_1x_2x_3x_4)) \hookrightarrow (p,(b_0,x_1x_2x_3x_4)(a_2,x_1x_2x_3x_4)) \mid x_i \in \{0,1\}\}$ *für beliebige* Variablenbelegung*en. Auf diese Weise wird die gesamte Aufrufhierarchie direkt im Keller repräsentiert.*
Endet eine Prozedur (Schlüsselwort return), so wird das oberste Kellersymbol vom Keller für alle Variablenbelegung*en entfernt, da die* Variablenbelegung*en lokaler Variablen nicht mehr wichtig sind. Im Beispiel führt dies zu den Trasitionenmengen* $\{(p,(a_2,x_1x_2x_3x_4)) \hookrightarrow (p,\varepsilon) \mid x_i \in \{0,1\}\}$ *sowie* $\{(p,(b_3,x_1x_2x_3x_4)) \hookrightarrow (p,\varepsilon) \mid x_i \in \{0,1\}\}$.
Globale Variablen (in Java bzw. C++ Klassenvariablen), die Halde (engl. Heap) sowie Ausnahmen (engl. Exceptions) können mit Hilfe der Zustände des Kellersystems beschrieben werden. Im Beispiel werden weder globaler Speicher noch Ausnahmen verwendet, so dass hier ein einziger Zustand $p \in P$ *genügt.*
Insgesamt wurden damit für das Beispiel PDS $16+16+16+15+15+1+16+16 = 111$ *Transitionen definiert. Es gibt insgesamt* $7*16 = 112$ *Köpfe und einen globalen Zustand* $P = \{p\}$*, da es keine globalen Variablen gibt. Der* Konfigurationenraum *ist hingegen unendlich wegen potentieller rekursiver Aufrufe. Sämtliche* Variablenbelegung*en und Rekursionsaufrufe sowie* erreichbare Programmpunkte *sind 1:1 nachgebildet.*
PDS *verfügen im Gegensatz zu* SPDS *über eine Annotationsfunktion L. Zur Definition von L werden sog. Annotationsausdrücke* $AExpr \supseteq Expr$ *genutzt, welche eine Erweiterung von* SPDS-Ausdrücke*n darstellen. Merkmale A für ein* PDS $\mathcal{P}$ *werden durch solche Annotationsausdrücke dargestellt. Die Annotationsfunktion L für das Beispiel wird für* Konfigurationen $s \in P \times \Gamma^*$ *definiert als die Menge aller erfüllter Annotationsausdrücke AExpr, d.h.* $L(s) := \{e \in AExpr \mid [\![e]\!]_{env^s} = 1\}$*. So ist z.B. „*$x <= 10$*“* $\in L((p,(a_1,0101)))$ *und* $a, a_1 \in L((p,(a_1,0101)))$*, aber es ist* $b \notin L((p,(a_1,0101)))$.

Definition 3.2.5 (*Annotationsausdrücke*)
Annotationsausdrücke werden durch eine boolesche Algebra über arithmetische Ausdrücke*, Marken und Prozeduren gebildet (Syntax):*

$$\begin{aligned} \{e \in Expr \mid env \in ENV, [\![e]\!]_{env} \in \{0,1\}\} &\subseteq AExpr \\ Marken(S) &\subseteq AExpr \\ Prz(S) &\subseteq AExpr \\ \forall a,b \in AExpr &: a \wedge b \in AExpr \\ \forall a \in AExpr &: \neg a \in AExpr \end{aligned}$$

Die Semantik eines Annotationsausdrucks $e \in AExpr$ *für eine* Konfiguration s *eines* PDS $\mathcal{P}$ *mit* $Marken(Kopf(S)) = \{l\}$ *ist:*

$$[\![x]\!]_s := \begin{cases} [\![x]\!]_{env^s} & falls\ x \in Expr \\ 1 & falls\ x = l \\ 0 & falls\ x \in Marken(S) \setminus \{l\} \\ 1 & falls\ x \in Prz(S) \wedge l \in Marken(x) \\ 0 & falls\ x \in Prz(S) \wedge l \notin Marken(x) \\ [\![a]\!]_s \wedge [\![b]\!]_s & falls\ x = a \wedge b \\ \neg[\![a]\!]_s & falls\ x = \neg a \end{cases} .$$

Der Operator $\vee$ *der booleschen Algebra wird als* $[\![a \vee b]\!]_s := \neg(\neg[\![a]\!]_s \wedge \neg[\![b]\!]_s)$ *interpretiert.*

$Marken(m)$ sei die Menge von Marken (Sprungzielen) innerhalb der Prozedur m, so dass $Marken(S) \supseteq \bigcup_{m \in Prozeduren} Marken(m)$. Es ist $Marken(s) := \{l\}$ für eine SPDS-*Anweisung* $s =$ „$l : z$“ $\in Stats(S)$.
Mit diesen Mitteln kann nun die formale Semantik eines *SPDS* erklärt werden.

Die *Semantik* eines *SPDS* $S = (Vgbl, PrzDesc, init)$ ist ein *PDS* $\gamma[\![S]\!] = (\Sigma_{Vgbl}, \Gamma, \hookrightarrow, I, L)$, das wie folgt durch S definiert ist. Die Zustände Σ_{Vgbl} von P entsprechen sämtlichen *Variablenbelegung*en globaler *SPDS*-Variablen:

$$\Sigma_{Vgbl} := \{env_{Vgbl} \mid env_{Vgbl} \in ENV\}.$$

Das Kelleralphabet Γ besteht aus der Menge aller Kellertupel (stack frames):

$$\Gamma := \{(l, env_{loc_m}) \mid m \in Prz; l \in Marken(m), env_{loc_m} \in ENV\}.$$

Γ umfasst sämtliche *Variablenbelegung*en von lokalen und Parametervariablen von Prozeduren sowie einer speziellen aktuellen Marke $l \in Marken$. Die Marke l gibt dabei an, mit welcher SPDS-*Anweisung* ein aktueller Lauf fortzusetzen ist. Eine SPDS-*Anweisung* $l : s$ an einer Marke l überführt *Konfigurationen* des *PDS* $\gamma[\![S]\!]$ in Nachfolgekonfigurationen. Dieser Effekt kann mittels Transitionsregeln aus $\gamma[\![S]\!]$ wie folgt beschrieben werden. *Initialkonfigurationen* von $\gamma[\![S]\!]$ sind:

$$I := \{(env_{Vgbl}, (l_{init}, env_{loc_{init}})) \mid env_{Vgbl} \in \Sigma_{Vgbl}, (l_{init}, env_{loc_{init}}) \in \Gamma\},$$

wobei l_{init} die erste Marke der Hauptprozedur *init* ist.

Definition 3.2.6 (*symbolische Konfiguration*)
Eine symbolische Konfiguration $s = (env_V, l)$ *mit* $V \subseteq Vars$ *besteht aus einer* Variablenbelegung env_V *und einer* SPDS-*Marke* $l \in Marken(S)$. *Die Menge aller* symbolisch*en* Konfigurationen *wird mit* $SymKonf(S)$ *bezeichnet. Eine* symbolische Konfiguration $s = (uenv_V, l)$ *beschreibt die Menge an* PDS-*Konfigurationen*[12] $Konf(s) :=$

$$\{(env_0, (l, env_1)(l_2, env_2)\dots) \in \Sigma_{Vgbl} \times \Gamma^* \mid \forall x \in V : \exists i \in \{0, 1\} : [\![x]\!]_{uenv_V} = [\![x]\!]_{env_i}\}.$$

Es ist $Marken(s) := \{l\}$ *für eine* symbolisch*e* Konfiguration $s = (env_V, l) \in SymKonf(S)$.

Für eine derartige *Konfiguration* s mit der Form $s = (env_0, (l, env_1)(l_2, env_2)\dots) \in \Sigma_{Vgbl} \times \Gamma^*$ sei die Schreibweise zum Ändern einer *Variablenbelegung* der Variable x (lokale oder globale Variable) auf den Wert $c \in range(x) \cup \{\bot\}$ wie folgt übertragen:

$$(env_0, (l, env_1)(l_2, env_2)\dots)|_x^c := (env'_0, (l, env'_1)(l_2, env'_2)\dots) \tag{3.1}$$

wobei ein minimales $k \geq 0$ existiert (erstes Auftreten von x), so dass $[\![x]\!]_{env_k} \neq \bot$ und für jedes $i \geq 0$ gilt:

$$env'_i := \begin{cases} env_i|_x^c & falls\ i = k \\ env_i & sonst. \end{cases}$$

Die Transitionsregeln $\hookrightarrow$: $(\Sigma_{Vgbl} \times \Gamma) \times (\Sigma_{Vgbl} \times \Gamma^*)$ des durch das *SPDS* S beschriebenen *PDS* $\gamma[\![S]\!]$ sind dann die Gesamtheit der an durch die SPDS-*Anweisungen* der entsprechenden Marken definierten Transitionsrelationen $\hookrightarrow_l$ (in Zeichen $\hookrightarrow := \bigcup_{l \in Marken} \hookrightarrow_l$). Eine Transitionsrelation $\hookrightarrow_l$ an einer Marke l in der Prozedur $m = Prz(l)$ ist wie folgt definiert. Hierzu sei $env := env_{Vgbl, loc_m}$ und $next(l)$ mit $next : Marken \to Marken$ die Marke der Nachfolge-*Anweisung* im Prozedurrumpf (Liste von SPDS-*Anweisungen*) von m.

(Z) *Zuweisung* $l : x = e$;
$\hookrightarrow_l := \{ \quad (env_{Vgbl}, (l, env_{loc_m})) \hookrightarrow (env_{Vgbl}|_x^{[\![e]\!]_{env}}, (next(l), env_{loc_m}|_x^{[\![e]\!]_{env}}))$
$\quad \mid [\![e]\!]_{env} \in range(x), env_{Vgbl} \in ENV, env_{loc_m} \in ENV\}$.
Lokale und Parametervariablen $x \in loc_m$ haben Auswirkung auf env_{loc_m} und ändern entsprechend deren Wertbelegungen. Globale Variablen $x \in Vgbl$ haben Auswirkung auf env_{Vgbl}.

env enthält keine Variable $v \in Vgbl \cup loc_m$ mit Wert $[\![v]\!]_{env} \notin range(v)$ (abgesehen von $\bot$) und daher auch $env_{loc_m}|_x^{[\![e]\!]_{env}}$ sowie $env_{Vgbl}|_x^{[\![e]\!]_{env}}$ nicht.

Ein Lauf von $\gamma[\![S]\!]$ endet demnach an der *Zuweisung*, falls der Typ von e nicht anpassbar ist an den Typ von x. D.h. es muss gelten $bits(e) \leq bits(x)$ bzw. $[\![e]\!]_{env} \in range(x)$. Dies schließt z.B. auch Division mit Null ein, da $[\![e/0]\!]_{env} = \bot \notin range(x)$.

[12] Man beachte, dass Variablennamen (lokal/global) disjunkt sind: $Vgbl \cap loc_m = \emptyset$.

(C) *Prozeduraufruf* $l : p(x_1, x_2, \ldots, x_n)$;
$\hookrightarrow_l := \{(env_{Vgbl}, (l, env_{loc_m})) \hookrightarrow (env_{Vgbl}, (l_p, new_{loc_p}) \cdot (next(l), env_{loc_m})) \mid env_{Vgbl} \in ENV, env_{loc_m} \in ENV, new_{Vlcl_p} \in ENV, \forall i : [\![x_i]\!]_{env} \in range(x_i)\}$.
Lokale und Parametervariablen env_{loc_m} der aufrufenden Prozedur bleiben erhalten. Neue lokale Variablen new_{Vlcl_p} und Parametervariablen new_{Param_p} sowie die Anfangsmarke l_p für die aufgerufene Prozedur p werden neu im Keller als zusätzliches Kellersymbol abgespeichert (push). l_p ist dabei die erste Marke im Prozedurrumpf von p. Und $new_{loc_p} = new_{Vlcl_p, Param_p} \in ENV$ sind dabei neue *Variablenbelegung*en, wobei $new_{Vlcl_p} \in ENV$ beliebig gewählt und $new_{Param_p} \in ENV$ fest definiert sind. Die Prozedurparameter $p_i \in Param_p$ besitzen für $i = 1 \ldots n$ den Wert $[\![p_i]\!]_{new_{loc_p}} = [\![x_i]\!]_{env}$. D.h. es ist

$$new_{loc_p}(v) = \begin{cases} [\![x_i]\!]_{env} & falls\ v = x_i \in Param_p \\ choose(0, 2^{bits(v)} - 1) & sonst. \end{cases}$$

Wegen $\forall i : bits(x_i) \leq bits(p_i)$ gibt es stets eine Nachfolgekonfiguration, so dass hier kein Lauf terminiert im Gegensatz zur *Zuweisung*.

(R) *Prozedurende* $l : return$;
$\hookrightarrow_l := \{(env_{Vgbl}, (l, env_{loc_m})) \hookrightarrow (env_{Vgbl}, \varepsilon) \mid env_{Vgbl}, env_{loc_m} \in ENV\}$.
$\varepsilon \in \Gamma^0$ ist das leere Wort. Lokale und Parametervariablen sowie die aktuelle Marke der aufgerufenen Prozedur werden vom Keller entfernt. Die Bearbeitung eines aktuellen Laufs wird automatisch fortgesetzt in der aufgerufenen Prozedur (gemäß tieferem Kellerinhalt).

(G) *Verzweigung* $l : if\ (e)\ goto\ L$;
$\hookrightarrow_l := \{(env_{Vgbl}, (l, env_{loc_m})) \hookrightarrow (env_{Vgbl}, (l', env_{loc_m})) \mid [\![e]\!]_{env} \in \{0, 1\}, env_{Vgbl}, env_{loc_m} \in ENV\}$.
Ist die Bedingung e erfüllt, so wird ein bedingter Sprung an die intraprozedurale Marke $L \in Marken(m)$ durchgeführt. D.h. es ist

$$l' = \begin{cases} L & falls\ [\![e]\!]_{env} = 1 \\ next(l) & falls\ [\![e]\!]_{env} = 0. \end{cases}$$

Nur $[\![e]\!]_{env} \in \{0, 1\}$ ist erlaubt.

Damit wurden die *Anweisungen* für *SPDS* mittels *Konfigurationenübergänge* eines *PDS* erklärt. Im Anhang A ab Seite 133 wurden dazu abkürzende Schreibweisen für *SPDS* definiert, welche einerseits die Beschreibung von *SPDS* erleichtern und andererseits spätere Experimente mit der Modellsprache Remopla des gewählten Modellprüfers Moped erlauben. Diese Abkürzungen erweitern die SPDS-*Anweisungen* zu SPDS'-*Anweisungen* und werden im Folgenden zur verständlicheren und kompakteren Darstellung der Beispiele verwendet.

3.2.3. Beschränkungen

Ein *PDS* beschreibt den Graphen von *Konfigurationen* eines Kellerautomaten[13] [245]. Daher sind neben *PDS* auch *SPDS* nicht turingmächtig [245, 114]. Durch *Abstraktion* von turingmächtigen Hochsprachen auf nichtturingmächtige *SPDS* geht Präzision für die *Modellprüfung* verloren. Wie beschrieben, entstehen bei einer *konservativen* Approximation *False Negatives*. Durch verschiedene Beschränkungen (endlicher globaler Speicher, Wertübergabe bei Methodenaufrufen) werden *Programme* (wie Beispiel 3.2.1 auf Seite 24) als *SPDS* modellierbar [70]. Variablen und damit auch Zeiger einer höheren *Programmiersprache* haben oft einen beschränkten Typ. Daher ist der adressierbare Speicher in der Praxis üblicherweise beschränkt. Endlicher globaler Speicher ist daher in der Praxis keine wesentliche Einschränkung. Dennoch erlauben *SPDS*, wie auch in höheren *Programmiersprachen* auch weiterhin beliebig viel Speicher durch immer tiefere Rekursion, so dass *Konfigurationen* unbeschränkt viele *Variablenbelegung*en speichern können.

[13] Je nach Kellerautomaten-Eingabe sind verschiedene Läufe möglich, welche durch das *PDS* definiert sind.

Weitere rein konzeptionelle Beschränkungen, welche aber im wesentlichen erst in den Experimenten von Bedeutung werden, sind die Berücksichtigung negativer Zahlen, Gleitkommazahlen, komplexe Zahlen sowie Zeichenketten. Diese können durch Abbildung auf reguläre *SPDS*-Variablen zurück geführt werden. Gleitkommazahlen würden dann z.B. via Mantisse und Exponent in einer Struktur gespeichert und per Methodenaufrufe arithemtische Operationen erlauben. Analoges gilt für alle anderen nicht als *SPDS*-Basisdatentypen vorhandenen Datentypen.
Hochsprachen besitzen gelegentlich Ringarithmetik. *SPDS* hingegen besitzen keine Nachfolgekonfiguration, falls der Wert eines ausgewerteten Ausdrucks zu groß für den Variablentyp ist. Diese Semantik gilt es, bei der *Abstraktion* zum *SPDS* zu berücksichtigen[14]. Analoges gilt für Ausnahmen.

Bemerkung 3.2.4 *Unbeschränkte Parallelität in* SPDS *führt wegen gemeinsamen globalen Speichers zur Unentscheidbarkeit der* Modellprüfung *(turingmächtig wegen modellierbaren unbeschränkten Zählern [201])*[15].

3.3. Modellprüfung von Kellersystemen

Bisher wurde erklärt, was *SPDS* sind und wie sie definieren wurden. In diesem Abschnitt wird nun kurz erläutert, wie das Verhalten von Kellersysteme analysiert werden kann, was *Modellprüfung* auf *SPDS* bedeutet und wie Modelle auf Korrektheit geprüft werden können.

3.3.1. Analyse des Verhaltens von Kellersystemen

Was sind Modellanalysen? Durch eine automatische statische Analyse von Modellen wie z.B. Kellersystemen können Aussagen zum Verhalten des Modells getroffen werden. Solche Analysen werden im Folgenden als *Modellanalyse*n bezeichnet. Sie berechnen statisch, d.h. ohne Simulation bzw. Ausführung des Modells, Aussagen bzw. Informationen der Form, dass z.B. eine Variable v stets nur Werte in einem gewissen Bereich bzw. Intervall $[a, b]$ annimmt. Das Intervall $[a, b]$ wird dabei als *abstrakte Information* für die Variable v bezeichnet. Und eine *Modellanalyse* kann für alle Variablen abstrakte Informationen in Form von Funktionen $\psi : Vars \to \mathbb{Z} \times \mathbb{Z}$ bestimmen. Eine Menge AI, die durch *Abstraktion* des Modellverhaltens gewonnen wurde, heißt *abstrakte Information*. abstrakte Informationen sind vergleichbar zu Merkmalen (siehe Abschnitt 3.1 ab Seite 21), jedoch im Gegensatz dazu nicht auf endliche Mengen und auch nicht auf Situationen bzw. *Konfigurationen* beschränkt. *Abstrakte Informationen* können für Situationen, Zustände, Variablen etc. berechnet werden mittels einer *Modellanalyse*.

Definition 3.3.1 (*Modellanalyse*)
Sei ein Modell $M \in$ KS$\cup$PDS$\cup$SPDS$\cup$SPDS$'$ *gegeben und* $N \subseteq M$ *ein Teil der Modellbeschreibung. Eine Funktion* $\Omega_M : 2^M \to AI$ *heißt* Modellanalyse*, welche zu Modellen* M *und* Modellbestandteilen N *abstrakte Informationen* $\Omega_M(N)$ *berechnet.*

Von Programmanalysen zu Modellanalysen Für Hochsprachen werden *Programmanalysen* verwendet, um Aussagen zum Programmverhalten zu gewinnen. Gängige *Programmanalysen* lassen sich auf *SPDS* übertragen, was in dieser Arbeit im Laufe der folgenden Abschnitte gezeigt wird. Anpassungen der *Programmanalysen* zu *Modellanalyse*n für *SPDS* sind nötig wegen verschiedener Konstrukte, die es sonst nicht im *Programmiersprachenumfeld* gibt (darunter *temporale Formel*n und insbesondere nicht in Kombination). Dazu gehören u.a. existenzielle ($\exists$) und allquantifizierte ($\forall$) Anweisungen, expliziter Nichtdeterminismus (Funktion *choose* bzw. Schlüsselwort *undef*) sowie *Parallelzuweisung*en[16] und Mehrfachverzweigung[17].
Im Zusammenhang mit *Programmanalysen* werden in der Praxis gern Programmanalysegeneratoren verwendet. Eine Übersicht zu *Programmanalysegeneratoren* findet sich in der Dissertation

[14] Dies kann wie beim Tool *JMoped* geschehen durch spezielle bedingte Abfragen. Diese Verbesserung in JMoped ist neben weiteren begründet durch eigene Fehlermeldungen an die entsprechenden Entwickler.
[15] Eine Form der beschränkten Parallelität für *SPDS* wurde in der Anlage auf beiliegender CD erläutert.
[16] Ein Variablentausch ist sehr einfach möglich: $x = y, y = x;$.
[17] ggf. auch interprozedurale Sprünge (siehe Anlage auf beiliegender CD)

von Martin [87]. I.d.R. wird dabei mit einem Worklist-Algorithmus [164] eine sog. Fixpunktiteration durchgeführt, welche aus einer durchzuführenden Datenflussanalyse und einem gegebenen *Programm* gewonnen wurde. Einige der effizientesten Fixpunktalgorithmen zum Lösen von Datenflussanalysen basieren auf dem Worklist-Ansatz [49]. In den Experimenten kommt daher auch ein Fixpunktverfahren zum Einsatz. Einen guten Einblick in dieses Thema der statischen *Programmanalysen* liefert z.B. [163]. Darin definierte Begriffe wie *Verband* (engl. lattice) usw. werden als gegeben und bekannt angenommen.

Einsatz von Modellanalysen. *Programmanalysen* werden meist nur verwendet, um einfache Eigenschaften zu bestimmen. Für *Modellanalyse*n gilt dies ebenso. Sie sind i.d.R. sehr effizient wegen der *Abstraktion* vom realen Verhalten. *Modellanalyse*n können konzeptionell auf SPDS-*Anweisungen* operieren, aber auch auf den abkürzenden Schreibweisen (SPDS*'-Anweisungen*). Werden abkürzende Schreibweisen dabei zuvor auf SPDS-*Anweisungen* zurückgeführt, so führt dies i.d.R. zu größeren Modellen mit größeren Zustands- bzw. *Konfigurationenraum*. Auch werden die Analyseergebnisse ungenauer, da spezielle Eigenschaften von SPDS*'-Anweisungen* nicht einfach ausgenutzt werden können. Besser werden die Analyseergebnisse, wenn abkürzende Schreibweisen direkt durch eine Modellanalse behandelt werden. Aus diesem Grund finden sich in meinen definierten *Modellanalyse*n aus den späteren Kapiteln auch immer wieder abkürzende Schreibweisen, welche direkt interpretiert werden.

Eigenschaften von Modellanalysen Begriffe für Eigenschaften, welche für *Programmanalysen* bereits etabliert sind, übertragen sich in natürlicher Weise auf *Modellanalyse*n.
Für *Modellprüfung* ist wichtig, dass die abstrakten Informationen *konservativ* sind. Für das vorige Beispiel (mit den Intervallinformationen gegeben durch die abstrakte Information ψ) bedeutet *konservativ*, dass in jedem möglichen realen Lauf des Modells und in jeder Situation mit *Variablenbelegung* env gelten wird: $[\![v]\!]_{env} \in [a, b]$, wobei $\psi(v) = (a, b)$. Wegen der *Abstraktion* vom Modellverhalten während der Analyse kann es jedoch auch vorkommen, dass sogar stärker stets $[\![v]\!]_{env} \in [a+1, b]$ gilt. Die Analyse war dann nicht in der Lage, dies zu bestimmen und wird daher als *unpräzise* bezeichnet. Aufgrund solcher *Abstraktion* des Verhaltens sind die Ergebnisse von *Programmanalysen* denen der *Modellprüfung* in Präzision weit unterlegen. Daher besitzen Techniken wie *Programmanalysen* weniger Bedeutung bei der Verifikation als die *Modellprüfung*. Dafür können *Programm-* bzw. *Modellanalyse*n i.d.R. sehr effizient durchgeführt werden, die *Modellprüfung* hingegen nicht. Gelingt es, eine *Modellanalyse* mit maximaler Präzision durchzuführen, so sind evtl. die Vorzüge beider Seiten vereint und hat damit die gewünschte Präzision von *Modellprüfung* mit der gewünschten Analysegeschwindigkeit der *Programmanalyse* kombiniert.

Definition 3.3.2 (*Exakte* Modellanalyse)
Eine Modellanalyse *heißt* exakt*, falls in den berechneten Analyseergebnissen (abstrakten Informationen) weder eine Über- noch eine Unterapproximation des Modellverhaltens feststellbar ist. Die Analyseergebnisse heißen dann ebenso exakt.*

Eine *Modellanalyse* mit maximaler Präzision ist demnach *exakt*. Es gibt dann zu jedem berechneten Analyseergebnis auch mindestens einen Lauf des Modells, welche die Exaktheit der Analyseergebnisse nachweisen. Im Falle des Beispiels muss es dann für jede Variable v Läufe geben (nicht notwendigerweise verschiedene) sowie *Variablenbelegung*en env und env' innerhalb dieser Läufe, so dass für $\psi(v) = (a, b)$ einmal $[\![v]\!]_{env} = a$ und einmal $[\![v]\!]_{env'} = b$ gilt. Auch wenn es keinen Lauf für ein c mit $a < c < b$ gibt, so dass $[\![v]\!]_{env''} = c$ für jede beliebige *Variablenbelegung* env'' der Läufe, so heißt die *Modellanalyse* dennoch exakt.

Definition 3.3.3 (Von *Programmanalysen* übertragene Eigenschaften)
Analog zu Programmanalysen *[221] heißen* Modellanalyse*n dann* flusssensitiv*, wenn die Analyseergebnisse (abstrakten Informationen) abhängig sind von der Position innerhalb eines Laufs, da eine flusssensitive Analyse die Reihenfolge auftretender* Anweisungen *(für* SPDS *Konfigurationenübergänge) beachtet. Flussinsensitive Analysen unterscheiden dies nicht.*

Berechnet eine Modellanalyse *für einzelne Pfade in Abhängigkeit der Bedingungen an Verzweigungen unterschiedliche Analyseergebnisse, so heißt die* Modellanalyse pfadsensitiv.
Eine Modellanalyse *heißt* intraprozedural *[109], wenn die Analyseergebnisse nur innerhalb von Prozeduren berechnet werden (Analyseergebnisse fließen nicht über Prozedurgrenzen hinweg). Andernfalls heißt eine* Modellanalyse interprozedural.
Kontextsensitiv *ist eine* Modellanalyse*, wenn die Analyseergebnisse von tiefer gelegenen Kellersymbolen und nicht nur vom obersten Kellersymbol abhängen. Die Analyse berücksichtigt dann den Kontext des obersten Kellersymbols. Kontextsensitive* Modellanalysen *sind daher auch interprozedural.*
Eine Modellanalyse *heißt* entscheidbar*, wenn es für die gleiche Art der Analyseergebnisse (im Beispiel die Funktionen ψ) eine exakte* Modellanalyse *gibt.*
Eingesetzte nicht exakte Modellanalysen *sind im Rahmen der Verifikation i.d.R.* konservativ *und bestimmen eine sichere nicht exakte Approximation der abstrakten Informationen.*

Exaktheit von *Programm-* und *Modellanalysen* zur Bestimmung maximal *präziser* Analyseergebnisse ist allerdings in allen untersuchten Publikationen im Zusammenhang mit dieser Arbeit nicht das Ziel. Die in dieser Arbeit experimentell untersuchten *Modellanalysen* können für *SPDS* (und damit auch für *SPDS'*, *PDS* sowie *KS*) oft exakt durchgeführt werden [69, 68]. Im Gegensatz zu Hochsprachen ist dies nicht möglich (Halteproblem). Dies führt auf der anderen Seite auch zu sehr *präzisen* Analysen für die Ausgangssprache, welche nicht auf endliche Modelle beschränkt sind und beliebige Rekursion berücksichtigen, sog. *Rekursionspräzise* Analysen. Erkenntnisse dazu wurden bereits in [69] veröffentlich. Wie später gezeigt wird, ist die exakte Feststellung von *Programmeigenschaften* je nach Art der durch die Analyse zu bestimmenden Eigenschaften teilweise auch eine spezielle Form der *Modellprüfung*.

3.3.2. Erreichbarkeitsproblem

Die einfachste Form der *Modellprüfung* besteht darin, die *Erreichbarkeit* von Zuständen z.B. einer Kripkestruktur zu prüfen. Bei gegebener Kripkestruktur $M = (S, \rightarrow, L)$ ist das *Erreichbarkeitsproblem* die Frage, ob es in M von einem Zustand $s \in S$ einen Pfad zu einem anderen Zustand $z \in S$ gibt ($s \rightsquigarrow z$?). Beim *verallgemeinerten* Erreichbarkeitsproblem können statt einzelnen Zuständen s bzw. z Mengen $A, B \subseteq S$ verwendet werden. Eine derartige Frage nach der *Erreichbarkeit* ist zumindest aus theoretischer Sicht **dann** trivial, wenn die Kripkestruktur **endlich** ist. Dann genügt eine einfache Tiefen- bzw. Breitensuche in der Menge der Zustände der Kripkestruktur. Und in der Tat funktionieren einige endliche explizite *Modellprüfer* im Grunde auf diese Weise, wenn auch wie in Abschnitt 1.2 auf Seite 13 und 2 auf Seite 15 optimiert. Die in dieser Arbeit zu Grunde liegenden Kripkestrukturen sind aber i.d.R. unendlich und sie werden durch *SPDS* beschrieben. Dennoch kann diese Formen der *Erreichbarkeit* auf *SPDS* mit *Konfigurationen* übertragen werden. Im Abschnitt 3.1 ab Seite 21 wurde erläutert, was unter *Erreichbarkeit* von *Konfigurationen* zu verstehen ist und auf das Vorhandensein eines Laufs des zu Grunde liegenden *PDS* $\gamma[\![S]\!] = (P, \Gamma, \hookrightarrow, I, L)$ eines *SPDS* S zurückgeführt. Es wird nun erläutert, wie *Konfigurationenmengen* zum Zwecke der *Modellprüfung* beschrieben werden können. Eine Menge $A = Konf(l) \subseteq S$ von *Konfigurationen* für ein *SPDS* kann durch Angabe einer Marke $l \in Marken$ bestimmt werden. Es ist $Konf(l) := \{w \mid w \in P \times \Gamma^*, l \in Kopf(w)\}$. Damit kann die *Erreichbarkeit* in *SPDS* auf Marken und *Variablenbelegungen* erweitert werden. Es heißt eine Marke $l_2 \in Marken$ von $l_1 \in Marken$ *erreichbar* (in Zeichen: $l_1 \rightsquigarrow l_2$), falls $Konf(l_1) \rightsquigarrow Konf(l_2)$. Analog kann $\rightsquigarrow$ wieder mit Mengen verwendet werden: $I \rightsquigarrow l_2 :\Leftrightarrow I \rightsquigarrow Konf(l_2)$. Eine *Variablenbelegung* $env \in ENV$ heißt *erreichbar* oder *realisierbar*, falls es eine entsprechende erreichbare *Konfiguration* s gibt: $I \rightsquigarrow env :\Leftrightarrow I \rightsquigarrow s \wedge env = env^s$. Die Menge aller *realisierbaren* Variablenbelegungen an einer Marke l sei mit $RENV_l := \{env^s \mid s \in Post^*(S), l \in Marken(Kopf(s))\} \subseteq ENV$ bezeichnet[18].
So beantwortet die *Erreichbarkeitsprüfung* schließlich die existentielle Frage

> Existiert ein Pfad, so dass eine bestimmte Situation erreicht werden kann?

Durch Allquantifizierung ergibt sich:

[18] $RENV_l$ ist berechenbar, da die *Erreichbarkeit* von *Konfigurationen* für *SPDS* bzw. *PDS* berechnet werden kann.

Führen alle Pfade zu einer bestimmten Situation?

Was eine andere und bessere Möglichkeit der *Modellprüfung* von Kellersystemen nahe legt. Und zwar jene, Eigenschaften in Form *temporaler Formel*n zu überprüfen.
So ist es mit *temporalen Formel*n z.B. möglich, auch Aussagen zur Terminierung von *Programmen* zu treffen, ohne dabei sog. Terminierungs-Analysen [12, 11, 165, 230] einsetzen zu müssen, welche versuchen, die Terminierung von *Programmen* nachzuweisen.

3.3.3. Temporale Formeln

Temporale Formeln sind aussagenlogische *Ausdrücke* erweitert mit temporalen Operatoren (vgl. Modallogik) [40, 101, 184]. Temporale Formeln können genutzt werden, um zeitabhängige Eigenschaften über ein Modell zu formulieren. Ihr *Wahrheitsgehalt* bezüglich eines Modells kann mittels *Modellprüfung* bestimmt werden.

Syntax und Semantik von CTL*, CTL, ACTL, LTL Die Logik CTL* besteht aus einer Menge von Zustandsformeln Zf ($AExpr \subseteq Zf$), welche Aussagen über einen Zustand eines Modells trifft. Innerhalb einer CTL* Formel selbst sind auch Pfadformeln (Menge Pf) erlaubt, welche wie folgt definiert sind. Die Quantoren A bzw. E werden verwendet, um Aussagen über *alle Pfade* bzw. *min. einen Pfad* zu treffen. Die Operatoren X, U, F sowie G repräsentieren die Operatoren *Nächster* (Next), *strenges Bis* (strong until), *irgendwann* (future/eventually) bzw. *immer* (always). Dabei sind E,F,G bzw. $\vee$ abkürzende Schreibweisen:

$$\begin{aligned} E\phi &= \neg A\neg\phi \\ F\phi &= true\ U\ \phi \\ G\phi &= \neg F\neg\phi \\ \phi \vee \psi &= \neg(\neg\phi \wedge \neg\psi), \end{aligned}$$

wobei $\phi, \psi \in P$ und $p \in Zf$ sind. Somit stellen $A, X, U, \neg$ und $\wedge$ die Basis-Operatoren dar. Induktiv sind dann Zf und CTL* Formeln wie folgt definiert:

- $AExpr \subset Zf$
- $\forall \phi, \psi \in Zf : \neg\phi \in Zf,\ \phi \wedge \psi \in Zf$
- $Zf \subseteq Pf$
- $\forall \phi, \psi \in Pf : A\phi \in Zf, \neg\phi \in Pf,\ \phi \wedge \psi \in Pf,\ X\phi \in Pf,\ \phi U \psi \in Pf$

CTL ist die Teillogik von CTL*, in der die temporalen Operatoren X, U (bzw. auch die Abkürzungen F und G) stets durch einen vorangestellten Pfadquantor A (bzw. E) kombiniert werden. ACTL* ist eine Teillogik von CTL* in Negationsnormalform[19], welche nur den Pfadquantor A enthalten. Die Logik LTL besteht aus CTL*-Formeln ohne *Verwendung* der Pfadquantoren A bzw. E. Die Logik *LTL-X* besteht aus LTL-Formeln ohne *Verwendung* des Operators X (analog *CTL*-X* und *CTL-X*). Eine LTL- bzw. LTL-X-Formel ϕ wird dann interpretiert als CTL*-Formel $A\phi$. Die Teilmengenbeziehung der Logiken findet sich in Abbildung 3.3. Die Logiken ACTL, ACTL-X und ACTL*-X sind analog definiert.
Gilt eine Formel ϕ an einem Zustand s einer Kripkestruktur $M = (S, \rightarrow, L)$ bzw. an einer *Konfiguration* s eines *SPDS* $M = (R, \Gamma, \hookrightarrow, I)$, so ist $s \models_M \phi$, bzw. $s \models \phi$, wenn M aus dem Kontext klar ist. Sei $p = s_0 \rightarrow s_1 \rightarrow \ldots$ ein unendlicher Lauf, wobei $fst(p) = s_0$ der Anfangszustand bzw. die *Anfangskonfiguration* des Laufs p ist und $p_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$ der Suffix des Laufs beginnend bei s_i. Terminiert der Lauf p an einem Zustand (*PDS*) bzw. einer Konfiguration(*SPDS*) s_n, so wird zur temporalen *Interpretation* der letzte Zustand bzw. die letzte *Konfiguration* immer wieder

[19] Negationen dürfen nur vor $e \in AExpr$ verwendet werden.

$$\begin{array}{ccc} \text{CTL*} & \supset & \text{CTL*-X} \\ \cup & & \cup \\ \text{CTL} & \supset & \text{CTL-X} \\ \cup & & \cup \\ \text{LTL} & \supset & \text{LTL-X} \end{array}$$

Abbildung 3.3.: Teilmengenbeziehungen der definierten temporalen Logiken [40, 101, 184]

wiederholt: d.h. $p_i \equiv s_i \to s_{i+1} \to s_{n-1} \to s_n \to s_n \to s_n \ldots$. Dann sei $\models$ neben Zuständen bzw. *Konfigurationen* s auch für Läufe p wie folgt erklärt (analog zu [40, 101, 184]):

$$\begin{aligned} s \models a &:\Leftrightarrow a \in L(s) \\ s \models \neg\phi &:\Leftrightarrow s \not\models \phi \\ s \models \phi \wedge \psi &:\Leftrightarrow s \models \phi \ \wedge \ s \models \psi \\ s \models A\phi &:\Leftrightarrow (\forall p = s \to s_1 \to \cdots : (p \models \phi)) \\ p \models \phi,\ mit\ \phi \in Z &:\Leftrightarrow fst(p) \models \phi \\ p \models \neg\phi,\ mit\ \phi \in Z &:\Leftrightarrow p \not\models \phi \\ p \models \phi \wedge \psi &:\Leftrightarrow p \models \phi \ \wedge \ p \models \psi \\ p \models \phi U \psi &:\Leftrightarrow \exists i \geq 0 : ((p_i \models \psi) \wedge (\forall j < i : p_j \models \phi)) \\ p \models X\phi &:\Leftrightarrow p_1 \models \phi \end{aligned}$$

Im Folgenden wird auch $S \models \phi$ für eine Menge *Konfigurationen* S verwendet, falls $\forall s \in S : s \models \phi$.

Bemerkung 3.3.1 *Wie für endliche Kripkestrukturen stets möglich, können oft auch für Kellersysteme CTL* Formeln auf LTL Formeln zurück geführt werden [120], so dass LTL i.d.R. genügt, um in der Praxis temporale Eigenschaften für Modelle zu spezifizieren und zu prüfen. Werden statt Mengen von Merkmalen sogar reguläre Merkmale (siehe [120]) bzw. Annotationsausdrücke über den Kellerinhalt erlaubt, so sind CTL*-Formeln auf LTL-Formeln reduzierbar [120]. Intuitiv klar ist daher: ist der durch ein* SPDS *Modell erreichbare* Konfigurationenraum *endlich ($|Post^*(I)| < \infty$), so ist CTL* (auch ohne regulärer Merkmale) auf LTL reduzierbar [120]. Für die in dieser Arbeit betrachteten* SPDS *gilt dies allerdings nicht!*

Beispiel 3.3.1 (Beispiel einer *temporalen Formel* über ein Merkmal „$x > 0$“)
Für ein Beispiel einer temporalen Formel *sei „$x > 0$“ $\in$ AExpr ein Merkmal. x ist darin eine globale Variable eines* SPDS *P, so dass das Merkmal „$x > 0$“ diejenigen* Konfigurationen *z von P besitzen, für welche die Variable x positiv ist. D.h. „$x > 0$“ $\in L(z) \Leftrightarrow [\![x]\!]_{env^z} > 0$. Die CTL*-X Formel EFa besagt dann, dass es beginnend bei z einen Lauf mit einer speziellen* Folgekonfiguration *gibt, bei der $x > 0$ ist. Es ist demnach $z \models EF$ „$x > 0$“ $\Leftrightarrow \exists z_n : z \to z_1 \to z_2 \cdots \to z_n \wedge$ „$x > 0$“ $\in L(z_n)$.*

Definition 3.3.4 (Annotationsfunktion $L_\phi(s)$ für *temporale Formel* ϕ)
Die Annotationsfunktion $L_\phi(s)$ für eine Konfiguration *$s \in P \times \Gamma^*$ beschreibe bei gegebener* temporaler Formel *ϕ sämtliche erfüllte Annotationsausdrücke über Variablen aus ϕ:*

$$L_\phi(s) := \{e \in L(s) \mid \forall v \in Vars(e) : v \in \phi\}, \tag{3.2}$$

welche über SPDS-*Variablen aus ϕ gebildet werden können.*

Ist eine *temporale Formel* ϕ im Rahmen der *Modellprüfung* gegeben, so wird damit die *Interpretation* des *SPDS* S zum *PDS* $\gamma[\![S]\!]_\phi$ konkretisiert, da in den Annotationsausdrücken von L evtl. Variablen, Prozeduren etc. verwendet werden, welche für den *Wahrheitsgehalt* von ϕ unbedeutend sind. Die *Semantik* eines *SPDS* $S = (Vgbl, PrzDesc, init)$ wird damit vom *PDS* $\gamma[\![S]\!] = (\Sigma_{Vgbl}, \Gamma, \hookrightarrow, I, L)$ zum *PDS* $\gamma[\![S]\!]_\phi = (\Sigma_{Vgbl}, \Gamma, \hookrightarrow, I, L_\phi)$ präzisiert. D.h. es wird die Annotationsfunktion L_ϕ statt L verwendet. Werden dann Eigenschaften für $\gamma[\![S]\!]$ gezeigt, so gelten diese wegen $L_\phi \subseteq L$ auch für $\gamma[\![S]\!]_\phi$.

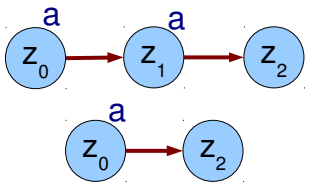

Abbildung 3.4.: Beispiel einer CLT-X, CTL*-X und LTL-X invarianten Transformation (oben original M, unten verkleinertes Modell M')

3.3.4. Modellprüfung temporaler Formeln

Bei der *Modellprüfung* mit *temporalen Formel*n wird zu einem gegebenen Modell M, einer Menge von Anfangssitutationen und einer Menge *temporaler Formel*n T der *Wahrheitsgehalt* der *temporalen Formel*n T für bestimmte Situationen berechnet. M kann dabei entweder ein *SPDS*, *PDS*, eine Kripkestruktur o.Ä. sein. Anfangssituation ist im Folgenden ein Oberbegriff für Anfangszustände von *PDS* bzw. Kripkestrukturen sowie *Anfangskonfigurationen* von *SPDS*. Handelt es sich um ein Modell mit endlichem Zustandsraum, so kann bei der *Modellprüfung* auch der *Wahrheitsgehalt* der *temporalen Formel*n T für *alle* Situationen bestimmt werden.
Die einfachste Form der *Modellprüfung* ist die Bestimmung der *Erreichbarkeit* von Situationen. Wird bei der *Erreichbarkeitsprüfung* nach der Erreichbarkeit einer Marke $l_0 \in Marken$ eines *SPDS* (analog *PDS*) aus einer gegebenen *Anfangskonfiguration* $i_0 \in I$ gefragt, so kann dazu der *Wahrheitsgehalt* der CTL-X-Formel EFl_0 an der *Konfiguration* i_0 berechnet werden. D.h. es ist $i_0 \rightsquigarrow l_0 \Leftrightarrow i_0 \models EFl_0$. Für diese kann dann durch *Modellprüfung* der *Wahrheitsgehalt* bestimmt werden.

3.3.5. Korrektheit von Modelltransformationen

In diesem Abschnitt wird erläutert, was Korrektheit von *Modelltransformationen* bedeutet. Dazu werden verschiedene Invarianz-Begriffe definiert. In der vorliegenden Arbeit wird ein ursprüngliches Modell M in ein neues Modell M' überführt zur effizienteren *Modellprüfung* (Transformation). Es ist wichtig zu wissen, wann, ob und wie sich Aussagen von *temporalen Formel*n bei der Transformation von M nach M' verändern.

Modelltransformationen. *Modelltransformationen* (siehe Definition 3.3.5) verändern gegebene Modelle in neue veränderte Modelle (Funktionsweise vergleichbar zu Source-2-Source-Compilern).

Definition 3.3.5 (*Modelltransformation*)
Eine Funktion T vom Typ $T : \text{SPDS} \to \text{SPDS}$ *oder* $T : \text{SPDS}' \to \text{SPDS}'$ *oder* $T : \text{PDS} \to \text{PDS}$ *oder* $T : \text{KS} \to \text{KS}$ *heißt* Modelltransformation.

Beispiel 3.3.2 (Temporale Wahrheitsveränderung)
Man betrachte das Beispiel aus Abbildung 3.4. Es ist ein sehr einfaches Transitionssystem mit 3 Zuständen z_0, z_1, z_2, den Transitionen $z_0 \to z_1$ sowie $z_1 \to z_2$ und dem Merkmal $a \in L(z_0), a \in L(z_1), a \notin L(z_2)$. Wird die Transitionenfolge $z_0 \to z_1 \to z_2$ vereinfacht zu $z_0 \to z_2$ und $z_0 \not\to z_1$, d.h. der Zustandsraum auf die 2 Zustände z_0 und z_2 verkleinert, so ist zwar weiterhin $z_0 \rightsquigarrow z_2$ ($z_0 \models_M EF\neg a$ sowie $z_0 \models_{M'} EF\neg a$), aber die temporale Formel *Xa gilt nun nicht mehr. D.h. $z_0 \models_M Xa$, aber $z_0 \not\models_{M'} Xa$. Wird der Zustands- bzw.* Konfigurationenraum *eines Modells verändert (z.B. verkleinert, so wie es auch Ziel von Untersuchungen dieser Arbeit ist), so kann sich der* Wahrheitsgehalt temporaler Formel*n ändern.*

Wie am Beispiel deutlich wird, führt die *Verwendung* des X-Operators ggf. zu einer Veränderung des *Wahrheitsgehalts*. Eine Möglichkeit den *Wahrheitsgehalt* wieder anzupassen, ist die Veränderung der *temporalen Formel*. Dadurch wird die temporale Formel i.d.R. umfangreicher, was zu

einer aufwändigeren *Modellprüfung* führt. Dies ist jedoch nicht im Sinne einer effizienteren Modellprüfung, da die *Modellprüfung temporaler Formel*n von Kellersystemen exponentielle Laufzeit in der Größe (bzw. Länge) einer temporalen Formel besitzt [220]. D.h. eine Vergrößerung der *temporalen Formel* führt zu einer rapiden Verschlechterung des Laufzeitverhaltens. Eine weitere Möglichkeit, wie der *Wahrheitsgehalt temporaler Formel*n unverändert bleibt, ist die *Verwendung* von Teillogiken. Wie im Beispiel ist es eben auch oft nicht wichtig, in wie vielen Schritten eine bestimmte (z.B. Fehler-) Situation auftritt. Und unter Berücksichtigung automatischer Modellgenerierung ist es i.d.R. auch unklar, nach wie vielen Transitionen der nächste *Programmpunkt* des ursprünglichen *Programms* erreicht wird. Dies gilt z.B. auch dann, wenn bei der Modellgenerierung *SPDS*-Erweiterungen verwendet werden, welche erst auf SPDS-*Anweisungen* abgebildet werden müssen. Temporale Eigenschaften werden für das ursprüngliche *Programm* und nicht für das Modell formuliert, was neben der Erleichterung bei der Berücksichtigung des *Wahrheitsgehalt*s *temporaler Formel*n in der Praxis der Softwaremodellprüfung dazu führt, auf den X-Operator i.d.R. zu verzichten [66]. D.h. es werden gern die Teillogiken CTL-X, CTL*-X und LTL-X eingesetzt, welche ihren *Wahrheitsgehalt* nicht so einfach ändern, wenn der Zustands- bzw. *Konfigurationenraum* verkleinert wird, wie es auch im Beispiel mit den *temporalen Formel*n Xa und $EF\neg a$ deutlich wird. Ändert sich der *Wahrheitsgehalt* einer *temporalen Formel*n nicht bei einer Veränderung des Modells, so wird im Folgenden der Begriff *invariant* dafür verwendet. Unter Korrektheit von *Modelltransformationen* wird dann verstanden, dass gezeigt wird, dass eine Transformation *invariant* bezüglich einer temporalen Logik ist. D.h. der *Wahrheitsgehalt* jeglicher Formel, die dann formulierbar ist, wird durch die Transformation nicht verändert. Im einfachsten Fall, d.h. bei der *Erreichbarkeitsprüfung*, besteht der Nachweis darin zu zeigen, dass eine a priori bestimmte Situation (z.B. Marke eines *SPDS*) genau dann im transformierten Modell M' *erreichbar* ist, wenn sie auch im ursprünglichen Modell M erreichbar ist. Für diesen Sachverhalt wird im Folgenden analog wie bei *temporalen Formel*n der Begriff *Erreichbarkeits-invariant* verwendet.

Definition 3.3.6 (*Erreichbarkeits-Invarianz* von Marken)
Sei ein SPDS S*, eine Marke* $l \in Marken(S)$ *und eine Transformation* $T_l : \text{SPDS} \to \text{SPDS}$ *gegeben, welche in Abhängigkeit von* l *ein Modell verändern kann.* S *beschreibe das* PDS $\gamma[\![S]\!] = (P, \Gamma, \hookrightarrow, I, L)$*. Die Transformation*[20] T_l *heißt* Erreichbarkeits-invariant *bezüglich der Marke* l*, gdw.* $(l \in Post^*(S) \Leftrightarrow l \in Post^*(T_l(S)))$.

Um einen Zusammenhang zu *temporalen Formel*n zu verdeutlichen, kann beispielsweise mittels der Annotationsfunktion $L : P \times \Gamma^* \to 2^A$, wobei

$$L(s) := \begin{cases} \{l\} & falls\ l \in Kopf(s) \\ \{\} & sonst, \end{cases}$$

für *Konfigurationen* $s \in P \times \Gamma^*$ des zu Grunde liegenden *PDS* $\gamma[\![S]\!]$, die *Erreichbarkeits-Invarianz* von Marken auch ausdrückt werden als $(I \models_S \neg Fl \Leftrightarrow I' \models_{T_l(S)} \neg Fl)$, wobei $I' := I(T_l(S))$ die Initialkonfigurationen für das zu Grunde liegende *PDS* des transformierten *SPDS* $T_l(S)$ sind. Für die Prüfung auf *Erreichbarkeit* von Situationen sind demnach zuvor gegebene Merkmale (Funktion L) unwichtig, da die Merkmale nicht in *temporalen Formel*n verwendet werden.

Bemerkung 3.3.2 *Durch logische* Ausdrücke *wie* $x > 5$ *und* $y < 7$*, können (analog zu Definition 3.3.6 mit Marken) statt dessen Mengen von Situationen bzw. Konfigurationen* $E = \{s \in P \times \Gamma^* \mid [\![x]\!]_{env^s} > 5 \wedge [\![y]\!]_{env^s} < 7\}$ *konstruiert werden, welche auf* Erreichbarkeit *zu prüfen sind. Beispiele solcher* Ausdrücke *sind Bedingungen, Zusicherungen, Invarianten oder spezielle* Programmeigenschaften *(vgl. Java Modelling Language JML [203] oder Standard Annotation Language SAL [211]), welche zu überprüfen sind. Insbesondere für* SPDS *sind solche Situationen-Mengen wie* E *üblicherweise unendlich und werden daher indirekt durch derartige* Ausdrücke *definiert an Stelle, wie bei endlichen* Modellprüfer*n gelegentlich üblich, exakt angegeben.*

[20] Für die ausgezeichnete a priori bekannte Marke l ist die *Erreichbarkeit* aus den *Initialkonfigurationen* I zu prüfen (in Zeichen $l \in Post^*(S)$, d.h. $\exists s \in P \times \Gamma^* : l \in s \wedge s \in Post^*(S)$).

Für die Zwecke dieser Arbeit werden nur logische *Ausdrücke* verwendet, welche anhand des Kopfes einer *Konfiguration* bestimmt werden können. D.h. es sind lediglich Aussagen zu globalen sowie lokalen (auch Parameter-) Variablen gestattet und über die aktuelle Marke[21]. Mit solchen logischen *Ausdrücke*n lässt sich die *Erreichbarkeits-Invarianz* wie folgt definieren.

Definition 3.3.7 (*Erreichbarkeits-Invarianz* von booleschen *Ausdrücke*n)
Seien ein SPDS S*, ein boolescher Ausdruck* $b \in Expr$ *(*$[\![b]\!] \in \{0,1\}$*) und eine* Modelltransformation $T_b : \text{SPDS} \to \text{SPDS}$ *gegeben, welche in Abhängigkeit von* b *ein Modell verändern kann.* S *beschreibe das* PDS $\gamma[\![S]\!] = (P, \Gamma, \hookrightarrow, I, L)$*. Die Transformation*[22] T_b *heißt* Erreichbarkeits-invariant *bezüglich des Ausdrucks* b*, gdw.* $(b \in Post^*(S) \Leftrightarrow b \in Post^*(T_b(S)))$.

Die beiden *Erreichbarkeits-Invarianz*en für Marken sowie für *Ausdrücke* zusammen ergeben den allgemeineren Begriff der *Erreichbarkeits-Invarianz*.

Definition 3.3.8 (*Erreichbarkeits-Invarianz*)
Eine Modelltransformation $T_{b,l} : \text{SPDS} \to \text{SPDS}$ *heißt* Erreichbarkeits-invariant*, wenn sie Erreichbarkeits-invariant für einen beliebigen booleschen Ausdruck* $b \in Expr$ *(*$[\![b]\!] \in \{0,1\}$ *und Erreichbarkeits-invariant für eine beliebige Marke* $l \in Marken(S)$ *ist.*

D.h. bei der *Erreichbarkeits-Invarianz* können Marken oder *Ausdrücke* (oder beides) gegeben sein. Allerdings genügt in der weiteren Betrachtung die *Erreichbarkeits-Invarianz* von Marken, da diejenige mit *Ausdrücke*n zurückgeführt werden kann auf die Erreichbarkeits-Invarianz von Marken. Dazu wird dem Modell nach jedem *Konfigurationsübergang* die Erfülltheit des gegebenen Ausdrucks b mit einem bedingten Sprung geprüft (siehe Beweisskizze folgenden Satzes).

Satz 3.3.1 (*Erreichbarkeits-Invarianz*)
Die Erreichbarkeits-Invarianz *von* Ausdrücke*n kann zurückgeführt werden auf die* Erreichbarkeits-Invarianz *von Marken.*

Der Beweis findet sich im Anhang B ab Seite 145. Ist eine Transformation T *Erreichbarkeits-invariant* bezüglich Marken (l beliebig gewählt in Definition 3.3.6), so folgt daraus automatisch auch die *Erreichbarkeits-Invarianz* beliebiger logischer *Ausdrücke*.
Neben der *Erreichbarkeits-Invarianz* kann eine Transformation aber auch invariant bezüglich einer *temporalen Formel* sein. Sei dazu für die folgende Definition $\alpha \in \{\text{LTL, CTL, CTL*, LTL-X, CTL-X, CTL*-X}\}$ eine temporale Logik.

Definition 3.3.9 (α-*Invarianz*)
Seien ein SPDS S*, eine Annotationsfunktion* $L : P \times \Gamma^* \to 2^A$ *und eine Transformation* $T_\phi : \text{SPDS} \to \text{SPDS}$ *gegeben, welche abhängig von einer beliebigen* α*-Formel* ϕ *das Modell transformiert. Weiter sei* $\gamma[\![S]\!]_\phi = (P, \Gamma, \hookrightarrow, I, L_\phi)$ *das* PDS*, welches von* S *beschrieben wird. Dann heißt die Transformation* T α-invariant*, gdw.* $\forall \phi \in \alpha : (S \models \phi \Leftrightarrow T_\phi(S) \models \phi)$.

Die Invarianzbegriffe für Transformationen stehen zueinander in Beziehung. Da z.B. $\neg Fl \in$ LTL-X, ist jede LTL-X-invariante Transformation (siehe Erklärungen zu Definition 3.3.6 und Satz 3.3.1) auch *Erreichbarkeits-invariant*. Eine Übersicht zu den Invarianz-Beziehungen für Transformationen ist in Abbildung 3.5 zu finden, welche sich ansonsten direkt aus den Teilmengenbeziehungen von Abbildung 3.3 auf Seite 36 ergibt [40, 101, 184]. Viele der in dieser Arbeit untersuchten Transformationen sind CTL*-X-invariant und damit auch CTL-X- und LTL-X-invariant. D.h. bei gegebener Formel $\phi \in$ CTL*-X ändert sich der *Wahrheitsgehalt* von ϕ nicht durch die Transformation. Informell erklärt sich dies wie folgt. Eine *temporale Formel* $\phi \in$ CTL*-X kann lediglich an Hand der Merkmale Veränderungen quasi „beobachten“. Wegen des fehlenden X-Operators ist ϕ dann prinzipiell nicht in der Lage, direkt aufeinander folgende Situationen zu unterscheiden. Genaueres wird in den jeweiligen Kapiteln zu den Transformationen erläutert und bewiesen.
Um α-Invarianz einer Transformation nachzuweisen, wird die Existenz einer Äquivalenzrelation gezeigt zwischen transformiertem und nicht transformiertem Modell. Dazu wurden die folgenden Definitionen orientiert an [190, 129, 78] und auf *PDS* übertragen.

[21] Z.B. ehemaliger *Programmpunkt* des ursprünglichen Programms.
[22] Es ist zu prüfen, ob aus den *Initialkonfigurationen* I eine *Konfiguration* $s \in Post^*(S)$ *erreichbar* ist, so dass $[\![b]\!]_{env^s} = 1$ (in Zeichen $b \in Post^*(S)$).

$$\begin{array}{ccc}
\text{CTL*-Invarianz} & \subset & \text{CTL*-X-Invarianz} \\
\cap & & \cap \\
\text{CTL-Invarianz} & \subset & \text{CTL-X-Invarianz} \\
\cap & & \cap \\
\text{LTL-Invarianz} & \subset & \text{LTL-X-Invarianz} \\
 & & \cap \\
 & & \text{Erreichbarkeits-Invarianz}
\end{array}$$

Abbildung 3.5.: Invarianzbeziehungen verschiedener Teillogiken und der *Erreichbarkeit*

Definition 3.3.10 (*Simulation*)
Ein PDS $\mathcal{P} = (P, \Gamma, \hookrightarrow, I, L)$ simuliert *ein* PDS $\mathcal{P}' = (P', \Gamma', \hookrightarrow', I', L')$ *gdw. wenn es eine Relation* $R_{sim} \subseteq (P \times \Gamma^*) \times (P' \times \Gamma'^*)$ *derart gibt, so dass* $\forall (s, s') \in R_{sim}$ *gilt*

- $L(s) = L'(s')$ *sowie*
- $\forall s' \rightarrow' z' : \exists s \rightarrow z : (z, z') \in R_{sim}$.

Ein *PDS* $\mathcal{P} = (P, \Gamma, \varnothing, I, L)$ ohne Transitionen wird z.B. simuliert durch ein beliebiges *PDS* $\mathcal{P}'$ mit gleichen *Initialkonfigurationen* $I = I'$ und gleichen Merkmalen für diese *Initialkonfigurationen* $(\forall s \in I : L'(s) = L(s))$. Um triviale Simulationen wie diese zu vermeiden, wird der Begriff der Bisimulation verwendet.

Definition 3.3.11 (*Bisimulation*)
$\mathcal{P}$ *und* $\mathcal{P}'$ *heißen* bisimular, *gdw.* $\mathcal{P}$ *simuliert* $\mathcal{P}'$ *und* $\mathcal{P}'$ *simuliert* $\mathcal{P}$.

Lemma 3.3.1 *Transformationen* T, *welche ein* PDS $\mathcal{P}'$ *in ein bisimulares* PDS $\mathcal{P}$ *überführen, sind CTL*-invariant und damit auch* α*-invariant für ein* $\alpha \in \{$*ACTL, ACTL-X, ACTL*, ACTL*-X, CTL, CTL-X, CTL*, CTL*-X, LTL, LTL-X*$\}$.

Beweis Nachweis siehe [251]. Informell gilt: wird ein *PDS* P' durch ein *PDS* P simuliert, so gibt es für jeden Lauf $a'_1 \rightarrow a'_2 \ldots$ von $\mathcal{P}'$ auch einen Lauf $a_1 \rightarrow a_2 \ldots$ von $\mathcal{P}$, so dass $(a_i, a'_i) \in R$ und $L_\phi(a_i) = L'_\phi(a'_i)$. Eine temporale CTL*-Formel ϕ kann keinen Unterschied zwischen dem *PDS* $\mathcal{P}$ und $\mathcal{P}'$ feststellen, da sich die Folge von Merkmalen nicht unterscheidet. α-Invarianz folgt dann aus den Teilmengenbeziehungen der Logiken.

□

Die natürliche Korrespondenz zum Überspringen von *Zwischenkonfigurationen* wie im Beispiel 3.3.2 liefert die Stotter-Simulation [73, 15].

Definition 3.3.12 (*Stotter-Simulation*)
Ein PDS $\mathcal{P} = (P, \Gamma, \hookrightarrow, I, L)$ stotter-simuliert *ein* PDS $\mathcal{P}' = (P', \Gamma', \hookrightarrow', I', L')$ *gdw. wenn es eine Relation* $R_{ssim} \subseteq (P \times \Gamma^*) \times (P' \times \Gamma'^*)$ *derart gibt, so dass* $\forall (s, s') \in R_{ssim}$ *gilt*

- $L(s) = L'(s')$ *sowie*
- *Für alle Pfade* $\pi' = c'_1 \hookrightarrow' c'_2 \hookrightarrow' \ldots$ *von* $\mathcal{P}'$ *existieren ein Pfad* $\pi = c_1 \hookrightarrow c_2 \hookrightarrow \ldots$ *von* $\mathcal{P}$ *sowie Teilpfade* π'_i *und* π_i *(* $\pi' = \pi'_1 \hookrightarrow' \pi'_2 \hookrightarrow' \ldots$ *und* $\pi = \pi_1 \hookrightarrow \pi_2 \hookrightarrow \ldots$*), so dass gilt:* $\forall k \geq 0, c \in \pi_k, c' \in \pi'_k : (c, c') \in R_{ssim}$.

Auch hier sollen triviale Simulationen vermieden werden, was zur Stotter-Bisimulation führt.

Definition 3.3.13 (*Stotter-Bisimulation*)
$\mathcal{P}$ *und* $\mathcal{P}'$ *heißen* stotter-bisimular, *gdw.* $\mathcal{P}$ *stotter-simuliert* $\mathcal{P}'$ *und* $\mathcal{P}'$ *stotter-simuliert* $\mathcal{P}$.

Lemma 3.3.2 *Transformationen* T, *welche ein* PDS $\mathcal{P}'$ *in ein stotter-bisimulares* PDS $\mathcal{P}$ *überführen, sind CTL*-X-invariant und damit auch* α*-invariant für ein* $\alpha \in \{$*ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X*$\}$.

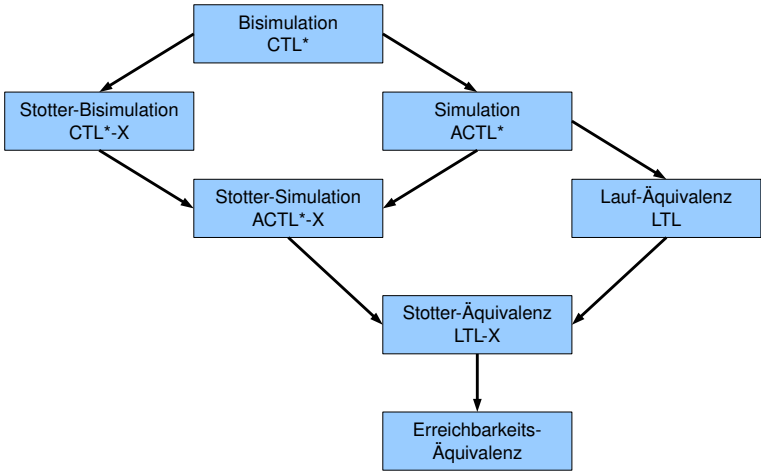

Abbildung 3.6.: Relationen zwischen Simulationsbegriffen und temporalen Logiken [40, 101, 184]

Beweis Analog wie Lemma 3.3.1. Nachweis siehe [251].

Die beschriebenen Simulationsbegriffe sind, wie der Zusammenhang zur Invarianz von *temporalen Formeln* zeigt, vergleichsweise starke Äquivalenzbeziehungen unter den *PDS*. *Modelltransformationen* ver-ändern ein Modell u.U. so stark, dass nur noch eine schwächere Form von Äquivalenz nachweisbar ist. Solche schwächere Formen der Äquivalenz können sein:

Definition 3.3.14 (Äquivalenzbegriffe [40, 101, 184])
Gegeben seien zwei PDS $\mathcal{P} = (P, \Gamma, \hookrightarrow, I, L)$ *und* $\mathcal{P}' = (P', \Gamma', \hookrightarrow', I', L')$ *mit den Merkmalen* $A = A'$. *Dann heißen* $\mathcal{P}$ *und* $\mathcal{P}'$ *...*

- *...* erreichbarkeitsäquivalent*, falls* $\forall a \in A : I \rightsquigarrow a \Leftrightarrow I' \rightsquigarrow a$, *wobei* $I \rightsquigarrow a$ *bedeutet, dass es einen Zustand* $s \in Post^*(\mathcal{P})$ *gibt, so dass* $a \in L(s)$.
- *...* laufäquivalent*, falls es zu jedem Lauf* $\pi = a_1 \rightarrow a_2 \ldots$ *von* $\mathcal{P}$ *auch einen Lauf* $\pi' = a'_1 \rightarrow a'_2 \ldots$ *von* $\mathcal{P}'$ *gibt, so dass* $L(a_i) = L'(a'_i)$ *und umgekehrt.*
- *...* stotteräquivalent *falls es zu jedem Lauf* $\pi = a_1 \rightarrow a_2 \ldots$ *von* $\mathcal{P}$ *auch einen Lauf* $\pi' = a'_1 \rightarrow a'_2 \ldots$ *von* $\mathcal{P}'$ *gibt sowie zwei Pfad-Zerlegungen* $1 = i_1 < i_2 < \ldots$ *und* $1 = j_1 < j_2 < \ldots$, *so dass* $\forall k : \forall i \in [i_k, i_{k+1}[, \forall j \in [j_k, j_{k+1}[: L(a_i) = L'(a'_j)$ *und umgekehrt.*

Die Begriffe Simulation, Bisimulation, Stotter-Simulation und Stotter-Bisimulation, Erreichbarkreits-Äquivalenz, Verklemmungs-Äquivalenz, Lauf-Äquivalenz und Stotter-Äquivalenz übertragen sich in natürlicher Weise auf *SPDS*, da durch *SPDS* nur *PDS* beschrieben sind. Eine zusammenfassende Übersicht der beschriebenen Teillogiken und deren Korrespondenz zu temporalen Logiken findet sich in Abbildung 3.6 (Details finden sich in [190]).
Um jedoch überhaupt Modelle transformieren zu können, müssen zunächst Modelle konstruiert werden (z.B. gegeben als *Quellcode*, siehe Beispiel 3.2.1 auf Seite 24).

Zusammenfassung Es wurden in diesem Abschnitt Prozesse im Zusammenhang mit *Modellprüfung* erklärt. Dazu wurde erläutert, was *SPDS*-Modelle sind, wie deren Verhalten überprüfen, wie diese Modelle prinzipiell aus höheren *Programmiersprachen* gewonnen werden können und was dabei beachtet werden muss. In der folgenden Arbeit wird gezeigt, wie der Vorgestellte Ansatz *Modellverkleinerung* bzw. -Verbesserung genutzt werden kann, um die *Modellprüfung* zu beschleunigen. Hierzu werden *Modelltransformationen* eingesetzt, welche das Modell verändern. Dem anschließend folgt ein Kapitel mit Experimenten, um den Erfolg der Methoden in der Praxis zu belegen.

4. Modellreduktionstechniken

In diesem Kapitel werden Verfahren entwickelt, welche die *Modellprüfung* durch eine Verkleinerung des *Konfigurationenraums* verbessern und gleichzeitig temporal invariant sind, so dass neue Fehlalarme vermieden werden. Es wird quantitativ nachgewiesen, **dass** und **welche** *Modellanalysen* dabei genutzt werden können, um *Modellprüfung* zu verbessern. Modellreduktionstechniken werden dazu in Form von *Modelltransformationen* beschrieben, um so zu zeigen, **wie** diese Verbesserungen erreicht werden können. Verdeutlicht wird auch, wie stark diese *Modelltransformationen* dann den *Konfigurationenraum* verkleinern. Im anschließenden Kapitel 5 ab Seite 123 werden die Erkenntnisse dann experimentell untersucht.

4.1. Beschreibung und Eigenschaften von Modellreduktionstechniken

Modelltransformationen (siehe Definition 3.3.5 auf Seite 37) verändern gegebene Modelle und liefern neue Modelle, mit denen die *Modellprüfung* fortzusetzen ist. Das Ziel von *Modelltransformationen* in dieser Arbeit ist die Verbesserung des Modells zur Begünstigung der *Modellprüfung*. Dabei wird eine *Modelltransformation* T beschrieben durch eine Regel der Form:

$$\frac{b_1\ b_2\ \ldots\ b_n}{m \Rrightarrow m'}$$

Wobei die b_i eine Menge von Bedingungen und m bzw. m' zwei Modellfragmente[1] sind. In den Bedingungen werden üblicherweise Analyseergebnisse (abstrakte Informationen, siehe Definition 3.3.1 auf Seite 32) der *Modellanalysen* verwendet. Die Regel gibt an, dass das Fragment m durch m' zu ersetzen ist, falls sämtliche Bedingungen $b_i = 1$ erfüllt sind.
Dabei können der *Wahrheitsgehalt temporaler Formeln* oder die *Erreichbarkeit* im Modell verändert werden, so dass es zu neuen Fehlalarmen kommt (z.B. durch zusätzliche neue Läufe, welche durch die Transformation im Modell entstehen). False Positives dürfen dann nicht auftreten, da es sich dann nicht mehr um einem korrekten *Modellprüfungs*prozess handelt. D.h. im transformiertem Modell $T(M)$ dürfen keine essenziellen Läufe bezüglich der zu prüfenden Eigenschaften fehlen, welche die *Erreichbarkeit* einer gegebenen Marke l oder die Aussage einer gegebenen *temporalen Formel* ϕ verändern. Derartige Tranformationen T werden als *konservativ* bezüglich ϕ bezeichnet. *Konservative* Tranformationen für alle *temporale Formeln* sind stets mindestens *erreichbarkeits*-invariant nach Satz 3.3.1 auf Seite 39. Nicht-*konservative* Transformationen sollten bei der *Modellprüfung* nicht zum Einsatz kommen, wenn keine neuen Fehlalarme toleriert werden sollen (wie in dieser Arbeit). Sämtliche in dieser Arbeit untersuchten *Modelltransformationen* sind daher stets *konservativ* und damit zumindest *erreichbarkeits*-invariant.
Anhand von Abbildung 3.5 auf Seite 40 wird klar, dass nicht *erreichbarkeits*-invariante Transformationen ebenso auch nicht α-invariant sind für ein beliebiges $\alpha \in$ {LTL, CTL, CTL*, LTL-X, CTL-X, CTL*-X}.

Definition 4.1.1 (*Modellreduktionstechnik*)
Eine Modellreduktionstechnik R *(im Folgenden auch kurz als Modellreduktion bezeichnet) besteht aus einer Menge von* Modelltransformationen *(Transformationsregeln)* $R = \{T_1, T_2, \ldots T_n\}$. *Sämtliche Begriffe für* Modelltransformationen *übertragen sich wie folgt auf Modellreduktionstechniken.*

- R *heißt* erreichbarkeits-*invariant, falls* $\forall i : T_i$ *ist erreichbarkeits-invariant.*

[1] Dies sind Teile der Modellbeschreibung, wie Prozeduren, Prozedurrümpfe, *Ausdrücke*, Variablen oder Marken etc.

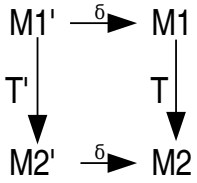

Abbildung 4.1.: Kombinationsmöglichkeiten von *Modelltransformation* und -*Interpretation* für *SPDS*' $M1'$ nach $M2$.

- *R heißt* α*-invariant* ($\alpha \in$ {*LTL, CTL, CTL*, LTL-X, CTL-X, CTL*-X*}), *falls* $\forall i : T_i$ *ist* α*-invariant.*
- *R heißt* konservativ, *falls* $\exists \alpha \in$ {*LTL, CTL, CTL*, LTL-X, CTL-X, CTL*-X*}, *so dass R* α*-invariant ist.*

Auch andere Eigenschaften, wie z.B. die Klassifikation und das Ziel von *Modelltransformationen* - die Verbesserung des Modells zur Begünstigung der *Modellprüfung* - übertragen sich auf Modellreduktionstechniken.

4.2. Modelltransformationen für SPDS-Erweiterungen

Wie Abbildung 4.1 zeigt, gibt es zwei verschiedene Möglichkeiten, eine *Modelltransformation* mit der *Interpretation* von SPDS*'-Anweisungen* zu kombinieren. Einerseits ist es möglich, *Modelltransformationen* T' für *SPDS*' zu konstruieren und anschließend die SPDS*'-Anweisungen* zu interpretieren: $M2 = \delta[\![T'(M1')]\!]$. Andererseits kann aber auch zuerst das *SPDS*' $M1'$ interpretiert und anschließend transformiert werden: $M2 = T(\delta[\![M1']\!])$. Eine Transformation T' definiert über SPDS*'-Anweisungen* kann dabei direkt auch für *SPDS* angewendet werden, da die SPDS-*Anweisungen* lediglich eine Teilmenge (Teilsprache) darstellen. Umgekehrt ergibt sich aber auch für Transformationen T, welche nur über SPDS-*Anweisungen* definiert sind, analoge Transformationen T' für SPDS*'-Anweisungen.*

Lemma 4.2.1 (δ-Homomorphismus) *(Abbildung 4.1)*
Zu jedem $M1' \in$ SPDS$'$ *und jeder* SPDS-*Transformation* T*, angewendet auf* $\delta[\![M1']\!]$*, gibt es eine entsprechende* SPDS*'-Transformation* T'*, so dass die* Interpretation $\delta[\![\cdot]\!]$ *zu einem Homomorphismus wird:* $\delta[\![T'(M1')]\!] = T(\delta[\![M1']\!])$.

Beweis(Skizze) Man interpretiere einen Teil der Modellbeschreibung notfalls bereits derart, dass dies stets gelingt, da auch bereits teilweise interpretierte *SPDS*' noch *SPDS*' sind.

□

Bemerkung 4.2.1 *Dieses Interpretieren ist z.B. dann nötig, wenn bei einer Transformation nur ein einzelner* Reihungseintrag *im Typ verkleinert werden soll. In einer* Reihung *haben aber alle Reihungseinträge den gleichen Typ, so dass dieses spezielle Reihungselement von der* Reihung *los gelöst sein muss, um einen anderen Typ haben zu können. Dies kann erreicht werden, indem die* Reihung *ganz oder teilweise interpretiert wird.*

Zur *Interpretation* von *SPDS*-Erweiterungen bzw. SPDS*'-Anweisungen* sind i.d.R. zusätzliche Variablen und/oder Prozeduren nötig, welche den *Konfigurationenraum* vergrößern. Durch direkte *Interpretation* der SPDS*'-Anweisungen* mittels *Modellanalyse*n und Transformationen T' (ohne Interpretation der SPDS*'-Anweisungen* durch SPDS-*Anweisungen*) kann dies vermieden werden. Für Transformationen direkt auf abkürzende Schreibweisen sind die Effekte zur Zustandraumverkleinerung noch besser, da auch weniger „*Interpretation*scode“ nötig ist, d.h. z.B. weniger zusätzliche

Fallunterscheidungen, aber auch weniger zusätzliche Hilfsvariablen, welche lediglich zur *Interpretation* dienen. Aus diesen Gründen werden geeignete SPDS'-*Anweisungen* in den *Modellanalyse*n und Tranformationen besonders betrachtet und ggf. speziell behandelt, so dass die Analyseergebnisse möglichst *präzise* sind, sich der *Konfigurationenraum* für die Modelle nicht sehr ändert und damit die Effekte der Transformationen in dieser Hinsicht weiter verbessert.
In der weiteren Arbeit werden daher nur Tranformationen T für SPDS-*Anweisungen* formuliert. Für einen Korrektheitsnachweis der Transformationen (Invarianz gegenüber *temporalen Formel*n) sind dann lediglich *SPDS* zu betrachten.
Sämtliche Überlegungen aus diesem Abschnitt übertragen sich in natürlicher Weise auch auf Modellreduktionen.

4.3. Untersuchte Modellreduktionen

In den folgenden Abschnitten werden *Modelltransformationen* konstruiert und untersucht. Dabei verlaufen die jeweiligen Untersuchungen nach folgendem Muster ab. Nach einer einleitenden Motivation zur Schaffung des Problembewusstseins an einem Beispiel und der Beschreibung der Idee werden daran anschließend die zur Erreichung der Idee notwendigen abstrakten Informationen beschrieben. Danach werden mögliche *Programmanalysen* auf ihren Einsatz hin zu diesem Zweck untersucht und ein Beispiel einer entsprechenden *Modellanalyse* angegeben. Das *Modellanalyse*beispiel hat das Ziel, die besagten abstrakten Informationen zu bestimmen. Danach wird die Modellreduktion durch ihre *Modelltransformationen* formal beschrieben und deren Korrektheit nachgewiesen. Abschließend werden verschiedene Eigenschaften der Modellreduktion bezüglich *Konfigurationenraum*veränderung, Einfluss auf *temporale Formel*n etc. erörtert.

4.3.1. Ausdrucksvereinfachung

In diesem Abschnitt wird eine Technik vorgestellt, um *Ausdrücke* $e \in Expr$ in *SPDS*-Beschreibungen zu vereinfachen, um auf diese Weise die Modellbeschreibung zu verbessern. Im Idealfall gelingt die Vereinfachung zu Konstanten.

4.3.1.1. Motivation

Vereinfachte *Ausdrücke* wirken sich in natürlicher Weise positiv auf die *Modellprüfung* aus, da einem Modellprüfer somit das Auswerten komplexer Ausdrücke mit Redundanzen erspart wird. Insbesondere ist dies dann wichtig, wenn diese *Ausdrücke* häufig verwendet werden und oft durch einen *Modellprüfer* mittels z.B. aufwändiger OBDD-Operationen ausgewertet werden. Auch sind *Modellanalyse*ergebnisse im Allgemeinen potentiell um so *präzise*r, je einfacher auftretende *Ausdrücke* sind. Im Idealfall lassen sich *Ausdrücke* auf Konstanten reduzieren. Entscheiden solche Ausdrücke dann über den weiteren Verlauf eines Laufs, so wird hieran die Bedeutung vereinfachter *Ausdrücke* deutlich. Dann nämlich ist es für einen Modellprüfer sehr einfacher möglich, verschiedene Modellpfade auszuschließen. Damit verkleinert sich dann der *Konfigurationenraum* des *SPDS*-Modells, was den *Modellprüfer* beschleunigt.

Problembewusstsein Auszuwertende *Ausdrücke* $e \in Expr$ haben Einfluss auf die Performance des *Modellprüfer*s. Dabei wäre bei einer Vereinfachung zu Konstanten keinerlei *Auswertung* mittels *Variablenbelegung*en für einen solchen Ausdruck nötig. Es ist daher klar, dass einfachere *Ausdrücke* die *Modellprüfung* begünstigen, indem weniger *Ausdrucksauswertungen* (z.B. weniger OBDD-Operationen) statt finden brauchen. Durch Automatismen kommt es i.d.R. vor, dass Artefakte wie $a + 2 > a$ oder $a * a + 5 > 0$ im Modell entstehen[2]. Nicht nur in booleschen *Ausdrücke*n können Artefakte entstehen, sondern auch in arithmetischen Ausdrücken (z.B. $z - z$ im *SPDS*-Fragment vom Beispielquellcode 4.1). Solche unnötig komplizierten *Ausdrücke* entstehen neben

[2] Z.B. sind im *Quellcode* A.5 ab Seite 141 durch die *Interpretation* von *SPDS*'-Anweisungen mittels *SPDS*-Anweisungen derartige Artefakte entstanden.

Quellcode 4.1: Beispiel *SPDS*-Fragment für die Ausdrucksvereinfachung.

```
    int x(2), y(2), z(2);
    ...
    x = (10+2)/4;
    y = x - 1;
l:  z = choose(0,3);
    if (x+z < y) goto l;
    y = z - z;
```

der *Interpretation* von *SPDS*'-Anweisungen nicht nur durch die in dieser Arbeit beschriebenen Reduktionstechniken, sondern vor allem auch bei der automatischen Generierung von *SPDS*-Modellen aus Quellcode. Die Vereinfachung solcher Ausdrücken soll folgendes Beispiel veranschaulichen.

Beispiel 4.3.1 *Im* SPDS-*Fragment aus* Quellcode *4.1 haben die Variablen* x, y *und* z *jeweils den Typ* $bits(x) = bits(y) = bits(z) = 2$*. Durch Analyse der Modellbeschreibung wird klar, dass* x *den konstanten Wert* 3 *annimmt und* y *zunächst den konstanten Wert* 2*. Dann wird klar, dass der Ausdruck* $x + z < y$ *niemals erfüllt sein kann. D.h. es ist stets* $[\![x + z < y]\!]_{env} = 0$ *für jede* realisierbare *Variablenbelegung* $env \in RENV_l$ *an der bedingten Verzweigung (if). So entscheidet diese Erkenntnis bereits über die Terminierung des* SPDS-*Fragments, da die Bedingung der bedingten Verzweigung nie erfüllt ist. Ebenso wie diese Erkenntnis kann auch nicht der Ausdruck* $[\![z - z]\!] = 0$ *als konstant durch eine gewöhnliche so genannte Konstanten-Propagation bzw. -Faltung [169, 195] erkannt werden.*

Lösungsidee *Ausdrücke* können mit Hilfe von Konstanten-Propagation und -Faltung dennoch zunächst vereinfacht werden. Diese Techniken sind jedoch nicht immer sehr *präzise*, wie bereits das einfache Beispiel zeigt. Es werden nun daher darüber hinaus mächtigere Methoden betrachtet, um *Ausdrücke* zu vereinfachen. Es werden dabei arithmetische *Ausdrücke* $e \in Expre$ algebraisch interpretiert und transformiert[3]. Neben Konstanten-Propagation bzw. -Faltung werden für *Ausdrücke* e durch vereinfachende Äquivalenztransformationen (wie z.B. $z - z \Rrightarrow 0$) bessere *Ausdrücke* e' gesucht. Besser ist ein Ausdruck dabei, wenn zur *Auswertung* $[\![\cdot]\!]$ eines Ausdrucks weniger Operationen und Variablenauswertungen benötigt werden. Ein Ausdruck $e \in Expr$ wird dann durch einen besseren bzw. kleineren Ausdruck $e' \in Expr$ ersetzt (*Modelltransformation*). *Ausdrücke* (insbesondere boolesche) werden zusätzlich als Erfüllbarkeitsproblem formuliert sowie gelöst und im Falle von Teilausdrücken vereinfacht.

4.3.1.2. Notwendige abstrakte Informationen

Um *Ausdrücke* mit anderen Ausdrücken ersetzen zu können, werden für auftretende Ausdrücke $e \in Expr$ an Marken $l \in Marken(S)$ des *SPDS* S abstrakte Informationen $\tau_l : Expr \to Expr$ in Form von *alternativen Ausdrücken* $\tau_l(e) \in Expr$ bestimmt. Die *alternativen Ausdrücke* haben zwei wichtige Eigenschaften zu erfüllen. Einerseits sollen sie möglichst klein und andererseits äquivalent bezüglich realisierbarer *Variablenbelegungen* sein. Die Eigenschaft besser bzw. kleiner kann wie folgt erklärt werden.

Definition 4.3.1 (*Größe von* Ausdrücken)
Ein Ausdruck $e \in Expr$ *hat die Größe* $|e| := (n, k) \in \mathbb{N}^2_{\geq 0}$*, wobei* n *die Anzahl verwendeter Operatoren ohne Zurückführung auf Basisoperatoren ist (siehe Abschnitt 3.2.1 ab Seite 24). Mehrfach verwendete Operatoren werden dabei mehrfach gezählt.* k *ist die Anzahl unterschiedlicher Variablen* $v \in Vars$ *im Ausdruck* e *(*$v \in e$*)*[4]*. Ein Ausdruck* $e_1 \in Expr$ *mit Größe* (n_1, k_1) *heißt* besser *oder* kleiner *als ein Ausdruck* $e_2 \in Expr$ *mit Größe* (n_2, k_2) *gdw.* $(n_1 < n_2) \vee (n_1 = n_2 \wedge k_1 < k_2)$*. In Zeichen:* $|e_1| < |e_2|$*.*

[3] D.h. für die arithmetischen Operationen werden die Gesetze der Algebra ausgenutzt.
[4] es ist $|x + x + x| = (2, 1)$

Die Eigenschaft „äquivalent bezüglich realisierbarer *Variablenbelegung*en“ führt zum Begriff der *Belegungsäquivalenz*. Es ist klar, dass äquivalente *Ausdrücke* z.B. $z - z$ und 0 miteinander ersetzt werden können, da sie für beliebige *Variablenbelegung*en gelten. Es können auch *Ausdrücke* miteinander ersetzten, welche bereits für wenige bestimmte *Variablenbelegung*en äquivalent sind. Das Beispiel aus *Quellcode* 4.1 zeigt dies für die *Ausdrücke* „$x + z < y$“ und „0“ (false) für alle realisierbaren Belegungen ($x = 3,\ y = 2,\ 0 \leq z \leq 3$). Der Äquivalenzbegriff kann daher wie folgt erweitert werden.

Definition 4.3.2 (*Belegungsäquivalenz*)
Zwei Ausdrücke $e_1, e_2 \in Expr$ *heißen* ***belegungsäquivalent*** *bezüglich einer Menge von* Variablenbelegung*en* $B \subseteq ENV$*, falls gilt* $\forall b \in B : [\![e_1]\!]_b = [\![e_2]\!]_b$*. In Zeichen:* $e_1 \doteq_B e_2$*. Die Menge aller zugehörigen belegungsäquivalenten* Ausdrücke e' *eines Ausdrucks* e *bei gegebener Menge von* Variablenbelegung*en* $B \subseteq ENV$ *sei mit* $[e]_B := \{e' \in Expr \mid e \doteq_B e'\}$ *bezeichnet.*

Für die abstrakte Informationsfunktion τ_l für *alternative Ausdrücke* muss daher gelten, dass für die Menge an **realisierbaren** *Variablenbelegung*en $RENV_l \subseteq ENV$ an der Marke l gilt: $e \doteq_{RENV_l} \tau_l(e)$. Dann ist für alle $s \in Post^*(S)$ mit $l \in Marken(Kopf(s))$:

$$[\![e]\!]_{env^s} = [\![\tau_l(e)]\!]_{env^s}. \tag{4.1}$$

Eigenschaften Wegen Division mit 0 z.B., sind die *Ausdrücke* $(x-1)^2/(x-1)$ und $(x-1)$ nur dann belegungsäqivalent, wenn $x \neq 1$ für alle realisierbaren *Variablenbelegung*en ist. Es kann nichtrealisierbare *Variablenbelegung*en $env' \in ENV$ geben (keinen Lauf für $S \rightsquigarrow l$ des *SPDS* S, um die *Variablenbelegung*en env' zu erzeugen), so dass $[\![e]\!]_{env'} \neq [\![\tau_l(e)]\!]_{env'}$. Dies ist bei gewöhnlichen Äquivalenztransformationen von arithmetischen *Ausdrücke*n nicht möglich, da diese allgemeingültig für beliebige *Variablenbelegung*en gelten. Auf diese Weise werden mehr Ausdrucksvereinfachungen möglich, was zu kleineren *Ausdrücke*n führt. Eine exakte *Modellanalyse* für *alternative* Ausdrücke bestimmt dann minimale belegungsäquivalente *Ausdrücke*.

Definition 4.3.3 (*Optimale Ausdrücke* $T_l(e)$)

$$T_l(e) := \{e^* \in [e]_{RENV_l} \mid |e^*| = \min_{e' \in [e]_{RENV_l}} |e'|\}$$

ist die Menge optimal*er* Ausdrücke *zu* e *an der Marke* l *(minimale belegungsäquivalente Ausdrücke).*

Leider gibt es i.d.R. potentiell unendlich viele kleinere *Ausdrücke* wegen unendlich vielen möglichen einsetzbaren Konstanten in Ausdrücken. Z.B. sind die unendlich vielen Ausdrücke $1+1, 1+2, 1+3, 1+4$, etc. sämtlich kleiner als der Ausdruck $(1+2)+3$, da die Konstanten in Ausdrücken nicht beschränkt sind im Gegensatz zu den Variablenbelegungen ENV. Hinzu kommt, dass diese Ausdrücke auch andere Variablen beinhalten können als in e vorkommen. Dennoch können *optimale Ausdrücke* in *SPDS* bestimmt werden:

Satz 4.3.1 (Berechenbarkeit *optimale*r belegungsäquivalenter *Ausdrücke*)
Für jede Marke $l \in Marken(S)$ *und jeden Ausdruck* $e \in Expr$ *ist* $T_l(e)$ *endlich und aufzählbar.*

Beweis (Skizze)
Genutzt wird eine Form von Dovetailing, um **alle** *Ausdrücke* e' der Größe nach aufzuzählen, welche kleiner oder gleich groß sind wie e. Auftretende Konstanten werden dann sukzessive vergrößert. Jeder dieser aufgezählten *Ausdrücke* e' ist dann auf Belegungsäquivalenz mit e zu prüfen. Dafür werden die **realisierbaren** *Variablenbelegung*en $RENV_l$ verwendet. Gilt dann für ein erstes $e' \doteq_{RENV_l} e$, so ist die minimal notwendige Größe $|e'|$ gefunden. Sobald für ein erstes e' gilt: $|e'| > |e|$, wird die Aufzählung beendet, da dann nur noch größere Ausdrücke aufgezählt werden. So wurden insgesamt endlich viele belegungsäquivalente optimale Ausdrücke aufgezählt.

□

Satz 4.3.2 (Entscheidbarkeit/Komplexität optimaler belegungsäquivalenter Ausdrücke)
τ_l ist für einen Ausdruck $e \in Expr$ entscheidbar, d.h. es existiert eine exakte Modellanalyse, *welche* optimale Ausdrücke *bestimmen kann. Die Berechnung bereits eines* optimalen *Ausdrucks $\tau_l(e) \in T_l(e)$ zu gegebener Marke l und Ausdruck e eines* SPDS *S ist NP-hart.*

Beweis (Skizze)
Die Entscheidbarkeit ist eine Folgerung aus Satz 4.3.1. Die NP-Härte wird mittels folgender Überlegung deutlich. Betrachtet wird folgende Teilmenge der arithmetischen *Ausdrücke*: Boolesche Variablen $v \in Vars$ ($bits(v) = 1$) und die Operatoren ! (Negation), && (logisches Und) sowie || (logisches Oder) bilden die boolesche Algebra. Nicht realisierbare *Variablenbelegung*en entsprechen dann so genannten Don't Cares bei der mehrstufigen Logiksynthese [181]. Dort wird versucht, minimale Schaltkreise (*alternative Ausdrücke*) für boolesche Funktionen (zu verkleinernde Ausdrücke) zu finden. Die NP-Härte dieses Problems [53] überträgt sich auf die Komplexität von $\tau_l(e) \in T_l(e)$, da boolesche *Ausdrücke* eine Teilmenge der arithmetischen Ausdrücke *Expr* sind. (siehe auch [166, 174, 9, 82]).

□

Bemerkung 4.3.1 *Für reelwerte* Ausdrücke $[\![e]\!] \in \mathbb{R}$ *kann die Unentscheidbarkeit der Äquivalenzfunktion nachgewiesen werden, so dass dann auch die Minimierung von* Ausdrücken *unentscheidbar wird [20].*

Eine exakte *Modellanalyse* um *optimale* nichtreelwertige belegungsäquivalente *Ausdrücke* für *SPDS* zu bestimmen ist danach prinzipiell möglich. Praktisch ist diese aber zu aufwändig um die *Modellprüfung* zu beschleunigen. Denn allein die Bestimmung von realisierbaren *Variablenbelegung*en $RENV_l$ ist aufwändiger als die *Modellprüfung* für *Erreichbarkeit.*

Satz 4.3.3 (Komplexität von $RENV_l$)
Sei ein SPDS *S gegeben. Die Berechnung von $RENV_l$ hat mindestens den Aufwand der* Modellprüfung *$S \rightsquigarrow l$.*

Beweis (Skizze)
Die Marke l ist genau dann *erreichbar*, wenn es mindestens eine realisierbare *Variablenbelegung* an der Marke l gibt. D.h. es gilt $(S \rightsquigarrow l) \Leftrightarrow (RENV_l \neq \emptyset)$. Somit löst die Berechnung von $RENV_l$ bereits das *Erreichbarkeitsproblem.*

□

Weitere Untersuchungen dazu wurden in [68] veröffentlicht, welche zeigt, wie exakte *Variablenbelegung*en für eine Variable (so genannte exakte *Wertebereich*e) effizient in Polynomialzeit bestimmt werden können.

4.3.1.3. Einsetzbare Programmanalysen und Werkzeuge bzw. Techniken

Wegen Satz 4.3.3 (Komplexität von $RENV_l$) ist für die Beschleunigung der *Modellprüfung* eine Approximation nötig. Eine Möglichkeit ist, die möglichen realisierbaren *Variablenbelegung*en mittels einer *Intervallanalyse* [207] oder *Wertebereichsanalyse* zu schätzen (Überapproximation). Zur Schätzung des möglichen *Wertebereich*s einer *SPDS*-Variablen v kann sehr vereinfachend die Anzahl der **unterschiedlichen** *SPDS-Zuweisung*en an v verwendet werden. Wegen Schleifen und vielen möglichen *Anfangskonfigurationen* mit unterschiedlichen *Variablenbelegung*en ist eine solche Approximation jedoch sehr *unpräzise.* Im *Programmanalysenumfeld* werden zur Charakterisierung des *Wertebereich*s einer Variable u.a. so genannte Intervallanalysen eingesetzt. Intervallanalysen für Hochsprachen dienen dort z.B. zur Elimination überflüssiger Prüfungen vor Feldzugriffen (Array Bound Checks) [194]. Cousot, Halbwachs und Schwarz (et al.) nutzen dazu abstrakte *Interpretation* mit statischer Datenfluss-Analyse [228], um überflüssige Prüfungen von Feldzugriffen zu identifizieren. Für den Einsatz auf *SPDS* sind diese ggf. entsprechend wegen Ringarithmetik anzupassen, da *SPDS* nicht über Ringarithmetik verfügen. Verfügbare Werkzeuge bestimmen für Programmiersprachen i.d.R. intraprozedural *konservative* Intervallgrenzen, welche oft nicht die Erwartungen

an Qualität und Laufzeit der Benutzer genügen [228]. Denn zwar liefern interprozedurale kontextsensitive Analysen *präzise*re Ergebnisse, bedürfen dafür aber deutlich mehr Analysezeit, wie Experimente in [228] zeigen. Modelle in der *Modellprüfung* besitzen einen vergleichsweise kleinen *Konfigurationenraum* gegenüber gewöhnlichen *Programmen* aus Hochsprachen, so dass Analysen wenn möglich interprozedural und kontextsensitiv durchgeführt werden sollten. Aber ein immer noch aktuelles Problem gängiger Intervallanalysen ist jedoch die *Fehlabstraktion* der Semantik bei Ringarithmetik mittels unbeschränkter Datentypen, welches sich gerade bei kleineren Bitbreiten, wie sie bei der *Modellprüfung* eingesetzt werden, zum Tragen kommt. So missachten z.B. einige Intervallanalysen Ringarithmetik, wenn sie mittels linearer *Programme* Variablen-Grenzen bestimmen [234]. Sind aber im Modell unbeschränkte Ganzzahlen erlaubt, so kann für Fixpunktalgorithmen (eingesetzt bei *Programmanalysen*) keine Konvergenz mehr garantiert werden [180]. Mittels Aufweitung (widening) und Einengung (narrowing) [179, 180] oder Beschränkung der Anzahl an Fixpunktiterationen [194] kann Konvergenz nur noch mit einhergehendem Qualitätsverlust garantiert werden. Dies führt zu einer ungenauen aber sicheren Approximation der Analyseergebnisse [234]. Die in dieser Arbeit betrachteten *SPDS* verfügen über beschränkte Ganzzahlen, womit Konvergenz auch ohne Aufweitung und Einengung möglich ist. Nach [47] werden zudem z.B. Aufweitungs-Operatoren auch von Hand erstellt und müssen bei unzureichender Genauigkeit immer wieder neu angepasst werden. Delzanno [228] verzichtet auf Aufweitung und Einengung, was ermöglicht gegenüber von Ungleichungssystemen (z.B. Polyeder[5] [222, 47]) mehrere Intervalle je Variable gleichzeitig in Form des gesamten *Wertebereich*s zu berücksichtigen (nicht nur eine untere und eine obere Schranke [194]). Dadurch kann der gesamte *Wertebereich* einer *SPDS*-Variable charakterisiert werden.
Eine andere Möglichkeit, die möglichen realisierbaren *Variablenbelegung*en zu schätzen, besteht darin, statt *optimal*er *Ausdrücke* nur eine Menge von vereinfachenden Äquivalenzen wie $x - x = 0$ einzusetzen, welche i.d.R. zu einem kleinen aber möglicherweise nicht *optimal*en Ausdruck führt. Die Äquivalenzen gelten für beliebige *Variablenbelegung*en und sind Grundbestandteil von Computeralgebrasystemen, welche kleinere *alternative* *Ausdrücke* suchen mittels algebraischer Äquivalenztransformationen (kanonischer Simplifikator/canonical simplifier [9, 174, 82]). Oft werden dazu noetherische Reduktionssysteme eingesetzt (Konfluenz), welche zu einer Normalform führen [166].

Computeralgebrasysteme Computeralgebrasysteme ermitteln üblicherweise nicht *optimale* *Ausdrücke*, sondern wenden nur so lange *symbolische* Vereinfachungsregeln an wie möglich. Im Gegensatz zu einigen Programmanalysen für Programmiersprachen agieren sie **nicht** auf Ringarithmetik und sind daher auch für *SPDS* geeignet (ebenfalls keine Ringarithmetik). Sie können für *SPDS* genutzt werden, da die Semantik arithmetischer *Ausdrücke* für *SPDS* (Abschnitt 3.2.1 ab Seite 24) nicht mittels Ringarithmetik definiert wurde. Ringarithmetik ist üblich bei höheren *Programmiersprachen* und muss beim *Abstraktions*- bzw. Transformationsprozess vom *Programm* ins Modell entsprechend berücksichtigt werden.
Verschiedene Computeralgebrasysteme wie Mathematica [253] und Maple [149] sind teilweise in der Lage, für *Ausdrücke* auch an Hand von Nebenbedingungen (Belegung oder *Wertebereich* von Variablen[6]) bessere *alternative* *Ausdrücke* zu finden, als durch reine allgemeingültige *symbolische* Vereinfachungsregeln. Derartige dort verwendeten Algorithmen (viele kommerziell mit nicht publizierten Vereinfachungsverfahren[7] [8]) können auch hier zum Suchen besserer *alternativer* *Ausdrücke* eingesetzt werden.
Beim Einsatz eines Computeralgebrasystems werden evtl. die bitweisen Operatoren ! (bitweise Negation) und & (bitweises Und) nicht unterstützt. Sie können auf arithmetische zurückgeführt werden oder es kann auf sie verzichtet werden (*konservative* Approximation: für *Ausdrücke* mit bitweisen Operationen können dann nur Teilausdrücke ohne diese Operationen vereinfacht werden). Alternativ wird ein Computeralgebrasystem gewählt, welches bitweise Operationen unterstützt. Mathematica [253] und Maple [149] unterstützen z.B. bitweise Operationen. Jacal [18] und Yacas [19] hingegen nicht.

[5] engl. polyhedra
[6] z.B. FullSimplify[expr, assumptions] mit zusätzlichen Annahmen in Mathematica
[7] http://axiom-developer.org/axiom-website/rosetta.html
[8] http://www.worldlingo.com/ma/enwiki/de/Comparison_of_computer_algebra_systems

Quellcode 4.2: Via gewöhnliche Konstanten-Faltung/-Propagation nicht erkennbare Konstante y

```
x = 0;
L: y = x/2;
if
  :: (x = 0) -> x = 1; goto L;
  :: else -> break;
fi
```

Techniken wie sie meist für beliebige *Variablenbelegung*en in gängigen Computeralgebrasystemen integriert sind, sind unabhängig von Läufen des *SPDS*. Um auch für Läufe den kleinsten belegungsäquivalenten Ausdruck zu finden, können weitere Techniken von optimierenden Übersetzern **kombiniert** werden. Diese Techniken sind wegen der Berücksichtigung des *Programmablaufs* (Läufe im Sinne von *SPDS*) bzw. des Daten- und Kontrollflusses besser in der Lage, mögliche *Variablenbelegung*en vorher zu sagen.

Propagationen und Faltungen Konstanten-Propagation und -Faltung (constant propagation/folding [169, 59, 221, 57]) sowie Kopien-Propagation (Copy-Propagation [221]) sind ähnliche Techniken, um Konstanz von (Teil-)*Ausdrück*en zu identifizieren. I.d.R. wird die Faltung als vollständige Propagation mit zusätzlicher *Auswertung* konstanter *Ausdrücke* einschließlich Entfernung der Konstantendefinition verstanden [169, 59, 221, 57].

Bemerkung 4.3.2 *Der Ausdruck* $(10 + 2)/4$ *im Beispielcode 4.1 auf Seite 46 kann zu* 3 *durch eine direkte* Auswertung *(bzw. Teilauswertung, d.h. Faltung) und der Ausdruck* $x - 1$ *zu* 2 *durch Propagation des konstanten Werts von* $x = 3$ *und anschließender Faltung vereinfacht werden.*

Knoop und Rüthing erklären in [125] eine spezielle Konstanten-Propagation für *Quellcode* mit Prädikaten (SSA). Im Übersetzerbau wird aus der SSA-Darstellung Maschienencode erzeugt. Eine solche zusätzliche interne SSA-Darsstellung ist für die Modifikation von *SPDS* allerdings ungeeignet, da aus der SSA-Darstellung wiederum ein *SPDS* erzeugt werden muss. Sagiv und Reps et al. stellen lineare Konstanten-Propagation (linear constant propagation) [169, 236] vor und zeigen von ihr, dass sie nicht als Bit-Vektor-Problem formulierbar ist. Als Erweiterung der Konstanten-Propagation und -Faltung können Äquivalenzanalysen zum identifizieren äquivalenter *Ausdrücke* zum Vereinfachen oder Finden von Konstanten genutzt werden. Auch *Wertebereichsanalyse*n und Intervallanalysen können hier genutzt werden, um die Konstanz von *Ausdrücke*n fest zu stellen. Sehr *präzise* kann die Konstanz festgestellt werden, wenn (Teil-)*Ausdrücke* auf Erfüllbarkeitsprobleme abgebildet und exakt durch einen SAT- oder SMT-Löser gelöst werden. Wie die Beispiele 4.3.1 auf Seite 46 und jenes in *Quellcode* 4.2 zeigt, können Variablen konstant sein, auch wenn sie von nicht Konstanten *Variablenbelegung*en abhängen. So hat die Variable y im *Quellcode* 4.2 stets den Wert 0, was durch eine *Intervallanalyse* oder *Wertebereichsanalyse* erkannt wird, nicht jedoch durch Konstanten-Faltung oder -Propagation, da die Belegung von den nicht konstanten *Variablenbelegung*en $x = 0$ und $x = 1$ abhängen. Daher können gewöhnliche Konstanten-Faltungen oder -Propagation aus dem Übersetzerbau diese Konstanz nicht erkennen.

SAT- und SMT-Löser So können boolesche *Ausdrücke* z.B. eine Tautologie darstellen oder unerfüllbar sein (für realisierbare *Variablenbelegung*en). Arithmetische *Ausdrücke* können bitweise als boolesche *Ausdrücke*, d.h. als boolesche Funktion mit mehreren Ausgängen (und Eingängen) angesehen werden. Durch einen SAT-Löser (Satisfiability Problem Solver) bzw. SMT-Löser (Satisfiability Modulo Theorie Solver) sind dann noch bessere *alternative* (Teil-)*Ausdrücke* findbar. Auch für SAT- und SMT-Löser sind bitweise Operatoren evtl. auf zu lösen. SAT- und SMT-Löser bieten den entscheidenden Vorteil sehr effizient die Erfüllbarkeit von booleschen *Ausdrücke*n prüfen zu können, wenn viele teils komplexe Nebenbedingungen zu berücksichtigen sind.

Verfahren der Logiksynthese Schließlich ist ein boolescher Ausdruck aber auch eine Repräsentation für eine boolesche Funktion mit mehreren Eingängen (*Variablenbelegung*en). Ein arithmetischer Ausdruck kann bitweise als Menge von booleschen Funktionen aufgefasst werden, so dass in beiden Fällen Verfahren der Logiksynthese zur Minimierung boolescher Funktionen eingesetzt werden können. Dann werden so genannte Don't Cares (Werte außerhalb des Definitionsbereichs der Funktion) [181] für nichtrealisierbare *Variablenbelegung*en genutzt, da deren Funktionswert unwichtig ist für die Schaltung. Zu Gunsten einer kleineren Schaltung dürfen die Funktionswerte für Don't Cares beliebig belegt sein. I.d.R. sind Schaltungen derart groß, dass heuristische Verfahren zum Finden kleiner Schaltungen genutzt werden. Für die Logiksynthese sind daher viele gute heuristische Verfahren bekannt [181], welche zur Ausdrucksvereinfachung genutzt werden können.

4.3.1.4. Modellanalysebeispiele

Im Folgenden werden ausgewählte Techniken miteinander kombiniert und für eine Realisierung genauer betrachtet, um kleinere *alternative Ausdrücke* zu bestimmen.

Idee Da *Ausdrücke* in *SPDS* eher klein sind, kommen für die Praxis die Techniken zur Logiksynthese eher nicht in Frage, da sie auf Effizienz für große Schaltungen ausgelegt sind. Favorisiert werden eher exakte Techniken wie SMT-Löser, um möglichst minimale *alternative Ausdrücke* zu finden, da diese SMT-Solver wie YICES[9] [41] bereits sehr effizient sind. Gleiches gilt für Computeralgebrasysteme. Mittels des Projekts *remote-maxima*[10] kann das Computeralgebrasystem Maxima zur Suche nach *alternative Ausdrücke*n für Experimente eingebunden werden. Maxima ist ein sehr ausgereiftes und umfangreiches, altes und nicht länger kommerzielles Computeralgebrasystem [166]. Kombiniert werden daher...

1. eine interprozedurale Konstanten-Faltung,
2. das Computeralgebrasystem Maxima [158] und
3. der SMT-Löser YICES [41].

In allen drei Fällen werden mögliche *Variablenbelegung*en von Variablen genutzt, um die Präzision der Analyseergebnisse zu steigern (kleinere *alternative Ausdrücke*).

Konkretisierung Da die Berechnung realisierbarer *Variablenbelegung*en $RENV_l$ sehr aufwändig ist, werden die realisierbaren *Variablenbelegung*en abgeschätzt. Im einfachsten Fall kann dies durch die triviale Abschätzung $0 \leq v < 2^{bits(v)}$ erfolgen, da der Typ $bits(v)$ und damit auch der *Wertebereich* $range(v)$ statisch bekannt sind. Besser geeignet ist jedoch die Verwendung einer *Wertebereichsanalyse*.

Wertebereichsanalyse Es wird eine *Wertebereichsanalyse* vorgestellt, welche zum Ziel hat, mögliche *Variablenbelegung*en $values_l \subseteq ENV$ an einer Marke $l \in Marken(S)$ zu bestimmen. Für die *Variablenbelegung*en $values_l \subseteq ENV$ wird eine Obermenge an Variablenbelegungen $In_l \subseteq ENV$ bestimmt, welche (mindestens) alle realisierbaren *Variablenbelegung*en an der Marke l enthält. Jeder realisierbare Variablenwert während eines Laufs wird durch die Analyse vorhergesagt. Es können durch die Analyse Werte vorhergesagt sein, welche nicht mittels Läufe realisierbar sind (*Konservativität* bei nicht exaktem $values(v)@l$). An der Marke l eingehende[11] *Variablenbelegung*en In_l werden mittels der SPDS-*Anweisung* an der Marke l *abstrakt* interpretiert zu ausgehenden *Variablenbelegung*en $Out_l \subseteq ENV$. Mittels In_l und Out_l wird ein Gleichungssystem erstellt, welches durch einen Fixpunktalgorithmus gelöst werden kann. Das prinzipielle Vorgehen ist dabei analog zur Bestimmung erreichender *Zuweisung*en (engl. *Reaching Definitions*) bei einer vorwärts gerichteten Kann-Analyse [163]. Analog zu Reaching Definitions gibt es mehrere Möglichkeiten, wie realisierbare *Variablenbelegung*en entlang des Kontrollflusses oder interprozedural über Prozedurgrenzen „fließen“ (siehe Abbildung 4.2). Man unterschiedet dabei...

[9] Gesamtsieger des SMT-Wettbewerbs im Rahmen der CAV 2007 (Computer-Aided-Verification).
[10] http://code.google.com/p/remote-maxima/
[11] Auch als *JOIN* bezeichnet [163].

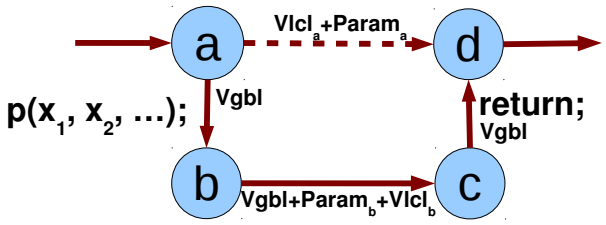

Abbildung 4.2.: Unterschiedliche Typen von Kontrollflusskanten und deren Variablenflüsse

... intraprozedurale *Konfigurationenübergänge* $IntraJoin_l$ (z.B. *Zuweisung* oder Verzweigung, Kante (b, c))

... *Prozeduraufrufe* $CallJoin_l$ (Kante (a, b))

... Prozedurbeendigung $RetJoin_l$ (*return*; Kante (c, d)) sowie

... Prozedurrückkehr $ContJoin_l$ (lokale Variablen bei *Prozeduraufruf*, Kante (a, d)).

Für jeden dieser Fälle wird im Folgenden definiert, wie die Analyseergebnisse zu berechnen sind.

Definition 4.3.4 (*Join-Definitionen für* Variablenbelegungen)
Sei $CFG \subseteq Marken(S)^2$ *der intra- und* $CFG' \subseteq Marken(S)^2$ *der interprozedurale Kontrollflussgraph eines* SPDS S *und* $l \in Marken(S)$ *mit ausgehenden* Variablenbelegungen $Out_l \subseteq ENV$. *Dann ist* $IntraJoin_l$ *die Vereinigung aller* Variablenbelegungen *aller Variablen* $(Param_k, Vlcl_k, Vgbl)$ *von Vorgängerknoten* k *des Kontrollflusses*[12]*:*

$$IntraJoin_l := \bigcup_{(k,l)\in CFG} Out_k$$

Um eine Menge von Variablenbelegungen $E \subseteq ENV$ *beim* Prozeduraufruf *auf eine Auswahl* $A \subseteq Vars$ *einzuschränken (z.B. globale Variablen) und Belegungen von neuen (lokalen) Variablen* $B \subseteq Vars$ *zu initialisieren, wird die Schreibweise* $E\big|_A^B$ *verwendet*[13]*:*

$$E\Big|_A^B := \{env_{A,B} \mid env_A \in E,\ env_B \in ENV\}$$

Weiter sei $Call \subseteq CFG'$ *die Menge der Kanten* (a, b) *mit einem* Prozeduraufruf *der Form „*$a : p(x_1, x_2, \ldots, x_n);$*“* [14]*. Bei einem solchen* Prozeduraufruf *werden die Parametervariablen* $p_i \in Param_p$ *der Prozedur* $p \in Prz(S)$ *entsprechend der Variablen* $x_i \in Vars$ *des* Prozeduraufrufs *„*$a : p(x_1, x_2, \ldots, x_n);$*“ belegt*[15]*:*

$$Out_a^{Call} := \{env|_{p_1}^{[\![x_1]\!]_{env}}|_{p_2}^{[\![x_2]\!]_{env}} \ldots |_{p_n}^{[\![x_n]\!]_{env}} \mid env \in Out_a\}$$

Bei einem Prozeduraufruf *bleiben die* Variablenbelegungen *globaler Variablen erhalten. Die Parametervariablen* p_i *werden gemäß der Variablen* x_i *belegt* (Out_a^{Call}) *und lokale Variablen werden mit beliebigen Werten initialisiert. Daher besteht* $CallJoin_l$ *aus allen* Variablenbelegungen *globaler Variablen (Vgbl) von Vorgängerknoten des Kontrollflusses und der Initialisierung lokaler Variablen* $Vlcl_l$ *bzw.* $Param_l$*:*

$$CallJoin_l := \bigcup_{(k,l)\in Call} Out_k^{Call}\Big|_{Vgbl}^{Vlcl_l}$$

[12] vgl. Kante (b, c) in Abbildung 4.2

[13] Für $x \in B$ gilt: $\{[\![x]\!]_{env} \mid env \in E\big|_A^B\} = range(x)$.

[14] vgl. Kante (a,b) in Abbildung 4.2

[15] gemäß Semantikdefinition des *Prozeduraufrufs* (C) aus Abschnitt 3.2.2 auf Seite 28

Sei $Ret \subseteq CFG'$ *die Menge der Kanten* (c,d) *mit Prozedurende „*$c : return;$*“*[16]*. Bei einem Prozedurende bleiben nur die* Variablenbelegung*en globaler Variablen erhalten:*

$$RetJoin_l := \bigcup_{(k,l)\in Ret} Out_k\Big|^{\emptyset}_{Vgbl}$$

Sei $Cont := \{(a,d) \mid (a,b) \in Call, (c,d) \in Ret, \exists p,q \in Prz(S) : (b,c \in Marken(p) \wedge a,d \in Marken(q))\}$ *die Menge der Kanten von einem* Prozeduraufruf *p bis zur Prozedurrückkehr der aufrufenden Prozedur* q[17]*. In diesem Fall bleiben lokale Variablen (einschließlich Parametervariablen* $Param_q$ *der Prozedur* q*) erhalten. D.h. es ist:*

$$ContJoin_l := \bigcup_{(k,l)\in Cont} Out_k\Big|^{\emptyset}_{Param_l \cup Vlcl_l}$$

Mit diesen Hilfsmitteln kann nun die *Wertebereichsanalyse* wie folgt mittels Datenflussgleichungen [163] definiert werden.

Definition 4.3.5 (*Wertebereichsanalyse*)
Sei $l \in Marken(S)$ *eine Marke eines* SPDS S*. Über die eingehenden bzw. ausgehenden* Variablenbelegung*en* $In_l \subseteq ENV$ *bzw.* $Out_l \subseteq ENV$ *werden folgende Datenflussgleichungen definiert. Eine* Zuweisung *„*$l : x = e$*“ wird so interpretiert, dass die Variable* $x \in Vars$ *sämtliche mögliche Wertebelegungen des Ausdrucks* $e \in Expr$ *annehmen kann:*

$$Out_l = \{env|_x^{[\![e]\!]_{env}} \mid env \in In_l\}$$

*Eine Choose-*Zuweisung *„*$l : x = choose(a,b);$*“ wird so interpretiert, dass die Variable* $x \in Vars$ *sämtliche mögliche Werte* $i \in [a,b] \cap range(x)$ *annehmen kann:*

$$Out_l = \{env|_x^i \mid env \in In_l,\ i \in [a,b] \cap range(x)\}$$

Für einen Prozeduraufruf *„*$l : p(x_1, x_2, \ldots, x_n);$*“, ein Prozedurende „*$l : return;$*“ und eine Verzweigung „*$l : if\ (e)\ goto\ l';$*“ gilt jeweils* $Out_l = In_l$*, da bei diesen SPDS-*Anweisungen *keine Variablenwerte verändert werden.*
Zusammenfließende bzw. eingehende Variablenbelegung*en werden vereinigt (*Konservativität*):*

$$In_l = IntraJoin_l \cup CallJoin_l \cup RetJoin_l \cup ContJoin_l.$$

Dann ist $values_l \subseteq ENV$ *die kleinste Lösung für* In_l *dieses Gleichungssystems (Fixpunkt).*

Die *Wertebereichsanalyse* ist *konservativ*, da es wegen Überapproximation der Läufe zu größeren *Variablenbelegung*en $values_l$ kommen kann. Ein Beispiel zur konkreten Realisierung einer solchen *Wertebereichsanalyse* findet sich als Anlage auf beiliegender CD. Dort werden die Fixpunktberechnung als Iterationsforschrift definiert und Mengen von *Variablenbelegung*en kompakt durch Wertemengen repräsentiert (sortierte disjunkte Intervallüberdeckung als kanonische Darstellung). Diese *Wertebereichsanalyse* ist zudem direkt auf *SPDS*-Erweiterungen anwendbar (insb. *Parallelzuweisung*en, ohne zusätzlichen *Interpretation*scode). Mit einer solchen *Wertebereichsanalyse* lassen sich die realisierbaren *Variablenbelegung*en $RENV_l$ *präzise*r vorhersagen (Überapproximation).

Lemma 4.3.1 (*Konservative* Approximation realisierbarer *Variablenbelegung*en)
Sei ein SPDS S *und eine Marke* l *gegeben. Dann kann* $RENV_l$ *durch eine* Wertebereichsanalyse $values_l$ konservativ *approximiert werden:*

$$RENV_l \subseteq RENV_l^* := \{env \in ENV \mid \forall v \in Vars : [\![v]\!]_{env} \in values_l(v)\}$$

Beweis (Skizze) Folgerung aus der *Konservativität* der *Wertebereichsanalyse* $values_l$. □

Diese *Variablenbelegung*en können dem Computeralgebrasystem oder dem SMT-Löser direkt durch Angabe der Ungleichungen der *Wertebereichsanalyse* $values_l$ übergeben werden. Die Approximation der realisierbaren *Variablenbelegung*en ergibt sich dann indirekt (*symbolisch*) an Hand der Analyseergebnisse der *Wertebereichsanalyse*.

[16] vgl. Kante (c,d) aus Abbildung 4.2
[17] Kante (a,d) in Abbildung 4.2

Konstanten-Faltung Da Konstanten nicht weiter ausgewertet werden brauchen, wird zunächst versucht, *Ausdrücke* zu Konstanten zu vereinfachen. Dabei werden Ausdrücke auf Ihren Wertebereich hin untersucht:

Definition 4.3.6 (*Wertebereich* $range_l(e)$ eines Ausdrucks $e \in Expr$)
Sei $e \in Expr$ ein arithmetischer Ausdruck und $B \subseteq ENV$ eine Menge von Variablenbelegung*en. Dann ist der* Wertebereich *des Ausdrucks e:*

$$range_B(e) := \{[\![e]\!]_{env} \mid env \in B\}$$

Im Folgenden sei abkürzend $range_l(e) := range_{RENV_l}(e)$.

An jeder Marke l wird jeder vorkommende Teilausdruck $e \in l : s \cap Expr$ betrachtet und auf Konstanz geprüft. Initial werden die *alternativ*en *Ausdrücke* (abstrakten Informationen) $\tau_l(e) = e$ gewählt und dann schrittweise weiter verbessert. Begonnen wird mit dem größten Teilausdruck und mit der Suche bei kleineren Teilausdrücken rekursiv fortgesetzt. Es wird jeweils der *Wertebereich* des Ausdrucks analysiert und entsprechend *konservativ* approximiert mittels konservativer Approximation realisierbarer *Variablenbelegung*en:

Lemma 4.3.2 (*Konservativ* approximierter *Wertebereich* $range_l^*(e)$)
Gemäß Definition 4.3.5 ist für Marken l und Ausdrücke *e:*

$$range_l(e) \subseteq range_l^*(e) := range_{RENV_l^*}(e) = [\![eval(e, values_l)]\!]$$

ein konservativ *approximierter* Wertebereich.

Beweis (Skizze)
Die *Konservativität* des approxmierten Wertebreicheichs folgt analog aus der Konservativität der Approximation realisierbarer *Variablenbelegung*en.

□

Ein Ausdruck e (analog Teilausdruck) ist dann konstant, wenn sein *konservativ* approximierter *Wertebereich* $range_l^*(e) = \{c\} \subseteq \mathbb{N}_{\geq 0}$ aus nur einer Zahl $c \in \mathbb{N}_{\geq 0}$ besteht ($|range_l^*(e)| = 1$). Es kann allerdings wegen *Konservativität* des *Wertebereich*s vorkommen, dass konstante *Ausdrücke* nicht als konstant erkannt werden. Andererseits ist auch $|range_l^*(e)| = 0$ möglich. In diesem Fall gibt es keine realisierbare *Variablenbelegung* für mindestens eine Variable, welche im Ausdruck vorkommt. Dann kann $\tau_l(e) = c$ für eine beliebige Konstante gewählt werden (*Konservativität* von $RENV_l^*$). Sobald ein Teilausdruck e' als konstant erkannt wurde ($|range_l^*(e)| = 1$), wird die rekursive Suche in diesem Teil des Ausdrucks e beendet und die gefundene Verbesserung $\tau_l(e') = c$ als *alternativ*er Ausdruck gespeichert. Da die *Wertebereichsanalyse* interprozedural *Wertebereiche* bestimmt, wird auch diese Form der Konstanten-Faltung als interprozedural bezeichnet.

Computeralgebrasystem Maxima Mit Maxima können arithmetische und boolesche *Ausdrücke* $e \in Expr$ vereinfacht werden. Untersützt werden mit dem Paket *functs* auch bitweise Operatoren via Funktionen *logand* (bitweises UND), *logor* (bitweises ODER), etc. Nicht jedes Computeralgebrasystem verfügt über Bitweise Operatoren. Falls nicht, können sie wie folgt interpretiert werden:

Definition 4.3.7 (*Interpretation* bitweiser mittels arithmetischer Operatoren)
Sei $[\![e_i]\!]_{env} := [\![(e >> i)\%2]\!]_{env}$ eine abkürzende Schreibweisezur Beschreibung des i-ten Bits eines Ausdrucks $e \in Expr$. Es ist stets $a_i \in \{0,1\}$ und es gilt für hinreichend großes[18] $n = \max_{b \in RENV_l}[\log([\![e]\!]_b)]$*:*

$$[\![!a]\!]_{env} := [\![\sum_{i=0}^{n}(1 - a_i) << i \,]\!]_{env}$$

$$[\![a\&b]\!]_{env} := [\![\sum_{i=0}^{n}((a_i + b_i)/2) << i \,]\!]_{env}$$

[18] Die Konstante n kann auch mittels einer *Wertebereichsanalyse* abgeschätzt werden.

Damit reduzieren sich die Basisoperationen von *SPDS* auf $Op = \{-, *, /, <\}$, welche in jedem gebräuchlichen Computeralgebrasystem *Verwendung* finden. Allerdings kann hierdurch u.U. keine Rücktransformation gerantiert werden, wenn die Summen zu stark verändert sind. Bei Maxima ist dieses Problem nicht vorhanden. Maxima wird direkt genutzt zum Suchen eines einfacheren Ausdrucks e' für *Ausdrücke* $e \in Expr$ an der Marke l sowie in den *alternativen Ausdrücken* (abstrakten Informationen) $\tau_l(e)$. Die Suche wird rekursiv dann auf Teilausdrücken fortgesetzt, wenn e nicht vereinfacht werden konnte.
Zur Übergabe an Maxima werden *Ausdrücke* vollständig geklammert und die *SPDS*-Operatoren $-, *, <$ bzw. & und ! auf die Maxima-Operatoren $-, *, <$ bzw. auf die Funktionen *logand* und *lognot* abgebildet (analog die daraus ableitbaren Operatoren). Die *SPDS*-Division / ist eine Ganzzahldivision und unterscheidet sich von der Gleitkomma-Maxima-Division /. Zur *Interpretation* der *SPDS*-Division a/b ohne Rest kann dann zusätzlich die Maxima-Funktion $floor$ zum Abrunden einsetzt werden (Verwerfen des Rests), so dass aus dem *SPDS*-Ausdruck a/b der Maxima-Ausdruck $floor(a/b)$ wird. Eine Rückinterpretation des vereinfachten Ausdrucks e' erfolgt nur dann, wenn direkt nach jeder Maxima-Division in e' auch eine $floor$-Funktion folgt. Experimente zeigen, dass dies häufig der Fall ist. Allerdings ist Vorsicht geboten bei *Ausdrücken* mit Division. Denn Ausdrücke wie a/a bzw. $(a+b)^2/(a+b)$ werden durch Maxima standardmäßig zu 1 bzw. $(a+b)$ ausgewertet für beliebige Belegungen für a und b. Für eine Belegung mit $a = b = 0$ werden diese *Ausdrücke* in *SPDS* wegen Division mit 0 aber zu $\perp$ ausgewertet und führen zur Beendigung eines Laufs des *SPDS*. Daher sind die *Ausdrücke* a/a und 1 sowie $(a+b)^2/(a+b)$ und $a+b$ nicht belegungsäquivalent. Eine Vereinfachung in so einem Fall führt dazu, dass mehr Läufe im *SPDS* möglich werden, welche zu neuen *False Negatives* führen können. Auch andere Computeralgebrasysteme wie Mathematica kürzen derartige Brüche zu Gunsten kleinerer Darstellungen, da aus mathematischer Sicht die Vergrößerung des Definitionsbereichs unerheblich ist. Aus diesen zwei Gründen wurde vom Einsatz der Division in Maxima abgesehen.
Paket *ineq* in Maxima enthält erweiterte Vereinfachungsregeln für Ungleichungen. Damit lässt sich dann z.B. der Ausdruck $e = (1+a^2) * (1/(1+a^2) <= 1)$ durch Maxima vereinfachen zu $1 <= a^2 + 1$ (Maxima erkennt, dass $1 + a^2 \geq 0$ ist). Allerdings unterscheidet sich an dieser Stelle die mathematische Semantik von der *SPDS*-Semantik für boolesche *Ausdrücke* mit Ungleichungen. Denn es lässt sich der Ausdruck e in der *SPDS*-Semantik nicht zu $1 <= a^2 + 1$ vereinfachen, sondern zu $(1 + a^2)$, da $[\![(1/(1+a^2) <= 1)]\!]_{env} = 1$ für beliebiges $env \in ENV$ (gleichgültig ob Ganzahl- oder Gleitkommadivision). Damit würde die *Verwendung* des Pakets *ineq* in Maxima zu falschen Vereinfachungen führen. Indem das Paket *ineq* aber nicht in Maxima geladen wird, werden solche Ungleichungen auch nicht weiter aufgelöst und damit auch nicht falsch vereinfacht. *SPDS-Ausdrücke* werden dann wie folgt in Maxima-Ausdrücke umgewandelt.

Definition 4.3.8 (*Konvertierung zu Maxima*-Ausdrücken)
Gegeben sei ein SPDS-*Ausdruck* e. *Dann ist der zugehörige Maxima-Ausdruck:*

$$M(e) \quad := \quad \begin{cases} c & \text{falls } e = c \in \mathbb{Z} \\ v & \text{falls } e = v \in Vars \\ logand(M(e_1), M(e_2)) & \text{falls } e = e_1 \& e_2 \\ lognot(M(e')) & \text{falls } e = !e' \\ (M(e_1) \ - \ M(e_2)) & \text{falls } e = e_1 - e_2 \\ (M(e_1) \ * \ M(e_2)) & \text{falls } e = e_1 * e_2 \\ (M(e_1) < M(e_2)) & \text{falls } e = e_1 < e_2 \end{cases}$$

Die anderen Operatoren ergeben sich entsprechend der abkürzenden Schreibweisen. Es ist

$$\text{„}ev(M(e), simp)\text{“}$$

durch Maxima auszuwerten und damit e zu vereinfachen. Tritt innerhalb des Ausdrucks e ein bitweiser Operator auf, so ist zuvor das Paket $functs$ zu laden: „$load(functs); ev(M(e), simp)$“.

Die Funktion *ev* dient in Maxima zum *Auswerten* von *Ausdrücken*. Das Argument *simp* gibt an, das der Ausdruck $M(e)$ dabei zu vereinfachen ist. Das Resultat ist entweder ein Fehler oder ein Ausdruck e'. Es wird dann $\tau_l(e) = e'$ gewählt, wenn e' kleiner ist als e ($|e'| < |e|$).

Nicht immer sind Computeralgebrasysteme wie Maxima in der Lage, minimale *Ausdrücke* zu finden. So ist Maxima z.B. nicht in der Lage den Ausdruck $(x^2 - 1)/(x - 1) - (x + 1)$ für $x \neq 1$ zu 0 zu vereinfachen. Insbesondere für boolesche *Ausdrücke* an Verzweigungen, kann die *Modellprüfung* verbessert werden, indem ein solcher Ausdruck durch sehr mächtige Methoden als konstant (1 bzw. 0, d.h. *true* bzw. $false$) erkannt werden kann. So lassen sich dann nichtrealisierbare Kontrollflüsse besser statisch ausschließen.

Äquivalenzanalyse Um das Analyseergebnis des SMT-Lösers zu verbessern, wird neben des approximierten *Wertebereichs* für Variablen zusätzlich die Äquivalenz von Variablen betrachtet. Damit kann auch die Konstanz des Ausdrucks $x > y$ in folgendem Beispiel erkannt werden ($[\![x > y]\!]_{env} = 0$),

```
      x=choose(0,3);
      y=x;
l:    if (x>y) goto l;
```

Denn eine Äquivalenzanalyse kann die Äquivalenz von x und y erkennen. Konstanten-Propagation kann diese Konstanz nicht erkennen, da mit den approximierten *Wertebereich*en $0 \leq x \leq 3$ und $0 \leq y \leq 3$ die *Variablenbelegung* $x = 3$ und $y = 2$ zur Erfüllung des Ausdrucks $[\![x > y]\!]_{env} = 1$ und die *Variablenbelegung* $x = 3$ und $y = 2$ zur Nicht-Erfüllung des Ausdrucks $[\![x > y]\!]_{env} = 0$ führt.

Definition 4.3.9 (*Konservative Äquivalenzanalyse*)
Eine konservative *Äquivalenzanalyse bestimmt für ein* SPDS S *als abstrakte Information an jeder Marke l eine Relation $R_l \subseteq Expr^2$ zwischen* Ausdrücke*n, so dass für beliebige* Ausdrücke $e_1, e_2 \in Expr$ *und beliebige Marke $l \in Marken(S)$ gilt:*

$$(e_1, e_2) \in R_l \Rightarrow \forall env \in RENV_l : [\![e_1]\!]_{env} = [\![e_2]\!]_{env}$$

Lemma 4.3.3 (*Äquivalenzhülle*)
Sei M eine beliebige endliche Menge und $R \subseteq M^2$. Dann ist die symmetrische, reflexive, transitive Hülle $<R>$ *von R eine Äquivalenzrelation.*

Beweis (Skizze)
Trivial, da Äquivalenzrelationen reflexiv, transitiv und symmetrisch sind.

□

Im Folgenden wird eine Äquivalenzanalyse definiert für *Ausdrücke* $e \in Expr$. Es befinden sich z.B. zwei Ausdrücke e_1 und e_2 an der Marke l in der gleichen Äquivalenzklasse, falls deren Wert für jeden Lauf gleich ist $\forall p \in Post^*(S) : Marken(Kopf(p)) = \{l\} \Rightarrow [\![e_1]\!]_{env^p} = [\![e_2]\!]_{env^p}$. Diese Äquivalenzen werden später dem SMT-Löser YICES übergeben. Wie bei der *Wertebereichsanalyse* in Definition 4.3.5 auf Seite 53 „fließen“ ***konservativ*** realisierbare Äquivalenzen entlang des Kontrollflusses. Dabei werden wiederum durch Joins Kontrollflüsse gebildet, welche zu keinem realen Lauf gehören brauchen (Überapproximation). Damit ein endlicher Verband gebildet werden kann, wird angenommen, dass $\mathbb{Z}' := \{z \in \mathbb{Z} \mid |z| < range_{max}\}$ eine maximal nötige endliche Teilmenge ganzer Zahlen ist, wobei

$$range_{max} := \max_{e \in Expr(S), env \in ENV} [\![e]\!]_{env}.$$

Damit lässt sich auch für Ausdrücke ($Expr$ unendlich) eine endliche Teilmenge $Expr' \subseteq Expr$ bilden[19]. Die durch eine *Modellanalyse* für Marken $l \in Marken$ zu bestimmenden *abstrakten* Informationen $EqIn_l \subseteq Expr'^2$ sind Äquivalenzrelationen gemäß Definition 4.3.9 für Ausdrücke. Analog zur *Wertebereichsanalyse* (Abschnitt 4.3.1.4 ab Seite 51) werden dann unterschiedliche Kontrollflüsse berücksichtigt (siehe Abbildung 4.2 auf Seite 52).

[19] In Definition 3.2.3 auf Seite 26 wird dazu die erste Zeile $\mathbb{Z} \subset Expr$ geändert in $\mathbb{Z}' \subset Expr$.

Definition 4.3.10 (*Join-Definitionen für Ausdrucksäquivalenzen*)
Sei CFG der intra- und CFG' der interprozedurale Kontrollflussgraph eines SPDS *S und $l \in Marken(S)$ mit ausgehenden Ausdrucksäquivalenzen $EqOut_l \subseteq Expr'^2$. Dann ist $IntraEqJoin_l$ der Durchschnitt aller Äquivalenzen[20] aller Vorgängerknoten k des Kontrollflusses[21]:*

$$IntraEqJoin_l := \bigcap_{(k,l)\in CFG} EqOut_k$$

Um eine Äquivalenzrelation $E \subseteq Expr'^2$ bei einem Prozeduraufruf *auf eine Auswahl $A \subseteq Vars$ einzuschränken (z.B. globale Variablen) und Äquivalenzen von neuen (lokalen) Variablen $B \subseteq Vars$ zu unterbinden, wird die Schreibweise $E\big|_A^B$ verwendet:*

$$E\big|_A^B := E \cap \{e \in Expr' \mid Vars(e) \subseteq A\}^2 \cap \{e \in Expr' \mid Vars(e) \cap B = \emptyset\}^2$$

Bei einem Prozeduraufruf *„$k : p(x_1, x_2, \ldots, x_n)$; “ bleiben die Äquivalenzen für globale Variablen erhalten (Einschränkung auf Vgbl). Eine Parametervariable p_i wird äquivalent zur Variable x_i. Andererseits können Parametervariablen p_i und p_j äquivalent werden, wenn deren zugewiesenen Variablen x_i und x_j äquivalent sind ($(x_i, x_j) \in EqOut_k$). Äquivalenzen auf lokalen Variablen werden beim* Prozeduraufruf *unterbunden, da lokale Variablen bei Prozedureintritt beliebig belegt sein können:*

$$\begin{aligned} EqOut_k^{Call} \quad := \quad & < EqOut_k\big|_{Vgbl}^{Vlcl_p} \\ & \cup \{(x_i, p_i) \mid falls\ x_i \in Vgbl\} \\ & \cup \{(p_i, p_j) \mid falls\ (x_i, x_j) \in EqOut_k\} > \end{aligned}$$

Dann besteht $CallEqJoin_l$ aus dem Schnitt aller Äquivalenzen $EqOut_k^{Call}$ von Vorgängerknoten des Kontrollflusses:

$$CallEqJoin_l := \bigcap_{(k,l)\in Call} EqOut_k^{Call}$$

Bei einem Prozedurende „$c : return$; “ bleiben nur die Äquivalenzen für globale Variablen erhalten:

$$RetEqJoin_l := \bigcap_{(k,l)\in Ret} EqOut_k\big|_{Vgbl}^{\emptyset}$$

Im Falle einer Prozedurrückkehr bleiben Äquivalenzen auf lokalen Variablen (einschließlich Parametervariablen $Param_q$ der aufrufenden Prozedur q) erhalten. D.h. es ist:

$$ContEqJoin_l := \bigcap_{(k,l)\in Cont} EqOut_k\big|_{Param_l \cup Vlcl_l}^{\emptyset}$$

Nach einer Prozedurrückkehr besitzen die lokalen und Parametervariablen (gemäß Semantikdefinition des Prozeduraufrufs*) ihren alten Variablenwert wie vor dem* Prozeduraufruf*. Nur die globalen Variablen können sich geändert haben. Die Äquivalenzen geänderter globaler Variablen $RetJoin_l$ werden daher nach der Prozedurrückkehr mit den Äquivalenzen lokaler und Parametervariablen verbunden (Vereinigung von Ausdrücken über disjunkte Variablenmengen):*

$$RetContEqJoin_l := < RetEqJoin_l \cup ContEqJoin_l >$$

Mit diesen Hilfsmitteln kann nun die Äquivalenzanalyse wie folgt mittels Datenflussgleichungen [163] definiert werden.

[20] Bem: Der Schnitt zweier Äquivalenzrelationen $R_1 \cap R_2$ ist ebenfalls eine Äquivalenzrelation (Abgeschlossenheit).
[21] Dabei bleiben nur jene Äquivalenzen erhalten, welche durch jeden eingehenden Kontrollfluss erfüllt sind (*Konservativität*).

Definition 4.3.11 (*Äquivalenzanalyse*)
Sei $l \in Marken(S)$ eine Marke eines SPDS *S. Über die eingehenden bzw. ausgehenden Ausdrucksäquivalenzen $EqIn_l \subseteq Expr'^2$ bzw. $EqOut_l \subseteq Expr'^2$ werden folgende Datenflussgleichungen definiert. Eine* Zuweisung *„$l : x = e$“ wird so interpretiert, dass die Variable $x \in Vars$ zunächst ihre bisherige Äquivalenz verliert ($EqIn_l\big|_{Vars}^{\{x\}}$). Mit dem Ausdruck e wird dann die Äquivalenz $(x, e) \in EqOut_l$ induziert:*

$$EqOut_l = <\{(x, e)\} \cup EqIn_l\big|_{Vars}^{\{x\}}>$$

*Eine Choose-*Zuweisung *„$l : x = choose(a, b)$; “ wird so interpretiert, dass die Variable $x \in Vars$ nur ihre bisherige Äquivalenz verliert: $EqOut_l = EqIn_l\big|_{Vars}^{\{x\}}$.*
Für einen Prozeduraufruf *„$l : p(x_1, x_2, \ldots, x_n)$; “, ein Prozedurende „$l : return$; “ und eine Verzweigung „$l : if\ (e)\ goto\ l'$; “ gilt jeweils $EqOut_l = EqIn_l$, da bei diesen SPDS-*Anweisungen *keine Variablenwerte verändert werden.*
*Zusammenfließende bzw. eingehende Äquivalenzen werden geschnitten (*Konservativität*):*

$$EqIn_l = IntraEqJoin_l \cap CallEqJoin_l \cap RetContEqJoin_l.$$

Dann ist $Eq_l \subseteq Expr'^2$ die kleinste Lösung für $EqIn_l$ dieses Gleichungsystems (Fixpunkt).

Die *Modellanalyse* ist *konservativ*, da es an einer Marke $l \in Marken$ wegen des Schnittes der Äquivalenzrelationen eingehender Kontrollflüsse nur Äquivalenzen geben kann, welche auf jedem Lauf bei l realisiert werden. Damit kann es u.U. Äquivalenzen in Läufen geben, welche durch das *Modellanalyse*beispiel nicht erkannt werden (Überapproximation realisierbarer *SPDS*-Läufe).
Als Anlage auf beiliegender CD findet sich eine Konkretisierung für Ausdrücke der Größe $(0, 1)$ mit zugehöriger Implementation einer solchen Analyse.Dort wird wie bereits für die *Wertebereichsanalyse* geschehen ein Fixpunktverfahren definiert, welches auch für *SPDS*-Erweiterungen einsetzbar ist (insb. *Parallelzuweisung*en, ohne zusätzlichen *Interpretation*scode).
Zur besseren Lesbarkeit sei eine besser lesbare Syntax für Äquivalenzklassen definiert.

Definition 4.3.12 (*Schreibweise für Äquivalenzklassen*)
Seien $a_{i,j} \in Expr'$. Dann ist:

$$\begin{aligned} &[\![a_{1,1} = a_{1,2} = \cdots = a_{1,n_1} \quad a_{2,1} = a_{2,2} = \cdots = a_{2,n_2} \quad \ldots \\ &\qquad a_{m,1} = a_{m,2} = \cdots = a_{m,n_m}]\!] \\ := \quad &\{(a_{i,j}, a_{i,k}) \mid i \in \{1, 2..m\} \wedge j, k \in \{1, 2..n_i\}\} \subseteq Expr'^2 \end{aligned}$$

Weiter sei die Schreibweise $x \overset{A}{=} y$ für $x, y \in Expr'$ und $A \subseteq (Expr)^2$ verwendet, falls $(x, y) \in A$.

Es ist damit noch zu zeigen, dass so gebildete Mengen wirklich Äquivalenzrelationen darstellen, damit sie als Äquivalenzklassen bezeichnet werden dürfen.

Lemma 4.3.4 (Wohldefiniertheit der Äquivalenzklassenschreibweise)
Seien $a_{i,j} \in Expr'$. Dann ist $E := [\![a_{1,1} = a_{1,2} = \cdots = a_{1,n_1} \quad a_{2,1} = a_{2,2} = \cdots = a_{2,n_2} \quad \ldots a_{m,1} = a_{m,2} = \cdots = a_{m,n_m}]\!]$ eine Äquivalenzrelation.

Beweis (Skizze)
Reflexivität: Dies ergibt sich sofort durch Wahl von $k = j$ in der Menge $E = \{(a_{i,j}, a_{i,k}) \mid i \in \{1, 2..m\} \wedge j, k \in \{1, 2..n_i\}\}$.
Symmetrie: Sei ein $(a_{i,j}, a_{i',k}) \in E$ beliebig aber fest gewählt. Nach Konstruktion von E ist $i = i'$. Wähle nun $k' = j$ und $j' = k$. Wegen $(a_{i,j'}, a_{i,k'}) \in E$ ist dann auch $(a_{i,k}, a_{i,j}) \in E$.
Transitivität: Seien $(a_{i,j}, a_{i',k}), (a_{i'',j'}, a_{i''',k'}) \in E$ beliebig aber fest gewählt. Nach Konstruktion von E ist $i = i' = i'' = i'''$. Wähle nun $i''' = i, j'' = j$ und $k'' = k'$. Dann ist wegen $(a_{i,j''}, a_{i''',k''}) \in E$ auch $(a_{i,j}, a_{i''',k}) \in E$. Damit ist die Behauptung bewiesen.
□

Beispiel 4.3.2 (Äquivalenzklassen) Quellcode 4.3 *zeigt, wie sich damit Äquivalenzklassen kompakt darstellen lassen. Die Marke $m3$ besitze genau die zwei intraprozeduralen Vorgänger $m1$ und $m2$. Dann bestimmt die Äquivalenzanalyse die entsprechenden in den Kommentaren von* Quellcode 4.3 *enthaltenen Äquivalenzen.*

Quellcode 4.3: Beispiel zur Demonstration der Äquivalenzanalyse Eq_l.

```
      # Eq_m1: v1=s1 1=s0
m1:  goto m3, s0=s2, s1=s3, s2=s4;
...
      # Eq_m2: v0=s1 s2=s3=s4
m2:  goto m3, s0=s2, s1=s3, s2=s4;
...
      # Eq_m3: s1=s3 s2=s4
m3:  s0=e, s1=s0, s2=s1, s3=s2, s4=s3;
      # Eq_m4: e=s0 s2=s4
m4:  ...
```

SMT-Löser YICES Um boolesche *Ausdrücke* zu vereinfachen, kann der SMT-Solver YICES [41] verwendet werden. Der Einsatz von Maxima wäre zwar prinzipiell auch möglich, jedoch ist YICES deutlich effizienter und mächtiger. Es werden die *alternativen Ausdrücke* $\tau_l(e)$ durchsucht und weiter verbessert (zu Beginn der interprozeduralen Konstanten-Faltung wurde an jeder Marke l jeder vorkommende Teilausdruck $e \in l : s \cap Expr$ betrachtet, $\tau_l(e) = e$ gesetzt und verbessert.) Beginnend mit dem größten Teilausdruck wird wieder mit der Suche bei kleineren Teilausdrücken rekursiv fortgesetzt, so lange bis ein Teilausdruck als konstant identifiziert ist. Für jeden Teilausdruck werden dann jeweils zwei Erfüllbarkeitsprobleme formuliert (SMT-Probleme) und durch den SMT-Löser YICES gelöst. Zum einen wird der boolesche Ausdruck selbst und zum anderen wird seine Negation als SMT-Problem beschrieben. Im Gegensatz zu Maxima bietet YICES neben der Gleitkomma-Division / auch eine Ganzzahl-Division *div*, welche zur Übertragung auf ein SMT-Problem genutzt wird. Damit lässt sich das SMT-Problem für den SMT-Löser YICES wie folgt formulieren.

Definition 4.3.13 (*SMT-Problem eines Ausdrucks* $e \in Expr$)
Sei $e \in Expr$ ein Ausdruck an der Marke l. Seien $v_i \in Vars \cap e$ mit $1 \leq i \leq n$ sämtliche in e vorkommenden Variablen mit approximiertem Wertebereich $[\![values_l(v_i)]\!]$. *Dann sei $Pol(e)$ die polnische Notation des Ausdrucks e in YICES-Syntax über den Basisoperatoren $Op = \{-, *, /, <\}$:*

$$Pol(e) \quad := \quad \begin{cases} c & falls\ e = c \in \mathbb{N}' \\ v & falls\ e = v \in Vars \\ (-\ Pol(e_1)\ Pol(e_2)) & falls\ e = e_1 - e_2 \\ (*\ Pol(e_1)\ Pol(e_2)) & falls\ e = e_1 * e_2 \\ (div\ Pol(e_1)\ Pol(e_2)) & falls\ e = e_1/e_2 \\ (<\ Pol(e_1)\ Pol(e_2)) & falls\ e = e_1 < e_2. \end{cases}$$

Typdefinitionen werden in YICES-Syntax direkt übertragen und auf den approximierten Wertebereich *verfeinert. Sei dazu $V := \{(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)\} \in val$ und $Y(V) :=$*

$$\begin{aligned} &(subtype\ (x :: int)\ (or\ (or\ (\ \ldots\ (or \\ &(and\ (>=\ x\ a_1)\ (<=\ x\ b_1)) \\ &(and\ (>=\ x\ a_2)\ (<=\ x\ b_2))) \\ &\ \ldots \\ &(and\ (>=\ x\ a_m)\ (<=\ x\ b_m)))). \end{aligned}$$

Durch eine Äquivalenzanalyse identifizierte Äquivalenzen Eq_l werden ebenso in YICES-Syntax übertragen (polnische Notation). Sei dazu $Eq_l = [\![a_{1,1} = a_{1,2} = \cdots = a_{1,n_1} \quad a_{2,1} = a_{2,2} = \cdots =$

Quellcode 4.4: YICES SMT-Problem für den Ausdruck $x + z < y$

```
(define x::(subtype (n::int) (and (>= n 0) (<= n 3))))
(define y::(subtype (n::int) (and (>= n 0) (<= n 3))))
(define z::(subtype (n::int) (and (>= n 0) (<= n 3))))
(assert (< (+ x z) y))
```

Quellcode 4.5: YICES SMT-Problem $Y_l(x + z < y)$ mit approximiertem *Wertebereich*

```
(define x::(subtype (n::int) (and (>= n 3) (<= n 3))))
(define y::(subtype (n::int) (and (>= n 2) (<= n 2))))
(define z::(subtype (n::int) (and (>= n 0) (<= n 3))))
(assert (< (+ x z) y))
```

$a_{2,n_2} \quad \ldots \quad a_{m,1} = a_{m,2} = \cdots = a_{m,n_m}]\!]$ *und* $Y(Eq_l) :=$

$(assert\ (=\ Pol(a_{1,1})\ Pol(a_{1,2}))) (assert\ (=\ Pol(a_{1,1})\ Pol(a_{1,3}))) \ldots (assert\ (=\ Pol(a_{1,1})\ Pol(a_{1,n_1})))$
$(assert\ (=\ Pol(a_{2,1})\ Pol(a_{2,2}))) (assert\ (=\ Pol(a_{2,1})\ Pol(a_{2,3}))) \ldots (assert\ (=\ Pol(a_{2,1})\ Pol(a_{2,n_2})))$
$\ldots$
$(assert\ (=\ Pol(a_{m,1})\ Pol(a_{m,2}))) (assert\ (=\ Pol(a_{m,1})\ Pol(a_{m,3}))) \ldots (assert\ (=\ Pol(a_{m,1})\ Pol(a_{m,n_m})))$

Dann ist das zugehörige SMT-Problem definiert als $Y_l(e) :=$

$(define\ v_1 :: Y(values_l(v_1)))$
$(define\ v_2 :: Y(values_l(v_2)))$
$\ldots$
$(define\ v_n :: Y(values_l(v_n)))$
$Y(Eq_l)$
$(assert\ Pol(e))$

Ausdrücke können damit automatisch auf den *SPDS*-Operatoren

`! (boolesch), +, -, *, /, %, &&, ||, <, ==, >, <=, >= sowie !=`

erklärt sein (ohne bitweise Operatoren wie bitweises UND & oder Shifting <<). Die bitweisen Operatoren ! (bitweise Negation) und & (bitweises Und) werden gemäß Definition 4.3.7 auf Seite 54 auf arithmetische zurück geführt. Für einen Erfüllbarkeitstest ist dies im Gegensatz zur Ausdrucksminimierung via Computeralgebrasystemen problemlos möglich, da die Größe eines Ausdrücks hier nicht entscheidend ist.
Das konstruierte SMT-Problem aus Definition 4.3.13 wird an folgendem Beispiel veranschaulicht. Wie in *Quellcode* 4.4 zu sehen, werden ein gegebener boolescher Ausdruck $e = x + z < y$ (aus dem Beispielcode 4.1 auf Seite 46) sowie analog seine Negation $!e = !(x + z < y)$ als YICES-SMT-Problem beschrieben. Dabei wurde von der Verfeinerung des *Wertebereichs* und der Äquivalenzinformationen Eq_l zunächst abgesehen. Die ersten drei Zeilen beschreiben den Typ einer jeder in e vorkommenden Variable, welcher jeweils Werte im Bereich von 0 bis einschließlich 3 annehmen kann. Das SMT-Problem aus *Quellcode* 4.4 ist erfüllbar, indem $x = z = 0$ und $y = 1$ gewählt wird. Allerdings ist diese *Variablenbelegung* im Beispielcode 4.1 auf Seite 46 nicht realisierbar.
Wird das SMT-Problem allerdings bereits mit dem *konservativ* approximierten *Wertebereich* verfeinert (siehe *Quellcode* 4.5), so ist das SMT-Problem unerfüllbar. Daher ist $!e$ eine Tautologie und es kann $\tau_l(e) = 0$ $(false)$ gesetzt werden. Keine der Variablen x, y oder z sind an der Marke l äquivalent, so dass $Eq_l = \emptyset$ ist und daher kein weiteres *assert* im SMT-Problem von *Quellcode* 4.5 auftritt. Da $\tau_l(e) = 0$ $(false)$, kann die Verzweigung später samt zugehöriger *Konfigurationenübergänge* aus dem Modell entfernt werden (siehe *SPDS*-Beschneidung im Abschnitt 4.3.2 ab Seite 68).

Wie am Beispiel deutlich wird, sollten Konstanten bereits propagiert und gefalten werden bevor das SMT-Erfüllbarkeitsproblem formuliert wird. Dadurch wird das SMT-Erfüllbarkeitsproblem einfacher und kann effizienter gelöst werden.

Lemma 4.3.5 (SMT-Reduzierbarkeit boolescher *Ausdrücke*)
Sei $e \in Expr$ ein boolescher Ausdruck und $Y_l(e)$ das zugehörige SMT-Problem. Dann gilt:

$$Y_l(e) \text{ unerfüllbar} \Rightarrow \forall env \in RENV_l : [\![e]\!]_{env} = 0,$$
$$Y_l(!e) \text{ unerfüllbar} \Rightarrow \forall env \in RENV_l : [\![e]\!]_{env} = 1.$$

Ist demnach $Y_l(e)$ bzw. $Y_l(!e)$ unerfüllbar, so wird $\tau_l(e) = 0$ $(false)$ bzw. $\tau_l(e) = 1$ $(true)$ gesetzt. Dabei braucht allerdings YICES nur maximal einmal aufgerufen werden, wie folgendes Lemma zeigt. Denn gibt es durch einfache *Testauswertungen* bereits mehr als einen realisierbaren Wert für den Ausdruck $e \in Expr$, so kann weder $Y_l(e)$ noch $Y_l(!e)$ unerfüllbar sein.

Lemma 4.3.6 (Unerfüllbarkeitsvorbedingung für boolesche *Ausdrücke*)
Sei eine sehr kleine Konstante $q \in \mathbb{N}_{>1}$ fest gewählt. Nun seien für eine Marke l z.B. heuristisch oder rein zufällig Variablenbelegung*en $env_1, env_2, \ldots, env_q \in RENV_l^*$ gewählt, so dass $\forall i : \forall x, y \in Vars : (x \overset{Eq_l}{=} y) \Rightarrow ([\![x]\!]_{env_i} = [\![y]\!]_{env_i})$. Existiert dann ein i, so dass $[\![e]\!]_{env_i} = 0$ bzw. $= 1$, so ist $Y_l(!e)$ erfüllbar bzw. $Y_l(e)$ erfüllbar.*

Entsprechend können $Y_l(e)$ bzw. $Y_l(!e)$ nur dann unerfüllbar sein, wenn $\forall i : [\![e]\!]_{env_i} = 0$ bzw. $= 1$ ist. Ergeben einfache *Testauswertungen* $\forall i : [\![e]\!]_{env_i} = 0$, so erübrigt sich der Erfüllbarkeitstest $Y_l(!e)$ für YICES, da dieser bereits durch die Tests als erfüllbar erkannt ist. Analog erübrigt sich der Erfüllbarkeitstest $Y_l(e)$ für YICES, wenn $\forall i : [\![e]\!]_{env_i} = 1$. Beide Erfüllbarkeitstests können demnach entfallen, wenn die *Testauswertungen* verschiedene Werte liefern.

Um auch für arithmetische (Teil-)*Ausdrücke* $e \in Expr$ bessere *alternative* Ausdrücke zu finden, kann der approximierte *Wertebereich* $range_l^*(e)$ überprüft werden. Für ein $c \in range_l^*(e)$ wird dann $Y_l(e\ != c)$ auf Erfüllbarkeit geprüft.

Lemma 4.3.7 (SMT-Reduzierbarkeit arithmetischer *Ausdrücke*)
Sei $e \in Expr, c \in \mathbb{Z}$ und $Y_l(e\ != c)$ das zugehörige SMT-Problem. Dann gilt:

$$Y_l(e\ != c) \text{ unerfüllbar} \Rightarrow \forall env \in RENV_l : [\![e]\!]_{env} = c.$$

Ist $Y_l(e\ != c)$ unerfüllbar, so kann analog $\tau_l(e) = c$ gesetzt werden. Dieses $c \in range_l^*(e)$ wird analog zu Lemma 4.3.6 durch heuristisch oder rein zufällig gewählte *Variablenbelegung*en $env_1, env_2, \ldots, env_q \in RENV_l^*$ identifiziert.

Lemma 4.3.8 (Unerfüllbarkeitsvorbedingung für arithmetische *Ausdrücke*)
Sei eine sehr kleine Konstante $q \in \mathbb{N}_{>1}$ fest gewählt. Nun seien für eine Marke l z.B. heuristisch oder rein zufällig Variablenbelegung*en $env_1, env_2, \ldots, env_q \in RENV_l^*$ gewählt, so dass $\forall i : \forall x, y \in Vars : (x \overset{Eq_l}{=} y) \Rightarrow ([\![x]\!]_{env_i} = [\![y]\!]_{env_i})$. Existiert dann ein i, so dass $[\![e]\!]_{env_i} = c$ für ein $c \in \mathbb{Z}$, so ist $\forall c' \neq c : Y_l(e\ != c')$ erfüllbar.*

Wie für boolesche *Ausdrücke* kann dann für ein gefundenes c das SMT-Problem $Y_l(e\ != c)$ nur dann unerfüllbar sein, wenn wenn $\forall i : [\![e]\!]_{env_i} = c$ gilt. Auch für arithmetische *Ausdrücke* ist somit nur ein YICES SMT-Problem je Ausdruck auf Erfüllbarkeit zu prüfen.

Im Folgenden werden die Eigenschaften der vorgeschlagenen *Modellanalyse*beispiele erörtert.

Eigenschaften der Wertebereichsanalyse

Lemma 4.3.9 (*Wertebereichsanalyse* ist Monotones Rahmenwerk) [22]
$L = (2^{ENV}, \subseteq)$ ist ein Verband. Die Ausdrücke *zur Definition der Datenflussgleichungen von Out_l und In_l aus Definition 4.3.5 auf Seite 53 sind monoton.*

[22] engl. monotone framework

Der Beweis findet sich im Anhang B ab Seite 145.

Satz 4.3.4 (Lösbarkeit der *Wertebereichsanalyse*)
Die Datenflussgleichungen aus Definition 4.3.5 auf Seite 53 können mit einem Fixpunktverfahren gelöst werden.

Beweis (Skizze)
Dies ist eine Folgerung aus Lemma 4.3.9 und der Lösbarkeit monotoner Rahmenwerke, was für *Programmanalysen* bekannt ist [163].

□

Der Aufwand der Wertebereichsanalyse ist abhängig vom gewählten Fixpunktalgorithmus. Zur Berechnung von $values_l(v)$ wird daher als Beispiel der Aufwand näher untersucht für ein Fixpunktverfahren, welches in einem Iterationsschritt $(j \to j+1)$ alle Transfer- bzw. Datenflussgleichen für Out_l und In_l *auswertet*. In der Praxis genügt es dann, nur geänderte abstrakte Informationen via Worklist-Verfahren neu zu berechnen.
Der Aufwand für die Operation $E\Big|_A^B$ ist abhängig von der Implementation und sei mit $O(r)$ angenommen.

Bemerkung 4.3.3 *Eine gegebene Menge an* Variablenbelegung*en $E \subseteq ENV$ ist für $E\Big|_A^\emptyset$ einzuschränken auf eine Variablenauswahl $A \subseteq Vars$. Bei Realisierung z.B. mittels OBDDs kann die Einschränkung auf A durch die Operation restrict einer OBDD-Bibliothek erfolgen [210]. Diese hat einen Aufwand in der Größe m des OBDDs für E, d.h. $O(m)$, wenn zuvor aus A nur die E betreffenden Variablen ausgewählt wurden ($O(A)$). Die Variablen aus der Menge B sind beliebig zu belegen. Dazu wird für jede Variable in B der OBDD verwendet, welcher aus lediglich einem 1-Blatt besteht (jede Belegung der Variablen ist erfüllt bei der charakteristischen Funktion). Dieser OBDDs ist dann durch ein logisches Oder mit dem OBDD für $E\Big|_A^\emptyset$ zu verbinden. Jede einzelne dieser Operationen hat einen Aufwand von $O(m)$. Insgesamt lässt dich die Operation $E\Big|_A^B$ daher mit OBDDs in $O(r) = O(|A| + m \cdot |B|)$ realisieren.*

Die Join-Operationen für In_l können daher in $O(k_{max} \cdot r \cdot t)$ erfolgen, wenn t den Aufwand zur Vereinigung 2er *Variablenbelegung*smengen angibt.
Bei eingehenden *Variablenbelegung*en In_l ist zur Bestimmung von Out_l je nach SPDS-*Anweisung* unterschiedlicher Aufwand nötig. In den Fällen Prozedurende und Verzweigung ist $Out_l = In_l$ und damit der Aufwand $O(1)$. Beim *Prozeduraufruf* werden übergebene *Ausdrücke* genutzt, um Prozedurparameter zu initialisieren (Berechnung von Out_l^{Call}). Auch hier ist der Aufwand abhängig von der Implementierung der Analyse und durch $O(n \cdot |Out_l|) = O(n \cdot |ENV|)$ begrenzt. Eine *SPDS-Zuweisung* und die Choose-*Anweisung* verändern die *Variablenbelegung*en ähnlich, allerdings nur für eine Variable. Daher beträgt hier der Aufwand $O(|ENV|)$. Werden für einen Fixpunktalgorithmus zum Lösen der Datenflussgleichungen nun insgesamt j Iterationen angenommen, so führt dies zu einem Aufwand zur Bestimmung der $values_l(v)$ für alle Variablen und alle Marken von $O(|Marken(S)| \cdot |Vars| \cdot |ENV| \cdot j \cdot k_{max} \cdot r \cdot t)$.

Eigenschaften der Konstanten-Faltung Im Gegensatz zur Konstanten-Propagation von Rice [59, 57], terminiert die hier definierte interprozedurale Konstanten-Faltung für *SPDS* immer, insbesondere auch für rekursive *Programme*, da die definierte *Wertebereichsanalyse* stets gelöst werden kann (siehe Satz 4.3.4)[23]. Auch beschränkt sich die in diesem Abschnitt konstruierte Konstanten-Faltung nicht auf einfache lineare *Ausdrücke* der Form $y = ax + c$ $(x, y \in Vars, a, c \in \mathbb{Z})$ wie in [169], sondern versucht beliebig komplexe Ausdruck soweit wie möglich durch *symbolische Auswertung* (enthalten in der *Wertebereichsanalyse*) zu vereinfachen. Im Gegensatz zu einer gewöhnlichen Konstanten-Faltung können bei der Ausdrucksvereinfachung dann auch Variablen im Ausdruck

[23] Auch die Realisierung der *Wertebereichsanalyse* aus der Anlage auf beiliegender CD terminiert stets.

enthalten sein, wenn ein Ausdruck zu einer Konstanten ausgewertet werden kann. Durch die definierte Konstanten-Faltung können damit deutlich mehr konstante *Variablenbelegung*en erkannt werden, wie die Beispiele 4.3.1 auf Seite 46 und Quelltext 4.2 auf Seite 50 zeigen.
Der Rechenaufwand der Konstanten-Faltung ergibt sich einerseits aus der *Wertebereichsanalyse* und andererseits aus der Elementzählung des approximierten *Wertebereichs* $range_l^*(e)$. Der konkrete approximierte Wertebereich $[\![eval(e, values_l)]\!]$ braucht nicht bestimmt werden. Es kann die Frage $range_l^*(e) \stackrel{?}{=} \{c\}$ in polynomieller Zeit in der Größe des Ausdrucks bestimmt werden. Die *Wertebereichsanalyse* wertet derartige *Ausdrücke* sehr viel häufiger aus, so dass der Aufwand insgesamt durch die *Wertebereichsanalyse* (siehe Abschnitt 4.3.1.4 ab Seite 62) bestimmt wird.
Da die *Wertebereichsanalyse* wesentlicher Bestandteil der Konstanten-Faltung ist, übertragen sich aber auch die anderen Eigenschaften der *Wertebereichsanalyse* auf die Konstanten-Faltung. So ist die definierte Konstanten-Faltung ebenfall flusssensitiv, interprozedural, *konservativ*, kontextinsensitiv und pfadinsensitiv. Auch ist die Konstanten-Faltung entscheidbar, da (zumindest) formal alle realisierbaren *Variablenbelegung*en $RENV_l$ berechnet werden können. Ein Ausdruck $e \in Expr$ ist genau dann äquivalent zu einer Konstanten $c \in \mathbb{Z}$, wenn gilt: $range_l(e)=\{c\}$.

Eigenschaften zum Einsatz des Computeralgebrasystems Maxima Mittels des Projekts *remote-maxima*[24] kann das Computeralgebrasystem Maxima zur Suche nach *alternativ*en *Ausdrück*en eingebunden werden. In der Realisierung dieser Arbeit werden Orts- bzw. Laufunabhängig Ausdrucke vereinfacht. Die Analyse erfolgt daher flussinsensitiv, intraprozedural, kontextinsensitiv und pfadinsensitiv. Da keine minimalen *Ausdrücke* gefunden werden, ist die Analyse *konservativ*. Sie ist entscheidbar und ihr Aufwand ist Implementierungs- und Versionsabhängig. Üblicherweise erfolgt die Anwendung einer Reihe von Regeln bis zu einem lokalen Minimum (keine Fixpunktberechnung).

Eigenschaften zur Äquivalenzanalyse Wie auch die *Wertebereichsanalyse* kann auch die Äquivalenzanalyse durch eine komplexe Fixpunktberechnung realisiert werden.

Lemma 4.3.10 (Äquivalenzanalyse ist Monotones Rahmenwerk)
$L = (2^{Expr'^2}, \subseteq)$ ist ein Verband. Die Ausdrücke *zur Definition der Datenflussgleichungen von $EqOut_l$ und $EqIn_l$ aus Definition 4.3.11 auf Seite 58 sind monoton.*

Der Beweis findet sich im Anhang B ab Seite 145.

Satz 4.3.5 (Lösbarkeit der Äquivalenzanalyse)
Die Datenflussgleichungen aus Definition 4.3.11 auf Seite 58 können mit einem Fixpunktverfahren gelöst werden.

Beweis (Skizze)
Dies ist eine Folgerung aus Lemma 4.3.10 und der Lösbarkeit monotoner Rahmenwerke, was für *Programmanalysen* bekannt ist [163].

□

Im Allgemeinen ist das Äquivalenzproblem von *Ausdrück*en für gängige Hochsprachen unentschiedbar [90] (Reduktion auf das Postsche Korrespondenzproblem). Exakte Analyseergebnisse sind damit für Hochsprachen nicht immer bestimmbar. Dort wird es nur gelingen, eine sichere Abschätzung von Äquivalenzen zu erhalten. Entsprechend kann das Postsche Korrespondenzproblem natürlich **nicht** mittels Kellersystemen formuliert werden (anderfalls wäre das *Erreichbarkeitsproblem* für Kellersysteme nicht entscheidbar). Wie folgender Satz zeigt, sind aber exakte Äquivalenzklassen für *SPDS* berechenbar.

Satz 4.3.6 (Entscheidbarkeit der Äquivalenzanalyse)
Die Äquivalenzanalyse gemäß Definition 4.3.9 ist entscheidbar. D.h. es können a priori Äquivalenzen unter beliebigen Ausdrücken *$a, b \in Expr$ für jedes $l \in Marken$ als Relation $R_l \subseteq Expr^2$* ***exakt*** *berechnet werden, so dass $a \backsimeq_{RENV_l} b \Leftrightarrow (a, b) \in R_l$.*

[24] http://code.google.com/p/remote-maxima/

Beweis (Skizze)
Das Äquivalenzproblem kann, wie folgender *SPDS*-Ausschnitt zeigt, auf das *Erreichbarkeitsproblem* reduziert werden. In einem gegebenen *SPDS* und vor einer gegebenen Marke l werden ein neuer *Konfigurationenübergang* und zwei neue Marken $l1$ und $l2$ wie folgt eingeführt:

```
...
l1: if (a != b) goto l2;
l:...
l2: skip 0;
```

Um dann festzustellen, ob an einer Marke l der Ausdruck $a \in Expr$ stets äquivalent zu $b \in Expr$ ist, kann die *Erreichbarkeit* der neu eingeführten Marke $l2$ überprüft werden. Es gilt dann offensichtlich: a ist (immer) äquivalent zu b in der Marke l gdw. $l2$ ist nicht *erreichbar*.

□

Die in Definition 4.3.11 auf Seite 58 definierte Äquivalenzanalyse ist eine interprozedurale, flusssensitive, kontextinsesitive, pfadinsensitive Muss-Analyse (vgl. [221, S. 293ff]) und kann basierend auf dem Worklist-Algorithmus [49] berechnet werden.
Der Aufwand der Äquivalenzanalyse wird bestimmt durch den eingesetzten Fixpunktalgorithmus (analog zur *Wertebereichsanalyse*). Wie bei der *Wertebereichsanalyse* sei exemplarisch ein Fixpunktverfahren betrachtet, welches in jeder Iteration alle Transfer- bzw. Datenflussgleichen für $EqOut_l$ und $EqIn_l$ *auswertet*. In der Praxis werden diese Operationen effizienter (z.B. *symbolisch*) durchgeführt und es genügt dann auch nur geänderte abstrakte Informationen via Worklist-Verfahren neu zu berechnen.
Der Schnitt und die Vereinigung von Relationen $A, B \subseteq Expr'^2$ geht in Zeit $O(n)$, wobei $n = |Expr'^2|$ (bei sortierter und damit normalisierter Darstellung). Sei k_{max} wieder der maximale Verzweigungsgrad im Graphen des Kontrollflusses. Dann hat die Bestimmung von $IntraEqJoin_l$ den Aufwand $O(k_{max} \cdot n)$. Der Aufwand für die Operation $E\Big|_A^B$ ist $O(n)$, da hier Schnitte und Vereinigungen von Relationen berechnet werden (abhängig von der Implementation ggf. effizienter). Der Hüllenoperator $< \cdot >$ benötigt den Aufwand $O(n^2)$ (je nach Implementation im Erwartungswert sogar linear). Betrachtet sei nun ein einzelner Iterationsschritt eines Fixpunktverfahrens. Die Bestimmung von $EqOut_k^{Call}$ hat dann den Aufwand $O(n^2)$ und $CallEqJoin_l$ den Aufwand $O(k_{max} \cdot n^2)$. Analog ergibt sich der Aufwand für $RetEqJoin_l$ und $ContEqJoin_l$ mit jeweils $O(k_{max} \cdot n)$. $RetContEqJoin_l$ schließlich benötigt daher den Aufwand $O(n^2 + n + 2 \cdot k_{max} \cdot n^2) = O(k_{max} \cdot n^2)$. Damit ergibt sich der Aufwand für einen Iterationsschritt von $EqIn_l$ zu $O(k_{max} \cdot n + 2 \cdot k_{max} \cdot n^2) = O(k_{max} \cdot n^2)$. Analog ergibt sich der Aufwand für einen Iterationsschritt von $EqOut_l$ somit als $O(n^2)$.
Ein gesamter Iterationsschritt ($j \to j+1$) benötigt daher den Aufwand $O(|Marken(S)| \cdot k_{max} \cdot n^2)$. Für angenommene j Iterationsschritte bis zum Fixpunkt, ist damit wie bei der *Wertebereichsanalyse* insgesamt ein Aufwand nötig von $O(j \cdot |Marken(S)| \cdot k_{max} \cdot n^2)$.

Eigenschaften zum SMT-Löser YICES Die *Modellanalyse*eigenschaften der *Wertebereichsanalyse* und Äquivalenzanalyse werden durch dan SMT-Löser verwendet und übertragen sich darum direkt auf diese Form der *Modellanalyse*. So ist diese Suche nach *alternativ*en *Ausdrück*en mittels SMT-Löser YICES flussinsensitiv, interprozedural, *konservativ* entscheidbar, kontextinsensitiv und pfadinsensitiv. Eine Fixpunktberechnung findet nur indirekt durch die *Wertebereichsanalyse* und Äquivalenzanalyse statt.
Es könnten *Ausdrücke* dank *Interpretation* bitweiser Operatoren aus Definition 4.3.7 auf Seite 54 über alle *SPDS*-Operatoren erklärt werden. Für den Erfüllbarkeitstest wäre dies hier im Gegensatz zur Ausdrucksminimierung via Computeralgebrasystemen problemlos möglich, da die Größe eines Ausdrücks hier nicht entscheidend ist. Dennoch ist es für einen Ausdruck a^b ohne konkrete *Variablenbelegung* für b nicht möglich, zu exponentieren. Da YICES keine solchen Operationen (wie auch Shifting) anbietet, können solche (Teil-)*Ausdrücke* nicht behandelt werden.

Quellcode 4.6: Abstrakte Informationen des Motivationsbeispiels aus *Quellcode* 4.1.

```
    int x(2), y(2), z(2);
    ...
    x = (10+2)/4;          # tau((10+2)/4) = 3
    y = x - 1;             # tau(x-1) = 2
l:  z = choose(0,3);       # tau(0)=0, tau(3)=3
    if (x+z < y) goto l;   # tau(x+z)=3+z, tau(y)=2, tau(3+z<2)=0
    y = z - z;             # tau(z-z) = 0
```

Der Aufwand ergibt sich einerseits aus dem Aufwand für die *Wertebereichsanalyse* und die Äquivalenzanalyse. Und andererseits ergibt er sich aus dem beinhaltenden SAT-Problem, welches NP-vollständig in der Größe des Ausdrucks ist, da formal durch einen booleschen Ausdruck ein komplexes beliebiges Erfüllbarkeitsproblem beschrieben sein kann. Dieser Aufwand übersteigt die Formulierung des zugehörigen SMT-Problems. Experimente zeigen allerdings gutartig effizientes Verhalten, da *Ausdrücke* in den untersuchten Modellen i.d.R. nicht so komplex sind.

Abstrakte Informationen des Motivationsbeispiels *Quellcode* 4.6 zeigt in den Kommentaren mit den vorgestellten Techniken bestimmten *alternativen Ausdrücke* (ohne Marke als Subindex, da dieser aus dem Kontext klar ist). Dabei wurden $\tau((10+2)/4) = 3, \tau(x-1) = 2, \tau(x+z) = 3+z$ und $\tau(y) = 2$ durch die Konstanten-Faltung gefunden. Die *alternativen Ausdrücke* $\tau(3+z<2) = 0$ bzw. $\tau(z-z) = 0$ wurden mit dem SMT-Löser YICES bzw. mit Maxima gefunden. Maxima ist leider ohne das Paket *ineq* zum Vereinfachen für Ungleichungen nicht in der Lage, für den Ausdruck $3+x<2$ den besseren *alternativen* Ausdruck $x<-1$ zu finden.

4.3.1.5. Modellreduktion

Im vorigem Abschnitt 4.3.1.4 ab Seite 51 wurden Verfahren entwickelt, um einfachere belegungsäquivalente *alternative Ausdrücke* zu finden. Nun werden diese genutzt, um das *SPDS*-Modell zu verbessern.

Idee Im *SPDS* auftretende *Ausdrücke* werden syntaktisch durch einfachere belegungsäquivalente *alternative* Ausdrücke ersetzt.

Konkretisierung Seien ein *SPDS* S und für jeden darin enthaltenen (Teil-)Ausdruck e an Marke l ein zugehöriger *alternativer* Ausdruck $\tau_l(e)$ mit $|\tau_l(e)| \leq |e|$ gegeben. Dann umfasst die Ausdrucksvereinfachung T_{Expr} die *Modelltransformationsregel*:

$$\frac{l \in \mathit{Marken}(S) \quad e \in \mathit{Expr}(l) \quad \tau_l(e) \neq e}{e \;\Rrightarrow\; \tau_l(e)} \tag{4.2}$$

Sie wird so lange angewendet wie möglich.

Modelloptimierung des Motivationsbeispiels Werden so der Beispielcode 4.1 auf Seite 46 mit den bestimmten *alternativen Ausdrücken* aus *Quellcode* 4.6 transformiert, entsteht das verbesserte *SPDS* aus *Quellcode* 4.7. Die *Ausdrücke* wurden darin entsprechend τ vereinfacht.

Wohldefiniertheit Die *Modelltransformation* ist wohldefiniert, da lediglich *Ausdrücke* durch belegungsäquivalente *alternative* **kleinere** Ausdrücke ersetzt werden (Monotonie bei endlicher Modellgröße).

Quellcode 4.7: *SPDS*-Fragment aus *Quellcode* 4.1 nach der Ausdrucksvereinfachung.

```
    int x(2), y(2), z(2);
    ...
    x = 3;
    y = 2;
l:  z = choose(0,3);
    if (0) goto l;
    y = 0;
```

Korrektheit (Invarianz Temporaler Formeln und Erreichbarkeit)

Satz 4.3.7 (Korrektheit der Ausdrucksvereinfachung T_{Expr})
Die Ausdrucksvereinfachung T_{Expr} ist α-invariant für beliebiges $\alpha \in \{ACTL, ACTL\text{-}X, ACTL^*, ACTL^*\text{-}X, CTL, CTL\text{-}X, CTL^*, CTL^*\text{-}X, LTL, LTL\text{-}X\}$ und auch Erreichbarkeits-invariant *für eine Menge gegebener Marken $L0 \in Marken(S)$ eines* SPDS *S.*

Beweis Da sich durch die *Modelltransformation* (4.2) keine realisierbaren Variablenwerte ändern, bleibt nicht nur die *Erreichbarkeit* von *Fehlerkonfigurationen* erhalten, sondern auch die Aussagen von *temporalen Formel*n. Das Modellverhalten wurde nicht verändert.

□

4.3.1.6. Eigenschaften

Entscheidbarkeit der optimalen Transformation Die Existenz und Durchführbarkeit einer *optimal*en Transformation ist belegt durch die Existenz und Berechenbarkeit optimaler *Ausdrücke* fürτ_l (siehe Satz 4.3.1 auf Seite 47). Da durch die *Modelltransformationen* bereits sämtliche im Modell vorkommenden *Ausdrücke* betrachtet werden, kann es dann keine weiteren Ausdrucksvereinfachungen geben. Daher ist die Ausdrucksvereinfachung T_{Expr} bezüglich τ_l *präzise*, was bedeutet, dass durch T_{Expr} genau die durch τ_l ermittelten Ausdrücke substituiert wird. Die Modellreduktion T_{Expr} ist aber insbesondere deswegen auch genau dann *optimal*, wenn τ_l stets optimale *Ausdrücke* enthält.

Fehlalarme Ebenso wenig gibt es neue False Positives oder False Negatives, da sich das Modellverhalten durch die *Modelltransformation* T_{Expr} nicht ändert.

Konfigurationenraumveränderung Genauso wenig ändert sich daher natürlich auch der *Konfigurationenraum.*

Komplexität Die Bestimmung *alternativer Ausdrücke* benötigt je nach eingesetzter *Modellanalyse* i.d.R. deutlich mehr Zeit als die *Modelltransformation.* Sei $ex_l := |e_1| + |e_2| + ... + |e_j|$ die Gesamtlänge aller *Ausdrücke* an der Marke l (im Sinne der Anzahl von Operationen) und $ex_{max} := max_{l \in Marken(S)} ex_l$, wobei die $e_i \in Expr(l)$ alle an der Marke l im *SPDS* S vorkommenden *Ausdrücke* sind. Die *Modelltransformation* erfordert dann einem Aufwand $O(|Marken(S)| \cdot ex_{max})$. Im *Modellanalyse*beispiel beträgt der Aufwand gemäß voriger Betrachtungen bereits deutlich mehr und bestimmt damit den Gesamtaufwand des Verfahrens von $O(j \cdot |Marken(S)| \cdot |Vars| \cdot (ex_{max} + k_{max} \log |Vars|))$.

Aufwand vs. Nutzen Wie gezeigt, ist die exakte Minimierung arithmetischer *Ausdrücke* (Bestimmung *optimaler alternativer* Ausdrücke) zwar prinzipiell theoretisch möglich (entscheidbar). Wegen ihrer Komplexität ist die Bestimmung für die Praxis jedoch zu aufwändig, um damit die *Modellprüfung* zu beschleunigen. Daher wurden effiziente nicht exakte Verfahren wie interprozedurale Konstanten-Faltung, das Computeralgebrasystem Maxima und der SMT-Löser YICES genutzt.

Wie Experimente gezeigt haben, können so *Ausdrücke* mit erträglichem Aufwand gewinnbringend für die *Modellprüfung* eingesetzt werden.
Der Einsatz der aufwändigeren *Wertebereichsanalyse* im Vergleich mit gewöhnlicher Konstanten-Faltung- und -Propagation erscheint zunächst unverhältnismäßig aufwändig. Da die *Wertebereichsanalyse* aber auch für andere Modellreduktionstechniken genutzt wird, können die berechneten abstrakten Informationen (approximierte *Wertebereiche*) auch direkt bereits an dieser Stelle verwendet und später wiederverwendet werden. Auf diese Weise wird die Konstanten-Faltung deutlich *präziser*, wie das Beispiel aus *Quellcode* 4.2 auf Seite 50 zeigte.
Das Lösen aufwändiger Erfüllbarkeitsprobleme zum Vereinfachung von *Ausdrücken* macht im Rahmen von *Programmanalysen* i.d.R. wenig Sinn - für *Modellprüfung* allerdings schon. Denn insbesondere für boolesche *Ausdrücke*, wie die Experimente später zeigen werden können damit an bedingten Verzweigungen nicht realisierbare Pfade identifiziert werden.

4.3.1.7. Zusammenfassung

Rückblick In diesem Abschnitt wurde verdeutlicht, dass komplizierte *Ausdrücke* in einer *SPDS*-Bechreibung einerseits zu einer aufwändigeren *Modellprüfung* führen und andererseits die Qualität von *Modellanalysen* negativ beeinträchtigen. An einem Beispiel-*SPDS* wurde gezeigt, was komplizierte vereinfachbare Ausdrücke sind und wie gewöhnliche Konstanten-Propagation und -Faltung (Techniken aus dem Übersetzerbau) dabei versagen können. Nach Klärung der *Optimal*ität von *Ausdrücken* und deren Komplexität wurden verschiedene Heuristiken vorgestellt und deren Eigenschaften untersucht, um bessere *Ausdrücke* zu finden. Zum Einsatz kamen dabei einerseits klassische Techniken aus der Computeralgebra mittels des Computeralgebrasystems Maxima. Des Weiteren wurde eine Äquivalenzanalyse von *Ausdrücken* für den SMT-Löser YICES erläutert. Und anderseits wurde auch eine Form der Konstantenfaltung mittels *konservativer Wertebereichsanalyse* entwickelt. Hinsichtlich dieser Techniken für *SPDS* wurden einsetzbare *Programmanalysen* (Kontroll- und Datenflussanalysen) untersucht, konstruiert und zur Realisierung verwendet. Der Aufwand der vorgestellten Modellanalysen wird dabei maßgeblich durch die letztlich implementierte *Wertebereichsanalyse* und die *Äquivalenzanalyse* bestimmt und kann je nach verwendetem Fixpunktverfahren verschieden sein. Schließlich wurde die Ausdrucksvereinfachung T_{Expr} erläutert und an dem Beispiel-*SPDS* verdeutlicht. Danach wurden wichtige Eigenschaften wie Entscheidbarkeit, Komplexität und der Einfluss auf den *Konfigurationenraum* untersucht sowie die Korrektheit gezeigt. So ist T_{Expr} entscheidbar und bereits dann optimal, wenn die Modellanalyse optimale Ausdrücke bestimmt. Dabei ist der Aufwand für T_{Expr} linear in der Modellgröße und verändert den Konfigurationenraum nicht. Letzteres ist der Grund dafür, dass auch sämtliche temporalen Formeln ihren Wahrheitsgehalt bei behalten (temporale Invarianz).

Fazit Durch die vorgestellte Ausdrucksvereinfachung T_{Expr} können *Ausdrücke* nun auch in *SPDS* vereinfacht werden. Vereinfachte *Ausdrücke* können sich dadurch nicht nur positiv auf die *Modellprüfung* auswirken, sondern auch auf andere *Modellanalysen* wie jene aus den weiteren Abschnitten (abhängig von der Implementierung). Denn *Modellanalyse*ergebnisse sind dabei natürlicher Weise potentiell um so *präziser*, je einfacher auftretende *Ausdrücke* sind, da in dieser Arbeit eingesetzte Modellanalysen oft lediglich eine konservative Approximation bestimmen. Die vorgestellte Form der Konstantenfaltung mittels *konservativer Wertebereichsanalyse* wurde in der untersuchten Literatur so bisher noch nicht betrachtet und ermöglicht (wie gezeigt) eine über gewöhnliche Konstanten-Faltung und -Propagation hinaus gehende Erkennung von Konstanten. Im Gegensatz zu gewöhnlicher Konstanten-Propagation und Konstanten-Faltung können mit der vorgestellten Technik Terme als konstant identifiziert werden, auch wenn ihre Argumente nicht konstant sind (siehe Quellcode 4.2 auf Seite 50). Der dazu verwendete Ansatz zur Bestimmung von Wertbelegungen (Wertebereichsanalyse) unterliegt im Gegensatz zu anderen vergleichbaren Arbeiten (z.B. Intervallanalysen) weder Monotonie-Einschränkungen wie jener aus [234], noch unterliegt er Einschränkungen auf eine geringe Anzahl spezieller Operationen. Bodik, Gupta und Sarkar berücksichtigen z.B. nur die *Zuweisung* einer Konstanten und die Addition einer Konstanten [198]. Die in diesem Abschnitt entwickelte Technik unterliegt solchen Einschränkungen nicht und interpretiert Operanden oder

Operationen wie Multiplikation, Division und Restbestimmung (mod, %) vollständig. Diese Einschränkungen werden bei anderen Ansätzen u.a. getroffen, damit sie Konvergenz ihrer Verfahren garantieren können. Die in dieser Arbeit betrachteten *SPDS* verfügen allerdings über beschränkte ganze Zahlen, womit Konvergenz auch ohne Aufweitung (Widening) bzw. Einengung (Narrowing) möglich ist [179, 180]. Im Gegensatz zu Ungleichungssystemen (z.B. Polyeder[25] [47]) ermöglicht dieser Ansatz analog zu Delzanno [228] die Charakterisierung des gesamten Wertebereichs (mehrere Intervalle je Variable gleichzeitig in Form von Intervallmengen). Damit ist der entwickelte Ansatz anderen überlegen, welche nur eine untere und eine obere Schranke liefern (z.B. [194]). Auch die Nutzung von Computeralgebrasystemen ist weder im Übersetzerbau, noch bei der Modellprüfung üblich. Keine der untersuchten Arbeiten setzt zum Zweck der Ausdrucksvereinfachung in diesen beiden Gebieten Computeralgebrasysteme ein. Auch der Einsatz eines SMT-Lösers an dieser Stelle ist ebenso originell. Statt nur Ausdrücke einer bestimmten Form (wie z.B. bei Ungleichungssystemen mit Polyedern) können damit nun universell in *SPDS* sämtliche Ausdrücke analysiert und ggf. vereinfacht werden. Die untersuchten Arbeiten treffen keine Aussagen zur Existenz und Komplexität einer optimalen Ausdrucksvereinfachung in *SPDS* (nicht berechenbar in turingmächtigen Programmiersprachen). Aussagen darüber wurden in dieser Arbeit getroffen.

4.3.2. Modellbeschneidung

In diesem Abschnitt werden Slicing ähnliche *SPDS-Modelltransformations*techniken entwickelt, um *SPDS* zu „beschneiden“, d.h. auf den wesentlichen Kern zu reduzieren. Hier wird *Slicing* in der Form verwendet [152, 239], dass zu einem vorgegebenen *Programmpunkt* sämtliche *Anweisungen* identifiziert werden, welche einen Einfluss auf diesen Ausüben bzw. auf welche *Anweisungen* dieser *Programmpunkt* potentiell Einfluss hat. Es seien dazu die zu prüfende Formel ϕ bzw. die auf *Erreichbarkeit* zu prüfenden *SPDS*-Marken $L0 = \{l0, l0', ...\}$ a priori bekannt.

4.3.2.1. Motivation

Mit Slicing und Slicing ähnlichen Ansätzen für *Programmiersprachen* lassen sich in Programmtexten nie ausgeführte *Programmteile* (z.B. *Anweisungen* oder nicht benötigte Variablen) eines Programms identifizieren [152]. Dabei ist ein entsprechendes Slicing-Kriterium gegeben (z.B. ein oder mehrere *Programmpunkte*) und gesucht werden dann Anweisungen, welche einen Einfluss auf das Kriterium haben bzw. auf welche *Anweisungen* das Kriterium potentiell Einfluss hat. Durch Übertragung derartiger Techniken auf *SPDS* können in *SPDS*-Modelle überflüssige Teile der Modellbeschreibung identifiziert werden. Slicing angewendet auf *SPDS*-Modelle ermöglicht dann eine Reduktion der Modellgröße durch Elimination der entsprechenden Teile der Modellbeschreibung (SPDS-*Beschneidung*). Kleinere Modelle erlauben dann eine effizientere *Modellprüfung* oder ermöglichen diese erst. Es ist auch möglich, das *Modellprüfung*sproblem nur anhand eines solchen Slicings zu lösen, so dass sich die Anwendung eines *Modellprüfer*s erübrigt.

Problembewusstsein Gerade im Hinblick auf automatisch gewonnene bzw. generierte Modelle aus höheren *Programmiersprachen* werden i.d.R. viele niemals benötigte *Modellbestandteile* modelliert. Dies passiert z.B. dann, wenn bei Methodenaufrufen an eine Bibliotheksfunktion die gesamte Bibliothek mit vielen nie aufgerufenen Prozeduren im Modell modelliert werden. Dadurch wird ein *SPDS*-Modell unnötig groß und eine *Modellprüfung* erschwert. Andererseits profitieren Modellreduktionstechniken von Synergieeffekten. Denn durch Anwendung anderer *Modelltransformationen* (siehe nachfolgende Kapitel) können Teile der Modellbeschreibung erst überflüssig werden. Dies kann mit Slicing erkannt und genutzt werden, um die Modelle und damit auch die *Modellprüfung* zu beschneiden, d.h. zu verbessern. Dabei ist Slicing im Vergleich zur *Modellprüfung* sehr viel effizienter aber i.d.R. auch sehr viel *unpräziser* und erkennt darum meist nicht alle unwichtigen Teile der Modellbeschreibung.

Beispiel 4.3.3 (Beispiel)
Als Beispiel zur Demonstration diene Quelltext 4.8, eine Erweiterung des Beispiels aus Quellco-

[25] engl. polyhedra

Quellcode 4.8: Erweitertes Beispiel 3.1.

```
int a(8)[100];
init q;

void m() {                    void b(int i(8)) {
    int i(8);                       b0: i = i-1, a[i]=choose(i/2,i*2);
m0: i=99;                           b1: if (i>0) goto b2;
m1: b(i);                           b3: return;
m2: heapify(i);                     b2: b(i); goto b3;
m3: return;  }                      }

void heapify(int i(8)) {              void q() {
     int p(8);                            q0: b(99);
l0:  if (i>0) heapify(i-1);               q1: heapify(99);
l1:  p = (i-1)/2;                         q2: b(99);
l2:  if (a[p] >= a[i]) return;            }
l3:  a[p] = a[i], a[i] = a[p];
l4:  i = p;
l5:  if (i>0) goto l1; }
```

de 3.1 auf Seite 24. *Die* SPDS-*Variable* i *aus Beispielcode 3.1 auf Seite 24 wurde mit dem Typ* $bits(i) = 8$ *modelliert. In der Prozedur* heapify *wird die Haldenstruktur berechnet, welche z.B. für eine Haldensortierung (Heapsort) nötig ist. Die Haldenstruktur wird in der* Reihung a *mit* $len(a) = M$ *Elementen jeweils des Typs* $bits(a) = 8$ *hergestellt. Der Parameter* i *gibt an, bis zu welchem Index die Haldenstruktur hergestellt wird. Dabei ist* $p = (i-1)/2$ *der Index des Elternknotens vom Kind mit Index* i*. Für die Haldenstruktur werden die Elemente derart sukzessive getauscht, dass für alle Kinder-Knoten* i *und Elternknoten mit Index* $p = (i-1)/2$ *gilt:* $a[p] \geq a[i]$*. In Prozedur* b *wird die* Reihung a *mit* $a[i] = choose(i/2, i*2)$ *belegt. Prozedur* q *ruft dann Prozedur* b *auf, um die* Reihung *entsprechend für alle 100 Elemente der Reihung zu initialisieren. Dann ruft* q *die Prozedur heapify auf, um eine Haldenstruktur in der* Reihung *zu erzeugen. Die Prozedur* q *legt die* Startkonfigurationen $I = \{(env_a, (q0, env_\emptyset)) \mid env_a, env_\emptyset \in ENV\}$ *des* SPDS *fest.* $env_\emptyset$ *ist dabei die leere* Variablenbelegung*, da es in der Prozedur* q *keine lokalen* SPDS-*Variablen gibt. Bei genauerer Betrachtung der* Konfigurationenübergänge *des Beispiels fällt auf, dass die Prozedur* m *nicht aufgerufen wird. Die Prozedurbeschreibung von* m *ist daher überflüssig. Aus der Hauptprozedur* q *werden lediglich die Prozeduren* b *und heapify aufgerufen.* b *ruft sich dann rekursiv selbst auf. Im Anschluss danach ruft* q *noch die Prozedur heapify auf. Sei angenommen, dass im Rahmen einer* Modellprüfung *für dieses Beispiel die* Erreichbarkeit *der Marke* $L0 = \{l0\}$ *zu prüfen ist. Dies kann z.B. dann der Fall sein, wenn ein Modellprüfender Gewissheit darüber haben möchte, ob die Prozedur heapfy im Modell benötigt wird. Wie durch eine Rückwärtsanalyse der* Erreichbarkeit *im Kontrollfluss im Folgenden erklärt wird, sind dann auch die Marken* $l1, l2, l3, l4$ *und* $l5$ *für diese* Erreichbarkeitsprüfung *überflüssig, wie letztlich auch die Marke* $q2$ *sowie den Parameter* i *und die lokale Variable* p *von Prozedur heapify. Wie dies erkannt und eliminiert werden kann, wird nun beschrieben.*

Lösungsidee Um nicht benötigte Teile der Modellbeschreibung zu identifizieren, wird der Kontrollfluss und der Datenfluss *abstrakt* mittels statischem Slicing untersucht. Als mögliche unnötig erkennbare Teile dienen dabei SPDS-*Anweisungen*, *SPDS*-Marken, *SPDS*-Variablen und *SPDS*-Prozeduren. Wenn diese Teile durch Slicing als (definitiv) unwichtig erkannt sind, können sie aus dem Modell entfernt, d.h. heraus *geschnitten* werden.

4.3.2.2. Notwendige abstrakte Informationen

Sei $used_l(v) := used_l^{read}(v) \vee used_l^{write}(v)$ eine schreibende[26] oder lesende Verwendung[27] von v, d.h.

$$used_l(v) := \begin{cases} true & \text{falls } v \text{ schreibend oder lesend verwendet wird an Marke } l \\ false & \text{sonst.} \end{cases}$$

Die abstrakte Information *Slice* gibt dann Aufschluss über die *Erreichbarkeit* oder potentielle Beeinflussung des Slicing-Kriteriums für Teile der Modellbeschreibung.

Definition 4.3.14 (*Slice* Λ)
Sei ein SPDS S *und der zugehörige interprozedurale Kontrollflussgraph* $G = (V, E)$ *gegeben. Ein Slice* $\Lambda \subseteq Stats(S) \cup Marken(S) \cup Vars(S) \cup Prz(S)$ *ist eine Menge von Teilen der Modellbeschreibung*[28] *mit den Eigenschaften:*

$$\forall p \in Prz(S) : l \in \Lambda \cap Marken(p) \quad \Rightarrow \quad p \in \Lambda \tag{4.3}$$

$$\forall \text{„}l : s\text{“} \in Stats(S) : l, l' \in \Lambda \cap Marken(S) \wedge l \xrightarrow{s}_E l' \quad \Rightarrow \quad \text{„}l : s\text{“} \in \Lambda \tag{4.4}$$

$$\forall v \in Vars(S) : l \in \Lambda \cap Marken(S) \wedge used_l(v) \quad \Rightarrow \quad v \in \Lambda \tag{4.5}$$

Die Menge aller Slices sei mit Slices(S) *bezeichnet. Die abstrakte Information für Teile der Modellbeschreibung* x *sei definiert als zugehörige charakteristische Funktion* $\lambda : Stats(S) \cup Marken(S) \cup Vars(S) \cup Prz(S) \to \{true, false\}$ *mit*

$$\lambda(x) := \begin{cases} true & falls\ x \in \Lambda \\ false & sonst. \end{cases}$$

Gleiche SPDS-*Anweisungen* s an unterschiedlichen *SPDS*-Marken werden wegen ihrer unterschiedlichen Marken unterschieden, denn eine SPDS-*Anweisung* an der Marke „$l_1 : s$“ $\notin \Lambda$ könnte unnötig sein, die andere an Marke „$l_2 : s$“ $\in \Lambda$ aber nicht. Für SPDS-*Anweisungen*, *SPDS*-Marken, *SPDS*-Variablen und *SPDS*-Prozeduren wird die Zugehörigkeit zu einem Slice (Ergebnis von Slicing-Berechnungen) bestimmt.
Ein Slice kann sowohl vorwärts gerichtet sein und bei den *Anfangskonfigurationen* beginnen oder rückwärts gerichtet sein und bei der Fehlermarke (Menge an Konfigurationen) bzw. bei den durch ϕ beeinflussenden *Konfigurationen* beginnen.

Definition 4.3.15 (*Vorwärts-Slice*, *Rückwärts-Slice*)
Ein Vorwärts-Slice $\Lambda_+ \in Slices(S)$ *wird für gegebene* Anfangskonfigurationen I *derart bestimmt, dass gilt:*

$$\forall l \in Marken(S) : I \rightsquigarrow l \Rightarrow l \in \Lambda_+ \tag{4.6}$$

Ein Rückwärts-Slice $\Lambda_- \in Slices(S)$ *wird für a priori gegebene Marken* $L0$ *derart bestimmt, dass gilt:*

$$\forall l \in Marken(S) : l \rightsquigarrow L0 \Rightarrow l \in \Lambda_- \tag{4.7}$$

Es bezeichne $x \in \phi$ *das syntaktische Vorkommen der Marke* x*, der Prozedur* x *oder einer Variable* x *in der* temporalen Formel ϕ *oder in* ϕ *enthaltenen Annotationsausdrücken (AExpr). Wie bereits bei der Semantikdefinition für den* Prozeduraufruf *sei* l_p *die Anfangsmarke der Prozedur* p *(siehe Abschnitt 3.2.2 ab Seite 28).*
Dann wird ein Rückwärts-Slice $\Lambda_- \in Slices(S)$ *für eine a priori gegebene* temporale Formel ϕ *auf eine Menge* $L0 \subseteq Marken(S)$ *von Marken derart zurückgeführt, dass für* $L0 = L0_\phi$ *gilt:*

$$\forall v \in Vars(S), \text{„}l_1 : v = e;\ l_2 : \ldots\text{“} \in S \ : \ v \in \phi \Rightarrow l_1, l_2 \in L0_\phi \tag{4.8}$$

$$\forall p \in Prz(S), v \in loc_p \ : \ v \in \phi \Rightarrow l_p \in L0_\phi \tag{4.9}$$

$$\forall v \in Vgbl(S) \ : \ v \in \phi \Rightarrow Marken(I) \subseteq L0_\phi \tag{4.10}$$

$$\forall l \in Marken(S) \ : \ l \in \phi \Rightarrow Marken(S) \subseteq L0_\phi \tag{4.11}$$

$$\forall p \in Prz(S) \ : \ p \in \phi \Rightarrow Marken(S) \subseteq L0_\phi \tag{4.12}$$

[26] Zuweisung
[27] Vorkommen in einem Ausdruck
[28] Dies sind SPDS-*Anweisungen* $l : s \in Stats(S)$, *SPDS*-Marken $l \in Marken(S)$, *SPDS*-Variablen $v \in Vars(S)$ und *SPDS*-Prozeduren $p \in Prz(S)$.

Eigenschaften Für Regel (4.7) ist auch der leere Pfad $\rightsquigarrow$ möglich, so dass $L0 \subseteq \Lambda_-$. Ist $l_1 \in \Lambda_-$ für ein $l_1 \in Marken(S)$ gemäß Regel (4.8) und (4.7), so ist mit Regel (4.5) auch $\forall v \in Vars(S) : used_{l_1}(v) \Rightarrow v \in \Lambda_-$. Die Regeln (4.9) sowie (4.10) berücksichtigen Initialbelegungen von Variablen. Insbesondere für Parametervariablen ist die wichtig, da diese im Gegensatz zu globalen und lokalen Variablen nicht willkürlich belegt sind. Gemäß Regeln (4.11) und (4.12) umfasst jeder Rückwärts-Slice sämtliche Marken, wenn eine temporale Formel Aussagen über die *Erreichbarkeit* von Marken oder Prozeduren macht. Denn z.B. gilt die temporale LTL-Formel $\phi = F\ l2$ nicht an der Marke $l1$ wegen der möglichen Verzweigung zur Marke $l3$.

```
l1:  if (...) goto l3;
l2:  skip false;
l3:  skip false;
```

Würde ein Slice mit $l2 \in \Lambda_-$ aber $l3 \notin \Lambda_-$ die Beschneidung der Marke $l3$ (Streichen der Zeile) erlauben, so würde ϕ an der Marke $l1$ wahr werden, da dann jeder Pfad zur Marke $l2$ führt. Der *Wahrheitsgehalt* hätte sich unerwünschterweise geändert. Ist schließlich $p \in \phi$ für ein $p \in Prz(S)$ so ist gemäß Regel (4.12) $Marken(p) \subseteq L0$ und mit (4.7) sowie (4.3) ist dann auch $p \in \Lambda_-$. Es sei bemerkt, dass eine Variable ihren Wert lediglich durch eine *SPDS-Zuweisung* ändern kann. Für solche Zuweisungen gelte genau dann im Folgenden $changed_{l_1}(v) = true$[29].

Lemma 4.3.11 (*Slice-Schnitt*)
Seien $\Lambda_-, \Lambda_+ \in Slices(S)$. Dann ist auch $\Lambda_+ \cap \Lambda_- \in Slices(S)$ ein Slice.

Für einen korrekten *Modellprüfung*sprozess ist es wichtig, dass ein *konservativer* Slice berechnet wird, d.h. $\forall I \rightsquigarrow l$ ist $l \in \Lambda_+$, aber es kann Marken $l \in Marken(S)$ geben, für die gilt $l \in \Lambda_+$ und $I \not\rightsquigarrow l$. Bei Modellreduktion mittels nicht-*konservativer* Slices könnten reale Läufe aus dem Modell entfernt werden und es könnte so zu *False Positives* während des *Modellprüfung*sprozesses kommen[30]. Bei Modellreduktion mittels *konservativer* Slices kann es daher unnötige Teile der Modellbeschreibung geben, welche durch den Slice nicht als unnötig erkannt wurden. Identifiziert ein Slicing andererseits stets genau die unnötigen Teile der Modellbeschreibung, so heißt das Slicing *exakt* (siehe Definition 3.3.2 auf Seite 33). Weiser hat in [144] gezeigt, dass minimale Slices in gängigen Hochsprachen nicht berechenbar sind (Halteproblem). Ein exaktes Slicing für *SPDS* auf Ebene von *SPDS*-Marken entspricht aber der berechenbaren *Modellprüfung* von *Erreichbarkeit* in *SPDS*. Für exaktes Slicing erübrigt sich die Modellprüfung für Erreichbarkeit. Daher muss ein Slicing um so aufwändiger sein, je *präzise*r es ist. Hieran wird die Bedeutung eines *präzise*n Slicings deutlich, um so viel wie möglich unwichtige Teile der Modellbeschreibung vor der aufwändigen *Modellprüfung* zu erkennen und zu entfernen.

4.3.2.3. Einsetzbare Programmanalysen

Mit Slicing und Slicing ähnlichen Ansätzen lassen sich in *Programmtexten* von gängigen Hoch- und *Programmiersprachen* nie ausgeführte bzw. nicht benötigte *Programmteile* (insb. Anweisungen) eines *Programms* identifizieren. Temporale Formeln wie in dieser Arbeit werden dabei i.d.R. nicht berücksichtigt. In der Literatur finden sich für Hochsprachen viele verschiedene Slicing ähnliche Ansätze. Dazu zählen *Slicing* [197, 152], *Cone of Influence Analysis* [113, 105, 1], *Localization* [1], *Live Variables Analysis* [86], *Influence Analysis* [182] und andere *Lebendigkeits-Analysen* (engl. live range + dead code analysis) [143, 39, 60]. Die Unterscheidung ist wie folgt.

Cone of Influence Analysen bestimmen (Fixpunktberechnung direkter und indirekter Datenabhängigkeiten [105]) Variablen an *Programmpunkten* mit möglichem Einfluss auf einen Programmpunkt [1].

[29] Es gilt: $changed_l(v) \Rightarrow used_l(v)$ (schreibende *Verwendung*). Die Umkehrung gilt nicht (lesende V.).
[30] Der *Modellprüfer* findet keinen Fehler, der aber realisiert werden kann durch einen Lauf, welcher wegen nicht-*konservativen* Slicing eliminiert wurde.

Slicing bestimmt *Programmpunkte* mit möglichem Einfluß auf vorgegebenes Slicing-Kriterium (i.d.R. ein *Programm*-punkt). Nach [152] ist diese Form gut geeignet zur *Zustandsraum*reduktion und wird in diesem Abschnitt auf *SPDS*-Konfigurationen und *temporale Formel*n erweitert.

Localization dient zur (iterativen) Aufteilung des Problems wegen seiner Größe in kleinere Komponenten [1] und identifiziert dabei lokale Abhängigkeiten.

Live Variables Analyse bestimmt definitiv oder potentiell lebendige Variablen (solche, deren Wert noch benötigt wird) an jedem *Programmpunkt* [86]. Wegen nötiger *Konservativität* können Modelle nicht auf definitiv lebendige Variablen reduziert werden. Dazu sind potentiell lebendige Variablen zu verwenden.

Influence Analysen bestimmen zu jedem *Programmpunkt* eine Menge signifikanter Variablen [182], welche Einfluss auf bestimme Modelleigenschaften haben. Die Influence Analyse ist eine Erweiterung der Live Variables Analyse. Sie ist mächtiger als Cone of Influence Analysen, welche kein „Unsichtbarwerden“ von Variablen für Nachfolgezustände ermöglicht [182].

Lebendigkeitsanalysen bzw. Dead-Code-Analysen erkennen *erreichbaren* bzw. nichterreichbaren Code oder verwendete bzw. nicht verwendete Variablen oder essenzielle bzw. unnötige *Zuweisung*en. Eine Dead Code Analyse [39] ist vergleichbar zu einem einfachen *SPDS*-Vorwärts-Slice für Marken, erkennt aber innerhalb des Slices evtl. unnötige *Zuweisung*en etc [143].

4.3.2.4. Modellanalysebeispiel

Im Folgenden wird ein interprozedurales Kontrollfluss-basiertes Slicing für *SPDS* entworfen. Kontrollflüsse liefern eine einfache Approximation realer Läufe. Damit können viele Teile der Modellbeschreibung als unwichtig erkannt werden, wie die Experimente später zeigen.

Idee Es wird ein interprozeduraler Vorwärts-Slice beginnend bei den *symbolisch*en *Startkonfigurationen* I eines *SPDS* S berechnet, welcher alle von I aus über den Kontrollfluss *erreichbaren* *SPDS*-Kontrollpunkte (Marken) des Modells enthält sowie die darin verwendeten Variablen, Prozeduren und Anweisungen. Beginnend bei den auf *Erreichbarkeit* zu prüfenden Marken bzw. den durch ϕ beeinflussenden Teilen der Modellbeschreibung wird anschließend ein interprozeduraler Rückwärts-Slice berechnet, um potentiell relevante *Modellbestandteile* für die *Modellprüfung* zu identifizieren. Nur jene Teile der Modellbeschreibung sind für die *Modellprüfung* relevant, welche sowohl im Vorwärts- als auch im Rückwärts-Slice vorkommen.

Konkretisierung

Definition 4.3.16 (*Beispiel-Vorwärts- und -Rückwärts-Slice* Λ_+ und Λ_-)
Sei $K = (V, E)$ mit $V := Marken(S)$ der interprozedurale Kontrollflussgraph für das SPDS *S mit der Startmarke l_{main} der Hauptprozedur main. Dann ist*

$$L_+ := \{l_{main}\} \cup \{l_k \in V \mid \exists (l_{main}, l_1), (l_1, l_2), \ldots, (l_{k-1}, l_k) \in E\}$$

die Menge der über den interprozeduralen Kontrollfluss erreichbaren *Marken. Durch Berücksichtigung der Regeln (4.3 auf Seite 70), (4.4 auf Seite 70) und (4.5 auf Seite 70) aus Definition 4.3.14 auf Seite 70, wird daraus der Vorwärts-Slice $\Lambda_+ := \Lambda(L_+)$, wobei*

$$\begin{aligned} \Lambda(L) \quad := \quad & \{p \in Prz(S) \mid \exists l \in L : l \in Marken(p)\} \\ \cup \quad & \{l : s \in Stats(S) \mid \exists l, l' \in L : l \xrightarrow{s}_E l'\} \\ \cup \quad & \{v \in Vars(S) \mid \exists l \in L : used_l(v)\}. \end{aligned}$$

Seien $L0 \subseteq Marken(S)$ *die auf* Erreichbarkeit *zu prüfenden Marken bzw. die durch eine Formel* ϕ *gemäß Definition 4.3.15 auf Seite 70 relevanten Marken. Dann ist*

$$L_- := L0 \cup \{l_k \in V \mid \exists (l_1, l_0), (l_2, l_1), \ldots, (l_k, l_{k-1}) \in E \wedge l_0 \in L0\}$$

die Menge der über den interprozeduralen Kontrollfluss rückwärts erreichbaren *Marken. Hieraus wird analog der Rückwärts-Slice* $\Lambda_- := \Lambda(L_-)$.

Eigenschaften In den *Modellanalyse*beispielen Λ_- bzw Λ_+ wird der Kontrollfluss zur sicheren Approximation realer *Konfigurationenübergänge* $\rightarrow$ verwendet. Die *Modellanalyse*n sind daher *unpräzise*, aber *konservativ*. Die vorgestellten Beispiel-Slicings Λ_+ und Λ_- sind wegen Berücksichtigung des interprozeduralen Kontrollflusses sowohl flusssensitiv als auch interprozedural. Gemäß Betrachtungen aus Abschnitt 4.3.2.2 ab Seite 70 sind exakte Vorwärtsslicings entscheidbar. Exaktes Rückwärtsslicing kann ähnlich auf die *Erreichbarkeitsprüfung* zurückgeführt werden (Berechenbarkeit von Pre^* anstatt $Post^*$). Exaktes Rückwärtsslicing ist daher ebenso entscheidbar. Die vorgestellten *Modellanalyse*beispiele sind kontextinsensitiv, pfadinsensitiv und werden mittels einer Tiefen- oder Breitensuche im Kontrollfluss bestimmt (nicht mit einem Fixpunktalgorithmus).
Die Größe des Kontrollflusses ist dabei linear in der Länge n der *SPDS*-Beschreibung (Quelltext). Zur Bestimmung von $L0$ für eine Formel ϕ ist daher jeweils maximal linearer Aufwand in der *SPDS*-Beschreibung n sowie in der Länge der Formel, d.h. insgesamt $O(n + |\phi|)$ nötig:

- Markiere in ϕ verwendete Marken, Prozeduren und Variaben in $O(|\phi|)$.
- Analysiere *SPDS*-Beschreibung auf markierte Teile in $O(n)$.

L_+ bzw. Λ_+ und L_- bzw. Λ_- können dann durch eine Tiefensuche im interprozeduralen Kontrollflussgraphen in $O(n)$ bestimmt werden. Zur Berechnung der Slices wird damit insgesamt ein Aufwand $O(n + |\phi|)$ benötigt.

Vorwärts- und Rückwärts-Slices des Beispiels 4.3.3 Die Beispielmodellanalyse bestimmt folgende abstrakte Informationen: $L_+ = \{q0, b0, b1, b2,\ b3, q1, l0, l1, l2,\ l3, l4, l5\}$, $L_- = \{l0, m2, b3, b2, b1, b0, m1,\ m0, q1, q0\}$. Dann ist $\Lambda_+ \cap \Lambda_- = \Lambda(L_+ \cap L_-) = \Lambda(\{q0, b0,\ b1, b2, b3, q1, l0\})$. $\Lambda_+ \cap \Lambda_-$ enthält alle *Modellbestandteile* abgesehen von der Prozedur m (inkl. lokale Variable i) und auch nicht die Marken $l1, l2, l3, l4$ und $l5$ sowie deren zugehörige SPDS-*Anweisungen*. Auch die lokale Variable p und die Parametervariable i der Prozedur $heapify$ sind nicht im Slice $\Lambda_+ \cap \Lambda_-$ enthalten.

4.3.2.5. Modellbeschneidung T_{slice}

Sind die abstrakten Informationen in Form eines Vorwärts- und Rückwärts-Slice bestimmt, so kann das Modell mittels *Modelltransformationen* wie folgt beschnitten werden.

Idee Werden im Kontrollfluss aus den *Anfangskonfigurationen* nicht *erreichbare SPDS*-Marken identifiziert (Vorwärts-Slice), werden diese einschließlich zugehöriger SPDS-*Anweisung* aus dem Modell eliminiert. Ebenso werden *SPDS*-Marken l und SPDS-*Anweisungen* $(l : ...)$ entfernt, welche definitiv nicht zu den zu prüfenden *SPDS*-Marken $l0 \in L0$ führen ($l \not\rightsquigarrow L0$) oder keinen Einfluss auf den *Wahrheitsgehalt* der *temporalen Formel* ϕ haben (Rückwärts-Slice). Analoges gilt für Variablen und Prozeduren. Sie werden eliminiert, wenn sie nicht im Vorwärts-Slice bzw. nicht im Rückwärts-Slice liegen.

Konkretisierung Sei ein *SPDS* S sowie ein zugehöriger Vorwärts-Slice Λ_+ und ein Rückwärts-Slice Λ_- gegeben. Dann werden zur Modellbeschneidung T_{slice} nacheinander in der gegebenen Reihenfolge so lange wie möglich die folgenden *Modelltransformationen* angewendet:

(S_1) Eliminiere SPDS-*Anweisungen* gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{p \in Prz(S) \quad l : s \in Body(p) \quad l : s \notin \Lambda_+ \cap \Lambda_-}{l : s; \ \Rrightarrow \ l : \ skip\ false;}$$

(S_2) Eliminiere Prozeduren gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{p \in Prz(S) \quad p \notin \Lambda_+ \cap \Lambda_-}{\begin{array}{cc} Prz & \Rrightarrow \quad Prz \setminus \{p\} \\ PrzDesc & \Rrightarrow \quad PrzDesc \setminus p \end{array}}$$

Dabei bezeichnet $PrzDesc \setminus p$ das Entfernen der Prozedur p aus der Menge der Prozedurbeschreibungen $PrzDesc$ einschließlich zugehöriger Signatur $(p, Param_p, Vlcl_p)$ und des Prozedurrumpfes $[s_1, s_2, \ldots, s_m]$, d.h.

$$PrzDesc \setminus p = PrzDesc \setminus \{((p, Param_p, Vlcl_p), [s_1, s_2, \ldots, s_m])\}.$$

Die Berücksichtigung von *Prozeduraufruf*en durch eine Regel der Form „$l : p(x_1, x_2, \ldots x_n); \Rrightarrow l : skip\ false;$“ ist nicht nötig, da solche SPDS-*Anweisungen* durch die Transformationsregel (S_1) eliminiert werden[31].

(S_3) Eliminiere *SPDS*-Marken gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{p \in Prz(S) \quad l : s \in Body(p) \quad l \notin \Lambda_+ \cap \Lambda_-}{\begin{array}{cc} Body(p) & \Rrightarrow \quad Body(p) \setminus l : s \\ l' : \ if\ (e)\ goto\ l; & \Rrightarrow \quad l' : \ skip; \end{array}}$$

Dabei bezeichne $Body(p) \setminus l_i : s_i = [l_1 : s_1, l_2 : s_2, \ldots, l_m : s_m] \setminus l_i : s_i = [l_1 : s_1, l_2 : s_2, \ldots, l_{i-1} : s_{i-1}, l_{i+1} : s_{i+1}, \ldots, l_m : s_m]$. Die Regel „$l' : \ if\ (e)\ goto\ l; \Rrightarrow l' : \ skip;$“ ist nötig, da ein Slice u.U. nur l und nicht l' als unnötig erkennt, wenn $[\![e]\!]_{env} = false$, für alle realisierbaren *Variablenbelegung*en env an Marke l'.

(S_4) Eliminiere globale Variablen gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{v \in Vgbl \quad v \notin \Lambda_+ \cap \Lambda_-}{Vgbl \ \Rrightarrow \ Vgbl \setminus v}$$

Die Transformationsregeln (S_1), (S_2) und (S_3) garantieren, dass die globale Variable v nirgends im Restmodell *Verwendung* findet, so dass zur Elimination von v keine weitere *Modelltransformation* nötig ist und direkt aus der Typdeklaration gelöscht werden kann.

(S_5) Eliminiere Parameter-Variablen gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{p \in Prz(S) \quad v_i \in Param_p = [v_1, v_2, \ldots, v_m] \quad v_i \notin \Lambda_+ \cap \Lambda_-}{\begin{array}{cc} Param_p & \Rrightarrow \quad Param_p \setminus v_i \\ p(x_1, x_2, \ldots, x_m) & \Rrightarrow \quad p(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots x_m) \end{array}}$$

Beim Eliminieren unwichtiger Parameter müssen natürlich auch die entsprechenden Variablen x_i für den eliminierten Parameter v_i beim *Prozeduraufruf* gelöscht werden. Das Nicht-Vorkommen des Parameters v_i im Restmodell garantieren dann wieder analog die Transformationsregeln (S_1), (S_2) und (S_3).

(S_6) Eliminiere lokale Variablen gemäß Vorwärts- und Rückwärts-Slice:

$$\frac{p \in Prz(S) \quad v \in Vlcl_p \quad v \notin \Lambda_+ \cap \Lambda_-}{Vlcl_p \ \Rrightarrow \ Vlcl_p \setminus v}$$

Auch hier garantiert das Nicht-Vorkommen der lokalen Variable v im Restmodell wieder die Transformationsregeln (S_1), (S_2) und (S_3).

[31] Für jeden Slice Λ und *Prozeduraufruf* p an der Marke l gilt: $l, l_p \in \Lambda \Rightarrow$ „$l : p(x_1, x_2, \ldots x_n)$“ $\in \Lambda$. Ist $p \notin \Lambda$, so ist auch $l_p \notin \Lambda$ und damit „$l : p(x_1, x_2, \ldots x_n)$“ $\notin \Lambda$ und Regel (S_1) findet Anwendung.

Quellcode 4.9: *SPDS*-Beschneidung von Beispiel 4.3.3.

```
int a(8)[100];
init q;

void b(int i(8)) {
b0: i = i-1, a[i]=choose(i/2,i*2);
b1: if (i>0) goto b2;
b3: return;
b2: b(i); goto b3;
}

void heapify() {
l0:  skip false;
}

void q() {
   q0: b(99);
   q1: heapify(); }
```

Es wird erst die Transformationsregel (S_1) solange angewandt, bis es keine Anwendungsmöglichkeit mehr gibt, dann (S_2) etc. Eine Wiederholung der Transformationsregel (S_1) nachdem (S_6) nicht mehr angewandt werden kann, ist nicht nötig, da eliminierbare Teile der Modellbeschreibung gemäß der gegebenen Slices nur gelöscht werden, aber nicht neu hinzukommen. Die Anwendung der Transformationsregeln (S_3) und (S_5) z.B. führen allerdings dazu, dass *Ausdrücke* $e \in Expr$ gelöscht werden. Sind Variablen nur in solchen „wegoptimierten“ *Ausdrücke*n vorhanden, so kann durch **Neuberechnen** der Slicings und wiederholte Anwendung der Modellbeschneidung T_{slice} das Modell weiter verkleinert werden.

Modelloptimierung des Motivationsbeispiels Für $L0 = \{l0\}$ kann der Beispielcode 4.3.3 auf Seite 68 gemäß der durch die Beispiel-Vorwärts- und Rückwärts-Slicings Λ_+ und Λ_- bestimmten abstrakten Informationen beschnitten werden. Das Ergebnis dieser Beschneidung zeigt *Quellcode* 4.9.
Verwendet ϕ die lokalen Variablen p und i der Prozedur *heapify* des Beispiels nicht $(p, i \notin \phi)$ und ist $p, i \notin \Lambda_+ \cap \Lambda_-$, so können die Variablen p und i auch für die temporale *Modellprüfung* eliminiert werden, da der *Wahrheitsgehalt* von ϕ nicht von diesen *Variablenbelegung*en ab hängt.

Wohldefiniertheit der Modellbeschneidung T_{slice} Weil das Modell und daher auch die Slicings Λ_- und Λ_+ endlich sind, kann es für jede der Transformationsregeln nur endlich viele Anwendungen geben, so dass die Modellreduktion wohldefiniert ist. Aber auch wenn die Slices Λ_- und Λ_+ neu berechnet und die Modellbeschneidung T_{slice} wiederholt eingesetzt werden, terminiert die wiederholte Modellbeschneidung wegen Monotonie in der Modellgröße (Wegschneiden aus einer endlichen Menge).

Korrektheit (Invarianz Temporaler Formeln und Erreichbarkeit) Für die Korrektheit des vorgestellten Verfahrens genügt es, die Invarianz bezüglich gegebener temporaler Formeln zu zeigen.

Satz 4.3.8
Die SPDS*-Beschneidung T_{slice} ist CTL*-invariant für gegebenes $\phi \in CTL^*$.*

Der Beweis findet sich im Anhang B ab Seite 145.

Satz 4.3.9 (Korrektheit von T_{slice})
Die Ausdrucksvereinfachung T_{Expr} ist α-invariant für beliebiges $\alpha \in \{ACTL, ACTL\text{-}X, ACTL^*, ACTL^*\text{-}X, CTL, CTL\text{-}X, CTL^*, CTL^*\text{-}X, LTL, LTL\text{-}X\}$ und auch Erreichbarkeits-invariant *für eine Menge gegebener Marken $L0 \in Marken(S)$ eines* SPDS *S.*

4.3.2.6. Eigenschaften

Im Folgendem sollen einige Eigenschaften der Modellbeschneidung T_{slice} erörtert werden.

Entscheidbarkeit einer optimalen Modellbeschneidung Auch hier entscheidet die Güte der *Modellanalyse* über das Modellreduktionsergebnis. Wie beschrieben, löst ein *optimale*s (exaktes) Slicing bereits die *Modellprüfung* für die *Erreichbarkeit*. Die *Optimal*ität der Modellbeschneidung ist daher in Abhängigkeit der Optimalität des Slicings wie folgt definiert.

Definition 4.3.17 (*Optimale Modellbeschneidung*)
Eine Modellbeschneidung T für einen Slice Λ heißt präzise*, wenn T das Modell auf genau die Teile der Modellbeschreibung des Slice Λ beschneidet. Ist Λ exakt, so heißt T* optimal*.*

Satz 4.3.10 (Präzision von T_{slice})
T_{slice} ist eine präzise *Modellbeschneidung für den Slice $\Lambda_+ \cap \Lambda_-$.*

Beweis Die *Modelltransformationsregeln* beschneiden das *SPDS* auf genau die minimale Menge an Teilen der Modellbeschreibung, welche durch den Slice $\Lambda_+ \cap \Lambda_-$ definiert ist.

□

Satz 4.3.11 (Entscheidbarkeit einer *optimale*n Modellbeschneidung)
Die optimale *Modellbeschneidung ist entscheidbar.*

Beweis Wegen der Existenz von T_{slice} als *optimale* Modellbeschneidung und der Möglichkeit auch Slicing exakt durchführen zu können, ist die *optimale* Modellbeschneidung entscheidbar.

□

Fehlalarme Im Gegensatz zu anderen Reduktionstechniken bzw. *Abstraktion*stechniken entstehen durch die Modellbeschneidung T_{slice} keine neuen **False Negatives**, da alle essenziellen Läufe bezüglich $L0$ erhalten bleiben und durch die Beschneidung keine neuen Läufe hinzu kommen.

Konfigurationenraumveränderung Der durch das *SPDS* S beschriebene *Konfigurationenraum* wird auf den notwendigen *Verwendungsteil* der Slicings beschnitten und dadurch verkleinert. Der *erreichbare Konfigurationenraum* $Post^*(\gamma[\![S]\!]_\phi)$ verändert sich in dem Sinne, dass er auf den Teil beschnitten wird, welche noch potentiell einen Einfluss auf das *Modellprüfung*sergebnis hat. Insbesondere werden nicht benötigte Variablen aus dem Modell entfernt, was den *Konfigurationenraum* für *Konfigurationenübergänge* substantiell um Größenordnungen ändert. Die Beschneidung T_{slice} führt dazu, dass ein kleineres Kelleralphabet benötigt wird für das zu Grunde liegende *PDS*, was einen *Modellprüfer* weiter beschleunigt.
Die einzelnen *Modelltransformationen* verändern den *Konfigurationenraum* wie folgt:

(S_1) Läufe werden an der Marke l abgeschnitten (verkürzt). Dadurch wird der *Konfigurationenraum* eingeschränkt.

(S_2) Die Elimination einer Prozedur p verkleinert das Kelleralphabet um $|Marken(p)|$ Marken (Konfigurationenübergänge) und $\sum_{v \in loc_p} bits(v)$ Variablenbits[32]. Der *Konfigurationenraum* (Marken mit lokalen *Variablenbelegung*en im Keller) für *PDS*-Konfigurationen, hervorgehend aus *SPDS-Konfigurationen* der Form $(env_{Vgbl}, (l_1, env_1) \ldots (l_n, env_n))$, wird daher um k *Konfigurationen* kleiner, wobei

$$k := \left((|Marken(p)| + 1) \cdot \sum_{v \in loc_p} range(v) \right)^n .$$

[32] $loc_p = Param_p \cup Vlcl_p$ (siehe Abschnitt 3.2.2 ab Seite 28)

(S_3) Durch die Regel $Body(p) \Rrightarrow Body(p) \setminus l : s$ wird ein *symbolischer Konfigurationenübergang* und eine Marke eliminiert. Der *Konfigurationenraum* ändert sich hierdurch kaum. Die Regel $l' : \ if\ (e)\ goto\ l; \Rrightarrow l' : \ skip$; schneidet Läufe ab. Dadurch wird der *erreichbare Konfigurationenraum* $Post^*(\gamma[\![S]\!]_\phi)$ kleiner, wenn es eine realisierbare *Variablenbelegung* $env \in ENV$ gibt, so dass $[\![e]\!]_{env} = 1$.

($S_{4,6}$) Analog zu (S_2) wird durch die Elimination von v der *Konfigurationenraum* kleiner.

(S_5) Analog zu (S_2) wird durch die Elimination des Parameters v_i der *Konfigurationenraum* kleiner. Das Entfernen der zugehörigen Variablen x_i ändert den *Konfigurationenraum* nicht. Lediglich die *Auswertung* de Variable x_i entfällt, was den *Modellprüfer* etwas beschleunigt.

Komplexität Zu einem gegebenem *SPDS* werden zunächst die Slices Λ_+ und Λ_- bestimmt. Dies benötigt nach Betrachtungen der Eigenschaften der *Modellanalyse*beispiele Λ_+ und Λ_- einen Aufwand von $O(n+|\phi|)$, wobei n die Länge der *SPDS*-Beschreibung und $|\phi|$ die Länge einer *temporalen Formel* ϕ ist (üblicherweise ist $n >> |\phi|$).
Danach wird die Modellreduktion T_{slice} eingesetzt, um das *SPDS* zu beschneiden. Dazu wird erst die Regel (S_1) solange angewandt, bis es keine Anwendungsmöglichkeit mehr gibt, dann (S_2) etc. Jede der 6 Transformationsregeln benötigt einen Aufwand von $O(n)$, da jede Regel maximal so oft angewand werden kann, wie es noch Teile der Modellbeschreibung gibt und jede Regelanwendung einen solchen Teil eliminiert (Monotonie).
Der insgesamte Aufwand zur Modellbeschneidung mittels T_{slice} und der *Modellanalyse*beispielslicings Λ_+ und Λ_- beträgt somit $O(n + |\phi|)$. Je nach Slicing kann der Prozess wiederholt werden. Eine solche iterative Wiederholung kann abhängig vom Slicing und des Modells bis zu n mal erfolgen, bis keine weitere Modellreduktion statt gefunden hat. In diesem Fall beträgt der Aufwand dann $O(n \cdot (n + |\phi|)) = O(n^2 + n|\phi|))$.

Aufwand vs. Nutzen Ein relativ gutes Slicing zu berechnen geht (wie gezeigt) sehr effizient. Das *SPDS* kann davon profitieren und das Modell substantiell verkleinern. Im Idealfall erübrigt sich die nachfolgende *Modellprüfung* sogar ganz, wenn die Hauptprozedur *main* (Anfangskonfigurationen) nicht im Slice-Schnitt liegt: $main \notin \Lambda_+ \cap \Lambda_-$. Denn dann konnte bereits statisch nachgewiesen werden, dass es keinen Lauf von den Anfangskonfigurationen zu $L0 - Marken$ gibt und es gilt $\Lambda_+ \cap \Lambda_- \cap Marken(S) = \emptyset$ (siehe Beweis Lemma 4.3.8 auf Seite 75). Je *präziser* das Slicing durchgeführt wird, desto wahrscheinlicher ist es, dass sich die *Modellprüfung* erübrigt, aber desto aufwändiger ist auch das Slicing.

4.3.2.7. Zusammenfassung

Rückblick Es wurde in diesem Abschnitt gezeigt, dass manche Teile der Modellbeschreibung eines *SPDS*-Modell unnötig für eine Modellprüfung sein können und dadurch die Modellprüfung einerseits erschweren und andererseits die Qualität von *Modellanalyse*n negativ beeinträchtigen können. Um unwichtige Modellteile von wichtigen zu trennen, wurde das Konzept Slicing erläutert und an einem Beispiel-*SPDS* verdeutlicht. Dabei wurden Vorwärts- und Rückwärts-Slices bezüglich bestimmter Kriterien betrachtet. Diese Kriterien richteten sich nach den zu prüfenden Eigenschaften. Der Durchschnitt von Vorwärts- und Rückwärts-Slice ist nach Lemma 4.3.11 auf Seite 71 wiederum ein Slice. Daher eignet er sich, ein *SPDS* zu verkleinern (zu beschneiden). Es wurden die Optimalität von Slices erklärt (exakte Slices) und deren Berechnungskomplexität untersucht. Exakte Slices in *SPDS* sind (im Gegensatz zu Hochsprachen) prinzipiell zwar berechenbar, beschleunigen aber wegen ihrer Komplexität die Modellprüfung[33] nicht. Einsetzbare effizientere *Programmanalysen* (Kontroll- und Datenflussanalysen) wurden daher untersucht und daraus eine entsprechende Heuristik (Modellanalysebeispiel) entwickelt, welche zudem temporale Formeln beachtet. Diese Heuristik wurde dann auf ihre Eigenschaften hin untersucht. Sie ist z.B. im Vergleich zu den Techniken aus Abschnitt 4.3.1 auf Seite 45 deutlich effizienter: $O(n + |\phi|)$. Schließlich wurde die

[33] auf Erreichbarkeit

Modellbeschneidung T_{Slice} für *SPDS* entwickelt, welche einen Slice nutzt, um ein *SPDS* zu verkleinern (zu beschneiden). Dies wurde wiederum am Beispiel-*SPDS* verdeutlicht. Auch für T_{Slice} wurden wichtige Eigenschaften wie Entscheidbarkeit, Komplexität und Einfluss auf den *Konfigurationenraum* untersucht und schließlich die Korrektheit dieser Modellreduktion gezeigt. So wurde der Unterschied zwischen Optimalität und Präzision einer Modellbeschreibung geklärt und gezeigt, dass T_{Slice} präzise bezüglich des vorgeschlagenen Slices der Heuristik ist. Optimalität von T_{Slice} ergibt sich dann aus der Optimalität des Slices[34]. Der *Konfigurationenraum* verändert sich nur so stark, dass temporale Formeln noch ihren Wahrheitsgehalt beibehalten.

Fazit Durch die beschriebenen Techniken können nun *SPDS* auf den wichtigen Teil der Modellbeschreibung verkleinert (beschnitten) werden. Unnötige Variablen (d.h. solche, die keinen Beitrag zu den zu prüfenden Eigenschaften haben) werden dabei aus dem Modell entfernt. Wie auch bei der Ausdrucksvereinfachung kann sich dies positiv auf die Modellprüfung und Modellanalysen auswirken (abhängig von der Implementierung). Slicing, wie es aus der untersuchten Literatur im Programmiersprachenumfeld bekannt ist, betrachtet i.d.R. keine temporalen Formeln, sondern spezielle Slicing-Kriterien (z.B. Programmmarken oder Variablen im *System Dependence Graph* (*SDG*)) [100]. Die Idee des temporalen Slicings wird eher bei Datenbanken für temporale XML-Modelle [148] oder UML-Statecharts [246] eingesetzt. Diese Formen der Modelle sind jedoch für *SPDS* ungeeignet (reine Rekursion). Der Modellprüfer SPIN setzt eine Kombination aus Vorwärts- und Rückwärts-Slice für Promela (parallele Prozesse/Protokolle) ein [136]. Der *SDP* wird dort entsprechend für parallele Konzepte wie gemeinsam genutzter Speicher und Kommunikationskanäle (Channel) angepasst, welche es für *SPDS* nicht gibt. Auch dort wird weder Rekursion noch der Erhalt temporaler Aussagen betrachtet (Statement-Slicing). Für die Kriterien K2 und K4 dieser Arbeit (siehe Abschnitt 1.1 auf Seite 7) wurde daher eine Modellbeschneidung für *SPDS* entwickelt, um Rekursion **und** temporale Formeln zugleich zu berücksichtigen.

4.3.3. Prozedurüberbrückung

In diesem Abschnitt werden *Prozeduraufruf*e als überflüssig erkannt und ggf. übersprungen, was vorrangig die *Modellprüfung* beschleunigt. Es seien daher wie bei der *SPDS*-Beschneidung die zu prüfende Formel ϕ bzw. die auf *Erreichbarkeit* zu prüfenden *SPDS*-Marken $L0 = \{l0, l0', ...\}$ a priori bekannt.

4.3.3.1. Motivation

Verschiedene *Prozeduraufruf*e in einem *SPDS* S sind überflüssig und können überbrückt werden, wenn sie weder die *Erreichbarkeitsprüfung* noch eine *temporale Formel* beeinflussen. Diese Überbrückung (Abkürzung der Läufe) bewirkt eine Verbesserung der *Modellprüfung* sowie der *Modellanalyse*n, weil der *Prozeduraufruf* dann nicht mehr analysiert werden braucht. Im Idealfall werden alle *Prozeduraufruf*e an eine Prozedur m als unnötig identifiziert. Dann wird die Prozedur m selbst überflüssig in der *SPDS*-Beschreibung und kann eliminiert werden.

Problembewusstsein Eine Prozedur kann im Vorwärts- und Rückwärts-Slicing liegen, so dass dies bei der *SPDS*-Beschneidung in Abschnitt 4.3.2 ab Seite 68 nicht aus der Prozedur eliminiert wird. Dennoch kann die Prozedur überflüssig sein, wenn die Prozedur z.B. keinen so genannten *Seiteneffekt* [52] auf das *SPDS* ausübt. Z.B. bei aus Hochsprachen automatisch gewonnenen Modellen können derartige überbrückbare Prozeduren entstehen.
Prozeduren können aber nicht nur im gegebenem Modell überflüssig sein, auch durch verschiedene *Modelltransformationsregeln* (z.B. jene aus Abschnitt 4.3.2 ab Seite 68) können Prozeduren erst überbrückbar werden. Durch Anwendung der Transformationsregeln $(S_1) - (S_6)$ aus Abschnitt 4.3.2 ab Seite 68 können (abhängig vom Slice) Entartungen wie z.B. leere Prozeduren entstehen. Solche leeren Prozeduren haben keinen Einfluss auf die restliche *Erreichbarkeit* oder die restlichen

[34] So ist eine optimale Modellbeschneidung möglich, aber aufwändiger als Modellprüfung.

Quellcode 4.10: Leicht modifiziertes Motivationsbeispiel aus *Quellcode* 4.8.

```
int a(8)[100];
init m;

void m() {                          void b(int i(8)) {
    int i(8);                           b0: i = i-1;
m0: i=99;                               b1: if (i>0) goto b2;
m1: b(i);                               b3: return;
m2: heapify(i);                         b2: goto b3;
m3: return;  }                          }

void heapify(int i(8)) {            void q() {
     int p(8);                          q0: b(99);
l0:  if (i>0) heapify(i-1);             q1: heapify(99);
l1:  p = (i-1)/2;                       q2: b(99);
l2:  if (a[p] >= a[i]) return;          }
l3:  a[p] = a[i], a[i] = a[p];
l4:  i = p;
l5:  if (i>0) goto l1; }
```

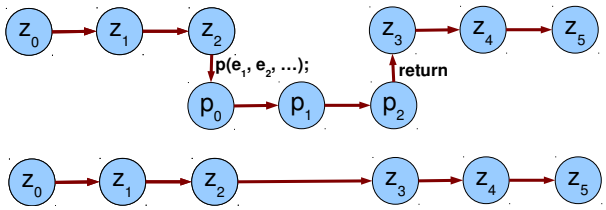

Abbildung 4.3.: Laufänderung bei Prozedurüberbrückung (oben mit Aufruf p, unten ohne)

*Variablenbelegung*en, da sie aufgerufen und sofort wieder beendet werden. Man nennt diese Nichtbeeinflussung des Restmodells *Seiteneffekt frei*. Bestimmte Seiteneffekt freie Prozeduren können übergangen werden, wenn nicht die *Erreichbarkeitprüfung* oder der *Wahrheitsgehalt* einer *temporale Formel* geändert wird.

Beispiel 4.3.4 *Als Beispiel zur Demonstration diene* Quellcode *4.10, eine leicht modifizierte Version von Beispiel 4.3.3 auf Seite 68. Die Hauptprozedur ist m. Die Prozedur b ruft sich nicht selber auf und ändert die globale* Reihung *a nicht. Sei wieder wie im Abschnitt 4.3.2 ab Seite 68 angenommen, dass im Rahmen einer* Modellprüfung *die* Erreichbarkeit *der Marken* $L0 = \{l0\}$ *zu prüfen sind. Dann wird durch Analyse des Verhaltens klar, dass die Prozedur b zwar an der Marke* $m1$ *aufgerufen wird, wegen der Erreichbarkeit von* $b3$ *dann aber keinen Einfluss auf die* Erreichbarkeitsprüfung *von* $L0$ *hat. Auch wenn b nicht aufgerufen würde (*$m1 : skip$*;) ergibt sich unabhängig vom Restmodell das gleiche* Modellprüfung*sresultat.*

Lösungsidee Die Idee ist, wie Abbildung 4.3 für einen Lauf des *SPDS* zeigt, unnötige *Prozeduraufrufe* zu erkennen und zu unterbinden. Um überbrückbare Prozeduren zu identifizieren, kann eine *Seiteneffektanalyse* eingesetzt werden. Zudem ist es nötig, dass Läufe nicht innerhalb von derartigen Prozeduren beenden, da sonst diese Läufe durch die Prozedurüberbrückung im weiteren Modell zu zusätzlichen nicht realisierbaren Läufen und damit zu False Negatives führen kann. In beiden Fällen wird dazu der Kontrollfluss und der Datenfluss *abstrakt* untersucht. Überflüssige Prozeduren, welche dann nirgends mehr aufgerufen werden, können dann durch die *SPDS*-Beschneidung aus Abschnitt 4.3.2 ab Seite 68 identifziert und eliminiert werden.

4.3.3.2. Notwendige abstrakte Informationen

Es ist zu identifizieren, wann eine Prozedur *überbrückbar* ist. Dazu sind die folgenden drei abstrakten Informationen mittels *Modellanalyse*n zu bestimmen:
Erstens ist es wichtig, ob eine Prozedur *Seiteneffekte* auf vorgegebene *Variablenbelegung*en ausübt.

Definition 4.3.18 (*Seiteneffekt* τ_{eff})
Eine Prozedur p hat ***keinen*** Seiteneffekt *bezüglich einer Menge $W \subseteq ENV$ an* Variablenbelegung*en gdw. die Prozedur bei Aufruf mit jeder* Variablenbelegung *$env \in W$ in einer unveränderten globalen* Variablenbelegung *env_{Vgbl} endet:*

$$\forall s \in Post^*((env_{Vgbl}, (env_{loc_p}, l_p))) : (\nexists s' : s \to s') \Rightarrow env^s = env_{Vgbl}$$

Die abstrakte Information $\tau_{eff} : 2^{ENV} \times Prz(S) \to \{true, false\}$ gibt an, ob p bezüglich W möglicherweise einen Seiteneffekt *hat $\tau_{eff}(W, p) = true$ oder nicht $\tau_{eff}(W, p) = false$. τ_{eff} muss* konservativ *sein, so dass $\tau_{eff}(W, p) = true$ sein darf auch wenn p bezüglich W* Seiteneffekt *frei ist.*

Zweitens ist es wichtig, ob in die Prozedur eintretende Läufe mit entsprechenden *Variablenbelegung*en ggf. nicht wieder aus der Prozedur austreten (z.B. wegen unendlicher Schleife). Denn zur Überbrückung von Prozeduren muss ebenso garantiert sein, dass der Lauf welcher den *Prozeduraufruf* überbrückt, nicht evtl. in der Prozedur endet (bzw. versackt).
Versackung für SPDS-*Anweisungen* kann dabei wie folgt formal definiert werden:

Definition 4.3.19 (*Versackende SPDS*-Anweisung)
*Eine SPDS-*Anweisung *$l : z \in Stats(S)$ ist* ***nicht*** versackend *oder* versackungsfrei *bezüglich einer Menge $W \subseteq ENV$ an* Variablenbelegung*en gdw. es stets eine* Nachfolgekonfiguration *bei Anwendung von s gibt: $\forall s \in Konf(S) : env^s \in W \Rightarrow \exists s' \in Konf(S) : s \hookrightarrow_l s'$. Die abstrakte Information $\tau_{stop} : 2^{ENV} \times Stats(S) \to \{true, false\}$ gibt an, ob die SPDS-*Anweisung *$l : z$ bezüglich W möglicherweise versackend ist $\tau_{stop}(W, l : z) = true$ oder definitiv nicht versackt $\tau_{stop}(W, l : z) = false$.*

Auf Prozeduren erweitert ergibt dies folgendes.

Definition 4.3.20 (*Versackung* τ_{stop} für Prozeduren)
Eine Prozedur p ist ***nicht*** versackend *oder* versackungsfrei *bezüglich einer Menge $W \subseteq ENV$ an* Variablenbelegung*en gdw. die Prozedur bei Aufruf mit jeder* Variablenbelegung *$env \in W$ stets mit einem return-Konfigurationenübergang beendet*[35]*. Die abstrakte Information $\tau_{stop} : 2^{ENV} \times Prz(S) \to \{true, false\}$ gibt an, ob p bezüglich W möglicherweise versackend ist $\tau_{stop}(W, p) = true$ oder nicht $\tau_{stop}(W, p) = false$. τ_{stop} muss* konservativ *sein, so dass $\tau_{stop}(W, p) = true$ sein darf auch wenn p bezüglich W nicht versackend ist.*

Und drittens muss analysiert werden, ob die Überbrückung der Prozedur das Resultat der *Modellprüfung* (*Erreichbarkeit* von $L0$ bzw. *Wahrheitsgehalt* von ϕ) beeinträchtigt. Dabei ist jedoch ein gewisser Grad an Beeinflussung gestattet. Denn wie Abbildung 4.3 zeigt, beeinträchtigt natürlich die Überbrückung einer Prozedur die Länge eines Laufs, so dass lediglich Logiken als invariant in Frage kommen, welche nicht die Länge eines Laufs „messen“ können. Wie in Abschnitt 3.3.4 ab Seite 37 erläutert, trifft dies für die Logiken ohne den Nachfolgeoperator X zu (ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X).

Definition 4.3.21 (*Invarianz-X* τ_{inv-X})
Sei $G = (V, E)$ der interprozedurale Kontrollflussgraph eines SPDS *S. Eine Prozedur p heißt* invariant-X *bezüglich einer Menge $W \subseteq ENV$ an* Variablenbelegung*en, wenn bei jeder* Variablenbelegung *$env \in W$ keine die* Modellprüfung *beeinflussende Marke aus $L0$ während der Abarbeitung von p auftritt:*

$$\forall s \in Post^*(I_W) : Marken(s) \cap L0 = \emptyset,$$

[35] Kein Lauf des zu Grunde liegenden *PDS* terminiert oder gerät in eine Endlosschleife in p.

wobei $I_W := \{(env_{Vgbl}, (env_{loc_p}, l_p)) \mid env \in W\}$. *Für eine a priori gegebene* temporale Formel ϕ *wird* $L0 = L0_\phi$ *dabei derart bestimmt, dass gilt:*

$$\forall v \in Vars(S),\, „l_1 : v = e;\ l_2 : ...“ \in S \quad : \quad v \in \phi \Rightarrow l_1, l_2 \in L0_\phi \tag{4.13}$$

$$\forall p \in Prz(S), v \in loc_p \quad : \quad v \in \phi \Rightarrow l_p \in L0_\phi \tag{4.14}$$

$$\forall v \in Vgbl(S) \quad : \quad v \in \phi \Rightarrow Marken(I) \subseteq L0_\phi \tag{4.15}$$

$$\forall l \in Marken(S), l_1 \rightarrow_E l \rightarrow_E l_2 \quad : \quad l \in \phi \Rightarrow l_1, l, l_2 \in L0_\phi \tag{4.16}$$

$$\forall p \in Prz(S), l \in Marken(p), l_1 \rightarrow_E l \rightarrow_E l_2 \quad : \quad p \in \phi \Rightarrow l_1, l, l_2 \in L0_\phi \tag{4.17}$$

Die abstrakte Information $\tau_{inv-X} : 2^{ENV} \times Prz(S) \rightarrow \{true, false\}$ *gibt an, ob* p *bezüglich* W *invariant-X ist* $\tau_{inv-X}(W, p) = true$ *oder evtl. nicht* $\tau_{inv-X}(W, p) = false$. τ_{inv-X} *muss* konservativ *sein, so dass* $\tau_{inv-X}(W, p) = false$ *sein darf auch wenn* p *invariant-X bezüglich* W *ist.*

Man beachte, dass nach (4.17) sofort $Marken(p) \subseteq L0$ folgt, wenn $p \in \phi$. Zusätzlich kommen aber auch noch jeweils eine Marke vor und hinter einem solchen *Prozeduraufruf* in $L0$.

Eigenschaften Die Berücksichtigung von bestimmten vorgegebenen *Variablenbelegung*en für die notwendigen abstrakten Informationen führt zu *präzise*ren Analyseergebnissen als wenn Prozeduren mit beliebigen *Variablenbelegung*en betrachtet werden, da es deutlich weniger Möglichkeiten sowie Fallunterscheidungen für die Kontrollflüsse gibt.

Lemma 4.3.12 (Monotonie von τ_{eff}, τ_{stop} und τ_{inv-X})
τ_{eff}, τ_{stop} und τ_{inv-X} sind monoton im ersten Argument. Es gilt für beliebiges $p \in Prz$ und beliebige Mengen von Variablenbelegung*en $A, B \subseteq ENV$ mit $A \subseteq B$:*

$$\begin{aligned} \tau_{eff}(A, p) &\Rightarrow \tau_{eff}(B, p) \\ \tau_{stop}(A, p) &\Rightarrow \tau_{stop}(B, p) \\ \tau_{inv-X}(B, p) &\Rightarrow \tau_{inv-X}(A, p) \end{aligned}$$

Beweis Dies ist eine Folgerung aus den Definitionen, wie nun gezeigt wird. Seien $p \in Prz$ und $A \subseteq B \subseteq ENV$ derart beliebig gewählt, dass gilt: $\tau_{eff}(A, p)$. D.h. p habe bezüglich A einen *Seiteneffekt*. Wegen $A \subseteq B$ hat p dann aber auch einen Seiteneffekt bezüglich B, da die zur Realisierung des *Seiteneffekts* notwendigen *Variablenbelegung*en dann auch bereits in B enthalten sind. Somit gilt dann auch $\tau_{eff}(B, p)$ und damit die Behauptung $\tau_{eff}(A, p) \Rightarrow \tau_{eff}(B, p)$. $\tau_{stop}(A, p) \Rightarrow \tau_{stop}(B, p)$ und $\tau_{inv-X}(B, p) \Rightarrow \tau_{inv-X}(A, p)$ wird analog gezeigt.

□

Ein *Prozeduraufruf* $p(x_1, x_2, \ldots x_n)$ mit den Parametervariablen $Param_p = \{p_1, p_2, \ldots p_n\}$ an der Marke l des *SPDS* S hat keinen *Seiteneffekt* bezüglich der realisierbaren *Variablenbelegung*en $RENV_l$, wenn für

$$RENV_l^p := \{env|_{p_1}^{[\![x_1]\!]_{env}}|_{p_2}^{[\![x_2]\!]_{env}} \ldots |_{p_n}^{[\![x_n]\!]_{env}} \mid env \in RENV_l\} \tag{4.18}$$

gilt: $\tau_{eff}(RENV_l^p, p) = false$. Analog ist er nicht versackend bzw. invariant-X, wenn gilt:

$$\tau_{stop}(RENV_l^p, p) = false \quad bzw. \quad \tau_{inv-X}(RENV_l^p, p) = true.$$

Die abstrakten Informationen sind exakt, wenn sie jeweils nicht *konservativ* bestimmt sind:

- τ_{eff} ist exakt, falls $\tau_{eff}(W, p) = false$ gdw. p bezüglich W *Seiteneffekt* frei ist.
- τ_{stop} ist exakt, falls $\tau_{stop}(W, p) = false$ gdw. p bezüglich W nicht versackend ist.
- τ_{inv-X} ist exakt, falls $\tau_{inv-X}(W, p) = true$ gdw. p invariant-X bezüglich W ist.

Wiederum im Gegensatz zu gängigen Hochsprachen (Halteproblem) sind diese exakten abstrakten Informationen für *SPDS* berechenbar.

Satz 4.3.12 (Entscheidbarkeit exakter τ_{eff}, τ_{stop}, τ_{inv-X})
Exakte abstrakte Informationen τ_{eff}, τ_{stop} und τ_{inv-X} sind entscheidbar.

Der Beweis findet sich im Anhang B ab Seite 145. Erwartungsgemäß ergibt sich wegen der komplizierten Berechnungen der exakten abstrakten Informationen eine entsprechende Komplexität.

Satz 4.3.13 (Komplexität exakter τ_{eff}, τ_{stop}, τ_{inv-X})
τ_{eff}, τ_{stop}, τ_{inv-X} *sind mindestens so schwer wie die* Erreichbarkeitsprüfung.

Beweis (Skizze) Für τ_{inv-X} ist dies einfach einzusehen, da die Aufgabe äquivalent ist mit der Prüfung der *Erreichbarkeit* der Marken $L0$. Ähnliches gilt für τ_{eff}, da hier nur mittels W aus realisierbare *Variablenbelegung*en betrachtet werden. τ_{stop} ist mindestens so schwer wie die *Erreichbarkeitsprüfung*, da sie die Erreichbarkeit von l_2 vom Beispiel im Beweis zu τ_{stop} in Satz 4.3.12 löst für jede beliebig gestaltete Prozedur p, auch wenn in der Praxis ggf. bereits wenige Simulationsversuche genügen.

□

Exakte abstrakte Informationen lohnen sich daher i.d.R. nicht, um damit eine einzige *Modellprüfung* zu beschleunigen. Sie werden im Folgenden daher approximativ (*konservativ*) bestimmt.

4.3.3.3. Einsetzbare Programmanalysen

Erste Analysen zum Verhalten von Prozeduren (*Seiteneffektanalysen*) gehen zurück in die 1970er auf Spillman, Allen, Rosen und Barth [139] und sind in der Literatur mittlerweile gut untersucht. Ursprünglich für FORTRAN wurden semantische Effekte von Prozeduren mittels interprozeduralen Datenflussanalysen zusammengetragen und spezielle Mengen MOD bzw. REF berechnet. Diese geben an, welche Variablen ihren Wert potentiell ändern bzw. welche Variablen potentiell verwendet werden. Zeiger wurden damals nicht unterstützt. Da *SPDS* aber über keine Zeiger verfügen, könne diese Analysen übertragen werden. Globale Datenflussanalysen mittels Multi-Graphen und Zeigeranalysen (alias analysis) wurden durch Banning [22] untersucht. Eine Übersicht zu verschiedenen Arten von *Seiteneffektanalysen* findet sich in [52]. Auch sie können auf *SPDS* übertragen werden. Da es in *SPDS* keine komplexen Konstrukte wie Zeiger, Referenzparameter, Klassen oder Ströme gibt, genügen diese vergleichsweise einfachen Analysen. Im Gegensatz zu typischen *Seiteneffektanalysen* für höhere *Programmiersprachen* ist die hier aufgeführte Definition der abstrakten Informationen abhängig von *Variablenbelegung*en und bestimmen keine (möglicherweise großen) Variablenmengen. Somit kann eine Prozedur bereits als *seiteneffektfrei* identifiziert werden, wenn es lediglich nur für die z.B. realisierbaren *Variablenbelegung*en keinen *Seiteneffekt* auf das Modell besitzt. Zudem ermöglicht diese Form von Seiteneffekten eine einfache Unterscheidung nach Aufrufkontexten. Letztlich ist zur Prozedurüberbrückung für diesen Abschnitt aber lediglich von Interesse, ob eine Prozedur p einen *Seiteneffekt* hat oder nicht. Wie dieser *Seiteneffekt* im Detail aussieht, ist an dieser Stelle unwichtig und vereinfacht somit einsetzbare Analysen.
Terminierungs-Analysen [12, 11, 165, 230] werden eingesetzt, um gezielt die Terminierung von *Programmen* zu untersuchen. Wegen des Halteproblems von Turingmaschinen kann natürlicher Weise nicht für jedes *Programm* einer turingmächtigen Hochsprache die Terminierung entschieden werden. Terminierungsanalysen sind einfache Formen einer *Modellprüfung* und liefern *unpräzise* aber effizient approximierte Terminierungsaussagen (z.B. indem auf den Kontrollfluss beschränkt und vom Datenfluss abstrahiert wird). Sie können zwar auch wie *Seiteneffektanalysen* auf Daten- und Kontrollflussanalysen basieren, allerdings werden hier oft auch andere Techniken eingesetzt. So basieren [12, 11, 165] auf so genannten Größen-Änderungs-Graphen (Size-Change Graphs). Auch solche Techniken können für τ_{stop} auf *SPDS* übertragen werden.
Für die Invarianz-X wird schließlich die Analyse der *Erreichbarkeit* benötigt. Dazu können approximative Erreichbarkeitstest durchgeführt werden. D.h. es werden einerseits für einen positiv Test einige Läufe simuliert mittels einiger *Variablenbelegung*en und abstrahiert andererseits sehr

stark vom *Programmverhalten* für eine einfache *Erreichbarkeitsanalyse* (*Modellprüfung*). Z.B. kann komplett *konservativ* vom Datenfluss abstrahiert werden und lediglich über den Kontrollfluss *erreichbare SPDS*-Marken betrachten.

4.3.3.4. Modellanalysebeispiele

Im Folgenden werden *Modellanalysen* für τ_{eff}, τ_{stop} und τ_{inv-X} vorgestellt, um zu erkennen, wann eine Prozedur *überbrückbar* ist.

Idee Datenfluss- und Kontrollflussanalysen können *konservatives* τ_{eff}, τ_{stop} und τ_{inv-X} bestimmen. Dazu werden *Prozeduraufrufe* betrachtet und dortige *Variablenbelegungen* mittels der *Wertebereichsanalyse* approximiert ($RENV^*_{l_p}$).

Konkretisierung Zuerst wird τ_{eff} *konservativ* approximiert. Dies ist mittels weniger einfacher Modelleigenschaften möglich, wie folgendes Lemma zeigt.

Lemma 4.3.13 (*Konservative Seiteneffektfreiheit* τ_{eff})
Eine Prozedur p ist für die approximierten Variablenbelegung*en* $RENV^*_{l_p}$ seiteneffektfrei

$$\tau_{eff}(RENV^*_{l_p}, p) = false,$$

falls p keine Zuweisung *an globale Variablen enthält und alle von p potentiell aufrufbaren Prozeduren q* Seiteneffekt *frei sind für die approximierten* Variablenbelegung*en* $RENV^*_{l_q}$*, d.h.*

$$\tau_{eff}(RENV^*_{l_q}, q) = false.$$

Der Beweis findet sich im Anhang B ab Seite 145. Die Existenz von *Zuweisung*en an globalen Variablen kann sehr einfach im attributiertem Syntaxbaum bestimmt werden. Dazu wird ein so genannter *Aufrufgraph* konstruiert:

Definition 4.3.22 (*Aufrufgraph*)
Der Aufrufgraph $Call = (Prz(S), C)$ eines SPDS S *besteht aus den Prozeduren $Prz(S)$ als Knoten und den Kanten $C \subseteq Prz(S) \times Prz(S)$ mit $(p, q) \in C$, falls p potentiell q aufruft (attributierter Syntaxbaum). Weiter ist $C(p) := \{q \mid (p, q) \in C\}$.*

In diesem können alle potentiell aufrufbaren Prozeduren $m_1, m_2, \ldots m_n$ einer Prozedur p gefunden werden. Gibt es keine und sind sonst sämtliche Voraussetzungen erfüllt, so ist p als *seiteneffektfrei* identifiert: $\tau_{eff}(p) = false$. Gibt es allerdings potentiell aufrufbare Prozeduren $m_1, m_2, \ldots m_n$, so sollten diese zunächst erst einmal auf *Seiteneffekte* hin untersucht werden. Kommt es dabei zu zyklischen Abhängigkeiten, so sind sämtliche am Zyklus beteiligten Prozeduren nicht *seiteneffektfrei*. Abbildung 4.4 fasst dieses Vorgehen als Pseudocode übersichtlich zusammen. Dabei besteht die Eingabe aus dem *SPDS* S und dem Aufrufgraphen $Call = (Prz(S), C)$.
Im ersten Schritt werden die noch zu berechnenden Prozeduren in der Menge M initialisiert. In Schritt zwei werden sämtliche an einem Zyklus beteiligten Prozeduren als nicht *Seiteneffekt* frei markiert: $\tau_{eff}(ENV, p_j) = true$. Nach Schritt 2 gibt es in den noch zu berechnenden Prozeduren M keinen Zyklus mehr. $C \cap M \times M$ ist daher ein Wald (Kreis freier Graph). Beginnend bei den Blättern kann dann ab Schritt drei τ_{eff} sukzessive entsprechend berechnet werden. Wegen $C(p) \cap M = \emptyset$ sind in Schritt 5 alle von p aufgerufenen Prozeduren $C(p)$ bereits berechnet, so dass das Verfahren effizient terminiert (Zyklussuche via Tiefensuche möglich).
Analog zu *Seiteneffekten* τ_{eff} kann auch die Versackungsfreiheit τ_{stop} *konservativ* an Hand weniger einfacher *SPDS*-Eigenschaften bestimmt werden.

Lemma 4.3.14 (*Konservative* Versackungsfreiheit τ_{stop})
Eine Prozedur p ist für beliebige Variablenbelegung*en $W \subseteq ENV$ versackungsfrei $\tau_{stop}(W, p) = false$, wenn p keinen arithmetischen Überlauf und keine Division mit 0 sowie keine Schleifen*[36] *enthält und alle von p aufgerufenen Prozeduren ebenfalls versackungsfrei sind.*

[36] zyklenfreier **inter**prozeduraler Kontrollflussgraph beginnend bei l_p (Zyklus entsteht z.B. bei rekursivem Aufruf)

Eingabe: *SPDS* S und Aufrufgraph $Call = (Prz(S), C)$
Ausgabe: *konservatives* τ_{eff}

1. Setze $M = Prz(S)$. (Noch zu berechnende Prozeduren)
2. Solange ein Zyklus $(p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n), (p_n, p_1) \in C$ existiert mit einem $p_j \in M$, setze $\tau_{eff}(ENV, p_j) = true$ und $M = M \setminus \{p_j\}$.
3. Wähle $p \in M$ mit $C(p) \cap M = \emptyset$ (ein Blatt). Wenn unmöglich, dann fertig.
4. Wenn p eine *Zuweisung* an globale Variable $x \in Vgbl$ hat, dann setze $\tau_{eff}(ENV, p) = true$ und $M = M \setminus \{p\}$. Gehe zu 3.
5. Setze $\tau_{eff}(ENV, p) = false$, falls $C(p) = \emptyset$ und andernfalls $\bigvee_{q \in C(p)} \tau_{eff}(ENV, q)$. Dann setze $M = M \setminus \{p\}$ und gehe zu 3.

Abbildung 4.4.: Pseudocode zur Bestimmung von τ_{eff}.

Der Beweis findet sich im Anhang B ab Seite 145. Rekursive Prozeduren sind bei dieser *konservativen* Versackungsfreiheit automatisch ausgeschlossen, da diese im interprozeduralen Kontrollflussgraph eine Schleife bilden würden. Ein möglicher arithmetischer Überlauf und eine potentielle Division mit 0 kann durch die *Wertebereichsanalyse* und eine einfache Typprüfung erkannt werden.

Lemma 4.3.15 (*Auswertbarkeit* von *Ausdrücken* und *Zuweisungen*)
Seien Variablenbelegung*en mittels der* Wertebereichsanalyse $values_l$ *an der Marke* l *approximiert:* $RENV_l^*$. *Ein Ausdruck* $e \in Expr$ *kann für realisierbare* Variablenbelegung*en* $RENV_l$ *ohne Division mit* 0 *ausgewertet werden, falls* $\forall env \in RENV_l^* : [\![e]\!]_{env} \neq \bot$. *Andernfalls kann* e *potentiell zu einer Division mit* 0 *führen. Ein arithmetischer Überlauf für eine* Zuweisung *„*$x = e$*“ an der Marke* l *ist ausgeschlossen, wenn* $range_l^*(e) \subseteq range(x)$ *gilt. Andernfalls kann es potentiell zu einem arithmetischen Überlauf kommen.*

Beweis Dies ist eine Folgerung aus der *Konservativität* der *Wertebereichsanalyse*, weil damit $RENV_l \subseteq RENV_l^*$.

□

Abbildung 4.5 nutzt die Eigenschaften des Lemmas 4.3.15 zusammen mit der *Wertebereichsanalyse* zur Berechnung von τ_{stop}. Das Vorgehen ist analog wie bei τ_{eff}. $\tau_{stop}(ENV, p)$ erhält nur dann den Wert $false$, wenn sämtliche Bedingungen aus Lemma 4.3.14 erfüllt sind.
Einfacher kann τ_{inv-X} approximiert werden, indem von Daten und des Datenflusses abstrahiert wird. Beginnend beim Prozedureintritt $l_p \in Marken(p)$ einer Prozedur p wird dazu ein intraprozeduraler Vorwärtsslice Λ_+ bestimmt (siehe Definition 4.3.16 auf Seite 72). Überdeckt der Slice Λ_+ ein *Prozeduraufruf* an eine Prozedur $q \subseteq C(p)$, so wird der Slice für diese Prozedur sukzessive um intraprozedurale Slices beginnend bei l_q erweitert, bis alle (rekursiv) potentiell aufrufbaren Prozeduren betrachtet wurden.

Definition 4.3.23 (*Semi-Interprozeduraler Vorwärtsslice* $\Lambda^*_{l_p+}$)
Sei Λ_{l+} *der mittels Definition 4.3.16 auf Seite 72 unter* Verwendung *von* l *statt* l_{main} *und der intra- statt interprozeduralen Kontrollflussgraphen bestimmbare intraprozedurale Vorwärts-Slice beginnend bei Marke* l. *Der Slice* Λ_{l_p+} *wird ergänzt zu* $\Lambda^*_{l_p+}$, *indem in ihm enthaltene* Prozeduraufrufe *weiter verfolgt werden (semi-interprozedural, durchführbar via Breitensuche).*

Lemma 4.3.16 (*Konservative* Invarianz-X τ_{inv-X})
Sei $\tau_{inv-X}(ENV, p) = true$ *gdw.* $L0 \cap \Lambda^*_{l_p+} = \emptyset$. *Dann ist* $\tau_{inv-X}(ENV, p)$ *gemäß Definition 4.3.21 auf Seite 80 eine* konservative *abstrakte Information für Invarianz-X bezüglich* ENV *und damit auch* $RENV_l$.

Eingabe: *SPDS* S, Ergebnisse der *Wertebereichsanalyse*, intraprozedurale Kontrollflussgraphen und Aufrufgraph $Call = (Prz(S), C)$
Ausgabe: *konservatives* τ_{stop}

1. Setze $M = Prz(S)$. (Noch zu berechnende Prozeduren)
2. Solange ein Zyklus $(p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n), (p_n, p_1) \in C$ existiert mit einem $p_j \in M$, setze $\tau_{stop}(ENV, p_j) = true$ und $M = M \setminus \{p_j\}$.
3. Wähle $p \in M$ mit $C(p) \cap M = \emptyset$ (ein Blatt). Wenn unmöglich, dann fertig.
4. Wenn p gemäß Lemma 4.3.15 einen potentiellen arithmetischen Überlauf oder eine Division mit 0 enthält, dann setze $\tau_{stop}(ENV, p) = true$ und $M = M \setminus \{p\}$. Gehe zu 3.
5. Der intraprozedurale Kontrollflussgraph von p besitzt min. einen Kreis, dann setze $\tau_{stop}(ENV, p) = true$ und $M = M \setminus \{p\}$. Gehe zu 3.
6. Setze $\tau_{stop}(ENV, p) = false$, falls $C(p) = \emptyset$ und andernfalls $\bigvee_{q \in C(p)} \tau_{stop}(ENV, q)$. Dann setze $M = M \setminus \{p\}$ und gehe zu 3.

Abbildung 4.5.: Pseudocode zur Bestimmung von τ_{stop}.

Beweis Sei angenommen $\tau_{inv-X}(ENV, p) = true$. Man betrachte ein beliebiges

$$s \in Post^*((env_{Vgbl}, (env_{loc_p}, l_p)))$$

für eine beliebige *Variablenbelegung* $env \in RENV_l \subseteq ENV$. Dann gibt es einen Lauf des *SPDS*, so dass $l_p \rightsquigarrow s$. Die über den Kontrollflussgraphen *erreichbaren* Marken sind eine Obermenge der realisierbaren Marken beginnend bei l_p. Daher gibt es einen Kontrollfluss von l_p nach $Marken(s) \subseteq \Lambda^*_{l_p+}$. Wegen $\tau_{inv-X}(ENV, p) = true$ gilt $L0 \cap \Lambda^*_{l_p+} = \emptyset$, d.h. insbesondere $L0 \cap Marken(s) = \emptyset$. Daher gilt: $\forall s \in Post^*((env_{Vgbl}, (env_{loc_p}, l_p))) : Marken(s) \cap L0 = \emptyset$ und $\tau_{inv-X}(ENV, p)$ ist gemäß Definition 4.3.21 auf Seite 80 eine *konservative* abstrakte Information für Invarianz-X bezüglich ENV bzw. $RENV_l$ ($RENV_l \subseteq ENV$).

□

Eigenschaften der Modellanalysebeispiele Das *Modellanalyse*beispiel τ_{stop} ist wegen der verwendeten *Wertebereichsanalyse* flusssensitiv, interprozedural, *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv. Die Pseudocodes 4.4 und 4.5 terminieren in $O(m)$ Durchläufen, wobei $m = |Prz(S)|$, da es nur endlich viele Prozeduren (und darum in Schritt 2 auch nur endlich viele Kreise) gibt und nach jedem Durchlauf von Schritt 3 eine Prozedur aus der Menge M entfernt wird (Monotonie). Die Laufzeit ist geprägt durch die *Wertebereichsanalyse* $O(|Vars| \cdot |Marken(S)| \cdot j \cdot (ex_l \cdot m^2 + k_{max} \cdot Imax) + |Vars| \log(|Vars|))$ (siehe Abschnitt 4.3.1.4 ab Seite 62) und durch den deutlich geringeren Aufwand aus Pseudocode 4.4 $O(m^2 \cdot |Marken(S)|)$ (Schritt 3 geht in Linearzeit via Topologischer Sortierung). τ_{eff} hingegen basiert nicht auf der *Wertebereichsanalyse* (siehe Pseudocode 4.4) und sucht im Wesentlichen *Zuweisung*en an globale Variablen. τ_{eff} ist daher fluss**in**sensitiv, interprozedural (Berücksichtigung des Aufrufgraphen), *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv. Der Aufwand beträgt analog $O(m^2 \cdot |Marken(S)|)$. τ_{inv-X} schließlich ist wiederum flusssensitiv, interprozedural (Berücksichtigung des Aufrufgraphen), *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv. Diese Eigenschaften wie auch der Aufwand für τ_{inv-X} sind durch das eingesetzte Slicing bestimmt[37].

Abstrakte Informationen des Motivationsbeispiels Der Aufrufgraph des Beispielcode 4.10 auf Seite 79 zur Eingabe für die *Modellanalyse*n ist in Abbildung 4.6 zu sehen. Für die *konservati-*

[37] In dieser Stelle wird angenommen, dass jene Form von Slicing aus Abschnitt 4.3.2 auf Seite 68 zum Einsatz kommt.

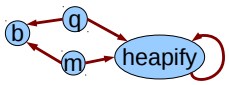

Abbildung 4.6.: Aufrufgraph $Call = (\{m, b, q, heapify\}, C)$ für Beispielcode 4.10

Quellcode 4.11: *Wertebereichsanalyse* von Beispielcode 4.10.

```
int a(8)[100];
init m;

void m() {                  void b(int i(8)) { #i=(99)
    int i(8);   #i=(0,255)    b0: i = i-1;                #i=(98)
m0: i=99;       #i=(99)       b1: if (i>0) goto b2;       #i=(98)
m1: b(i);       #i=(99)       b3: return;                 #i=(98)
m2: heapify(i);#i=(99)        b2: goto b3;                #i=(98)
m3: return;  }                }

void heapify(int i(8)) {                  void q() {
     int p(8);                  #i=(0,99)     q0: b(99);
l0:  if (i>0) heapify(i-1);     #i=(0,99)     q1: heapify(99);
l1:  p = (i-1)/2;               #p=(0,49)     q2: b(99);
l2:  if (a[p] >= a[i]) return;  #a=(0,255)    }
l3:  a[p] = a[i], a[i] = a[p];  #a=(0,255)
l4:  i = p;                     #i=(0,49)
l5:  if (i>0) goto l1; }
```

ve Seiteneffektanalyse τ_{eff} ergibt sich im Beispielcode 4.10 auf Seite 79 gemäß Pseudocode 4.4: $\tau_{eff}(ENV, heapify) = true$ wegen des Zyklus $(heapify, heapify) \in C$. Danach wird in Schritt 3 $p = b$ gewählt und in Schritt 6 $\tau_{eff}(ENV, b) = false$ gesetzt, da $C(b) = \emptyset$. Dann ergibt sich für Prozedur m ebenso jeweils in Schritt 6 $\tau_{eff}(ENV, m) = \tau_{eff}(ENV, b) \vee \tau_{eff}(ENV, heapify) = true$, sowie $\tau_{eff}(ENV, q) = \tau_{eff}(ENV, b) \vee \tau_{eff}(ENV, heapify) = true$. Somit ist lediglich die Prozedur b als *Seiteneffekt* frei erkannt. Für die *konservative* Versackungsfreiheit τ_{stop} sind die Ergebnisse der *Wertebereichsanalyse* wichtig, welche in *Quellcode* 4.11 in den Kommentaren zu sehen sind. Demnach kann statisch bestimmt werden, dass z.B. die Variable i an der Marke $l5$ stets kleiner 50 ist.

Daraus ergibt sich dann für τ_{stop} gemäß Pseudocode 4.5. Im ersten Schritt ist $\tau_{stop}(ENV, heapify) = true$ wegen des Zyklus $(heapify, heapify) \in C$. Danach wird in Schritt 3 $p = b$ gewählt und in Schritt 6 $\tau_{stop}(ENV, b) = false$ gesetzt, da $C(b) = \emptyset$ und p keinen arithmetischen Überlauf und keine Division mit 0 enthält ($i-1$ erfolgreich *auswertbar* und der Typ $bit(i)$ hinreichend groß für das Ergebnis ist) und der intraprozedurale Kontrollfluss keinen Kreis enthält (zu erkennen in Abbildung 4.7). Dann ergibt sich für Prozedur a ebenso jeweils in Schritt 6 $\tau_{stop}(ENV, m) = \tau_{stop}(ENV, b) \vee \tau_{stop}(ENV, heapify) = true$, sowie $\tau_{stop}(ENV, q) = \tau_{stop}(ENV, b) \vee \tau_{stop}(ENV, heapify) = true$. Somit ist lediglich die Prozedur b als nicht versackend erkannt, da die anderen Prozeduren potentiell über den Aufruf von $heapify$ in einer Endlosschleife (unendliche Rekursion) enden.

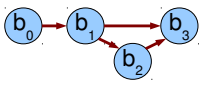

Abbildung 4.7.: Kreis freier intraprozeduraler Kontrollfluss der Prozedur b aus Beispiel 4.3.4.

Für die vorgestellte *konservative* Invarianz-X τ_{inv-X} ergebenen sich für $L0 = \{l0\}$ einerseits

$$\tau_{inv-X}(ENV, heapify) = false,$$

da $l0 \in Marken(heapify)$ und damit $L0 \cap \Lambda_{l_{heapify}+} = \{l0\} \neq \emptyset$ (es gilt $\Lambda_{l_p+} \subseteq \Lambda^*_{l_p+}$). Wegen $heapify \in C(q)$ bzw. $heapify \in C(m)$ ist somit $\tau_{inv-X}(ENV, q) = false$ bzw. $\tau_{inv-X}(ENV, q) = false$, da in beiden Fällen die Marke $l0$ erreicht werden kann. Einzig für die Prozedur b wird die Invarianz-X identifiziert $\tau_{inv-X}(ENV, b) = true$, da $l0$ nicht im semi-interprozeduralen Vorwärtsslice vor kommt: $l0 \notin \Lambda_{l_b+}$.

4.3.3.5. Prozedurüberbrückung T_{skip}

Mit den im vorigem Abschnitt 4.3.3.4 ab Seite 83 erklärten *Modellanalyse*n lassen sich überbrückbare Prozeduren identifizieren. Nun wird beschrieben, wie deren Aufruf verhindert werden kann.

Idee Wie bereits Abbildung 4.3 auf Seite 79 für einen Lauf des *SPDS* zeigt, können unnötige *Prozeduraufruf*e überbrückt werden. Dies geschieht, indem der *SPDS-Prozeduraufruf* in ein *skip* geändert wird. Das funktioniert z.B. dann, wenn p keinen Einfluss auf *Variablenbelegung*en hat (d.h. z_3 die gleichen *Variablenbelegung*en besitzt wie $z_2 : env^{z_3} = env^{z_2}$).

Konkretisierung

Definition 4.3.24 (*Prozedurüberbrückung* T_{skip})
Die Prozedurüberbrückung T_{skip} besteht aus der Modelltransformation*:*

$$\frac{p \in Prz(S) \quad l \in Marken(S) \quad \neg\tau_{eff}(RENV^p_l, p) \quad \neg\tau_{stop}(RENV^p_l, p) \quad \tau_{inv-X}(RENV^p_l, p)}{l : p(x_1, x_2, \ldots, x_n); \;\Rrightarrow\; l : \; skip;} \tag{4.19}$$

Diese wird so lange angewendet wie möglich.

Nur Prozeduren werden überbrückt, welche gemäß Definition 4.3.18 auf Seite 80 sicher keinen *Seiteneffekt* besitzen, gemäß Definition 4.3.20 auf Seite 80 nicht versackend sind und gemäß Definition 4.3.21 auf Seite 80 keinen Einfluss auf Marken $L0$ oder die a priori gegebene *temporale Formel* ϕ haben.

Modelloptimierung des Motivationsbeispiels An Hand der bestimmten abstrakten Informationen kann für die Marken $m1$, $q0$ und $q2$ der *Prozeduraufruf* der Prozedur b überbrückt werden. Es ergibt sich das optimierte Modell aus *Quellcode* 4.12. Danach erkennt die *SPDS*-Beschneidung die Prozedur b als überflüssig und kann es aus der *SPDS*-Beschreibung eliminieren.

Wohldefiniertheit Es gibt nur endlich viele *Prozeduraufruf*e in der endlichen *SPDS*-Beschreibung, welche überbrückt werden können. Daher kann die *Modelltransformation* 4.19 nur endlich oft angewendet werden. Die Prozedurüberbrückung T_{skip} ist darum wohldefiniert.

Korrektheit (Invarianz Temporaler Formeln und Erreichbarkeit)

Satz 4.3.14
Die Prozedurüberbrückung T_{skip} ist CTL-X-invariant für gegebenes $\phi \in$ CTL*-X und nicht LTL-invariant.*

Der Beweis findet sich im Anhang B ab Seite 145.

Satz 4.3.15 (Korrektheit der Prozedurüberbrückung T_{skip})
Die Prozedurüberbrückung T_{skip} ist α-invariant für gegebenes $\phi \in \alpha$ und ein beliebiges $\alpha \in \{$ACTL-X, ACTL-X, CTL-X, CTL*-X, LTL-X$\}$ sowie* Erreichbarkeits-invariant *für a priori gegebene Marken $L0$. Sie ist nicht invariant für $\alpha \in \{$ACTL, ACTL*, CTL, CTL*, LTL$\}$.*

Quellcode 4.12: Per Prozedurüberbrückung optimierter Beispielcode 4.10.

```
int a(8)[100];
init m;

void m() {                    void b(int i(8)) {
    int i(8);                       b0: i = i-1;
m0: i=99;                           b1: if (i>0) goto b2;
m1: skip;                           b3: return;
m2: heapify(i);                     b2: goto b3;
m3: return;  }                      }

void heapify(int i(8)) {        void q() {
     int p(8);                          q0: skip;
l0:  if (i>0) heapify(i-1);             q1: heapify(99);
l1:  p = (i-1)/2;                       q2: skip;
l2:  if (a[p] >= a[i]) return;          }
l3:  a[p] = a[i], a[i] = a[p];
l4:  i = p;
l5:  if (i>0) goto l1; }
```

Beweis Dies ist eine Folgerung aus Satz 4.3.14 und Betrachtungen aus Abschnitt 3.3.5 ab Seite 37 (insb. Abbildung 3.6 auf Seite 41). Demnach ist die Prozedurüberbrückung T_{skip} CTL*-X-invariant für gegebenes $\phi \in$ CTL*-X und nicht LTL-invariant. Aus den Teilmengenbeziehungen der Logiken ergibt sich dann gemäß Abbildung 3.6 auf Seite 41 die Behauptung.

□

4.3.3.6. Eigenschaften

Die Eigenschaften Flusssensitivität, Interprozeduralität, Kontextinsensitivität und Pfadinsensitivität etc. übertragen sich von den eingesetzten Modellanalysen auf die Modellreduktion T_{skip}.

Entscheidbarkeit der optimalen Transformation Im Sinne der bestimmten abstrakten Informationen τ_{eff}, τ_{stop} und τ_{inv-X} ist die Modellreduktion *optimal*, da genau die per τ_{eff}, τ_{stop} und τ_{inv-X} überbrückbaren Prozeduren überbrückt werden. Die definierte Prozedurüberbrückung T_{skip} entscheidet allerdings nur mittels der drei zusammen hinreichenden Kriterien τ_{eff}, τ_{stop} und τ_{inv-X} über die Überbrückung einer Prozedur. Ein zumindest formal einfaches notwendiges Kriterium ist, dass das *Modellprüfung*sresultat erhalten bleibt.

Satz 4.3.16 (*Optimale* Prozedurüberbrückung)
*Es ist **exakt** bestimmbar, ob eine Prozedur p überbrückbar ist oder nicht.*

Beweis Führe die *Modellprüfung* einmal mit und einmal ohne den *Prozeduraufruf* aus. Ist das Modellprüfungsresultat verschieden, so ist p nicht überbrückbar.

□

Die Prozedurüberbrückung ist daher *entscheidbar*. Die beschriebene Prozedurüberbrückung T_{skip} ist i.d.R. nicht *optimal* auch wenn die Analyseergebnisse der *Modellanalyse*beispiele exakt sind (nur hinreichend). Daher ist T_{skip} nicht *präzise*. Denn selbst wenn τ_{eff}, τ_{stop} und τ_{inv-X} exakt sind, kann es sein, dass eine Prozedur z.B. $heapify$ im Beispielcode 4.10 auf Seite 79 zwar *Seiteneffekte* besitzt, sich diese aber nicht auf das *Modellprüfung*sergebnis z.B. $\phi = F\ m1$ auswirken. Für $\phi = F\ m1$ wäre dann auch der *Prozeduraufruf* $heapify$ an der Marke $m2$ überbrückbar.

Fehlalarme Weder False Negatives noch False Positives können durch die Prozedurüberbrückung T_{skip} entstehen, wenn die *Modellprüfung* auf die Logiken ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X sowie die *Erreichbarkeit* beschränkt bleibt (siehe Satz 4.3.15).

Konfigurationenraumveränderung Der theoretische *Konfigurationenraum* verändert sich nicht (weiterhin $P \times \Gamma^*$), da keine *Variablenbelegung*en verändert und auch keine Variablen bzw. Marken eliminiert wurden. Der *erreichbare Konfigurationenraum* allerdings ändert sich sehr wohl. Denn Läufe werden evtl. verkürzt und *Konfigurationen* sind dann evtl. nicht mehr *erreichbar* (siehe Abbildung 4.3 auf Seite 79wo die *Konfigurationen* p_i nicht länger *erreichbar* sind). Eine überbrückte Prozedur p mit m Marken, verkürzt z.B. sämtliche durch p hindurch gehenden Läufe entsprechend. Das genaue Ausmaß der Veränderung wird durch die Form der nicht länger aufgerufenen Prozeduren bestimmt.

Komplexität Der Aufwand zum Anwenden der *Modelltransformationsregel* (4.19 auf Seite 87) für die Prozedurüberbrückung T_{skip} ist mit $O(|Prz(S)|)$ vernachlässigbar klein gegenüber des Aufwandes für die *Modellanalyse*beispiele. Diese sind insbesondere durch den Aufwand der *Wertebereichsanalyse* bestimmt.

Aufwand vs. Nutzen Für die praktische Anwendung ist klar, dass wegen der Monotonie (siehe Lemma 4.3.12 auf Seite 81) für $RENV_l^p \subseteq RENV_l^{*p} \subseteq ENV$ gilt: $\tau_{eff}(RENV_l^p, p) \Rightarrow \tau_{eff}(RENV_l^{*p}, p)$ sowie $\tau_{eff}(RENV_l^p, p) \Rightarrow \tau_{eff}(ENV, p)$. Somit genügt als hinreichende Bedingung auch die *Verwendung* von ENV oder $RENV_l^{*p}$ an Stelle von $RENV_l^p$ in der *Modelltransformation* (4.19 auf Seite 87). Die angegebenen *Modellanalyse*beispiele sind unabhängig von konkreten *Variablenbelegung*en, so dass für diese weder bei der *Modellanalyse* noch bei der *Modelltransformation* ein solcher Parameter in der Implementation berücksichtigt werden braucht.

Eine effizientere aber *unpräzise*re Prozedurüberbrückung wäre durch Verzicht der *Wertebereichsanalyse* möglich. Da diese aber ohnehin auch für andere *Modellanalyse*n benötigt wird, ist an dieser Stelle dessen *Verwendung* sinnvoll.

4.3.3.7. Zusammenfassung

Rückblick Obwohl die vorgestellte Modellbeschneidung (Abschnitt 4.3.2 auf Seite 68) in der Lage ist, unwichtige Teile der Modellbeschreibung mittels Slicing zu identifizieren und dann zu eliminieren, wurden in diesem Abschnitt die Grenzen davon erläutert und an einem *SPDS*-Beispiel verdeutlicht. So wurde gezeigt, dass es unwichtige Prozeduren geben kann, welche durch das vorgestellte Slicing nicht erkannt werden. In diesem Abschnitt wurde erläutert, wie unwichtige Prozeduren dennoch erkannt und schließlich überbrückt und damit das Modell vereinfacht werden kann. Dazu wurden die Konzepte Seiteneffekt, Versackung und Invarianz-X bezüglich gegebener temporaler Formeln vorgestellt und näher erläutert. Diese abstrakten Informationen sind für *SPDS* jeweils exakt bestimmbar. Deren exakte Bestimmung würde aber aufwändiger sein als die Modellprüfung selbst. Daher wurden einsetzbare Programmanalysen zu Einsatzzwecken untersucht und daraus entsprechend Heuristiken als Modellanalysebeispiele konstruiert, um konservative Approximationen zu erhalten. Eigenschaften dieser Heuristiken wurden untersucht und deren Konservativität nachgewiesen. Der Aufwand dieser Heuristiken wird durch die eingesetzte Wertebereichsanalyse und durch das eingesetzte Slicing bestimmt. Am *SPDS*-Beispiel wurden diese Heuristiken vorgeführt und schließlich die Prozedurüberbrückung T_{skip} konstruiert. Am *SPDS*-Beispiel wurde diese vorgeführt und es wurden Eigenschaften wie Entscheidbarkeit, Komplexität und Einfluss auf den *Konfigurationenraum* untersucht. Schließlich wurde auch die Korrektheit dieser Modellreduktion gezeigt. So ist T_{skip} nicht optimal wegen der Konservativität der Modellanalysen und verursacht potentiell neue Fehlalarme, wenn temporale Formeln mit einem X-Operator verwendet werden. Werden temporale Formeln ohne den X-Operator verwendet, so ist deren Wahrheitsgehalt bezüglich der Modellreduktion T_{skip} invariant.

Fazit Die präsentierte Prozedurüberbrückung T_{skip} ermöglicht es nun, unwichtige Prozeduren in *SPDS* zu überbrücken, auch wenn diese Prozeduren nicht *tot* sind und durchlaufen werden. Im Gegensatz zur Modellbeschneidung, werden bei der Prozedurüberbrückung auch im Slice liegende Prozeduren überbrückt. Dies beschleunigt die Modellprüfung wie auch andere Modellanalysen, insbesondere dann, wenn dadurch derartige Prozeduren durch die Modellbeschneidung T_{slice} erst beschnitten werden können. Die entwickelte Prozedurüberbrückung T_{skip} kann helfen, bei gleichem Ressourcenaufwand eine größere Rekursionstiefe zu erreichen (wichtig für endliche Modelle bzw. beschränkte Modellprüfung). Untersuchte Arbeiten im Bereich der Modellprüfung leisten dies nicht, da Modelle dort häufig endlich sind und dann über keine adäquate Prozedursemantik verfügen (z.B. Beschränkung der Aufruftiefe bei beschränkter Modellprüfung). Unendliche Rekursion mit Prozeduren wird nur durch unbeschränkte Modelle wie z.B. *SPDS* ermöglicht. Aus dem Übersetzerbau hingegen können Seiteneffektanalysen und Terminierungsanalysen eingesetzt werden. So werden Seiteneffektanalysen genutzt, um optimierenden Übersetzern mit wichtigen Informationen zu Anweisungen zu versorgen wie Lese-/Schreib-Mengen (read-/write-set) oder Speicherorte für Variablenzugriffe [52]. Dabei spielt vielfach lediglich das beobachtbare Verhalten des Programms eine Rolle [52]. Im Hinblick auf temporale Invarianz beim Überbrücken von Prozeduren in *SPDS* sind die Untersuchungen und Entwicklungen dieses Abschnitt gegenüber der betrachteten Literatur neu.

4.3.4. Variablenredundanzelimination

In diesem Abschnitt wird der *Konfigurationenraum* komprimiert, indem Redundanzen minimiert werden. *SPDS*-Variablen werden dazu derartig substituiert, dass minimal viele Variablen im Restmodell verbleiben.

4.3.4.1. Motivation

Nicht immer ist die *SPDS*-Beschneidung aus Abschnitt 4.3.2 ab Seite 68 in der Lage, unnötige *Programmteile* zu identifizieren. Ein *SPDS* kann spezielle Variablen enthalten, welche keinen Beitrag zum Modellverhalten liefern (*Redundanz*). Solche Variablen können, wenn sie im Slicing liegen, nicht von der *SPDS*-Beschneidung beschnitten (eliminiert) werden. Dennoch kann das *SPDS* ohne diese Variablen kompakter ausgedrückt werden ohne das Modellverhalten zu beeinflussen. Das Eliminieren solcher Variablen führt zu einem kleineren *Konfigurationenraum*, zu einfacheren *Modellanalyse*n sowie zu effizienterer und weniger *Variablenauswertungen* (z.B. weniger OBDD-Operationen; siehe Ausdrucksvereinfachung in Abschnitt 4.3.1 ab Seite 45).

Problembewusstsein Moderne Hochsprachen wie Java oder C# verfügen über einen so genannten *Operandenkeller*. Er wird zur Vereinfachung des Bytecodes eingesetzt (weniger nötige Befehle, etc.) und besitzt eine statisch a priori bekannte maximale Tiefe. Dieser Operandenkeller kann daher direkt mittels lokaler Variablen im *SPDS* simuliert werden[38]. Push- und Pop-Operationen werden dann zu *SPDS*-typischen *Parallelzuweisung*en. Für einen Operandenkeller mit maximaler Tiefe z.B. 4 können dann lokale Variablen v_0, v_1, v_2, v_3 dienen, um den Operandenkeller im *SPDS* zu realisieren. Eine Operation *Push* e mit einem Ausdruck $e \in Expr$ wird dann modelliert als *Parallelzuweisung* „$v_0 = e, v_1 = v_2, v_2 = v_3$“ (analog ein Pop). I.d.R. werden im Operandenkeller des Bytecodes Ausdrücke der Größe $(0, 1)$ abgelegt, welche für Berechnungen von *Ausdrücke*n verwendet werden. Diese und weitere können aber direkt in die Berechnung von *Ausdrücke*n eingesetzt werden. Auch dadurch können Variablen überflüssig werden, welche nur zur Simulation des Operandenkellers genutzt wurden. Es werden dann einerseits weniger OBDD-*Auswertungen* nötig und andererseits weniger *Konfigurationenübergänge* gebraucht wegen überflüssiger Push- und Pop-Operationen. Bei Simulation dieses *Operandenkeller*s, aber auch bei einer automatischen *SPDS*-Generierung aus Quellcode sowie bei neu eingeführten *SPDS*-Hilfsvariablen zur *Interpretation* von *SPDS*-Erweiterungen entstehen im *SPDS* zusätzliche Variablen.
Diese zusätzlichen Variablen werden ggf. nicht benötigt. Hierzu zählen insbesondere unnötige zusätzliche *Kopieranweisungen* (engl. *Copy Chains*), welche sehr häufig bei der Simulation des

[38] so geschieht es z.B. beim Tool JMoped

Quellcode 4.13: Beispiel für die Variablenredundanzelimination.

```
int a(8),c(8);
init m;

void m() {
m1: c=1, a=2;
m2: p(c,a), c=a, a=c;
m3: p(c,a); }

void p(int e(8), int d(8)) {
int b(8);
p1: a=c, c=d;
p2: b=d, c=e;
p3: a=b, c=a/(a+d);
p4: return; }
```

Operandenkellers auftreten. Aber auch durch andere *Modelltransformationen* wie die Ausdrucksvereinfachung aus Abschnitt 4.3.1 ab Seite 45 können unnötige *SPDS-Zuweisung*en und *Kopieranweisungen* entstehen. Das *SPDS* kann aus diesen Gründen Redundanzen aufweisen, welche den *Konfigurationenraum* vergrößern und eine *Modellprüfung* erschweren.

Beispiel 4.3.5 Quellcode *4.13 zeigt ein* SPDS*'-Beispiel mit Redundanzen. Alle 5 definierten Variablen* a, b, c, d, e *besitzen den Typ* $bits(a) = \cdots = bits(e) = 8$ *und werden auch mindestens einmal verwendet. Jedoch können* Variablenverwendungen *wegen Redundanzen im Modell derartig ausgedrückt werden, dass weniger* SPDS*'-Variablen benötigt werden. So kann z.B.* b *an der Marke* $p3$ *durch* d *ersetzt werden wegen der* Zuweisung $b = d$ *an der Marke* $p2$*. Sämtliche realisierbare* Variablenbelegung*en an der Marke* $p3$ *erfüllen die Eigenschaft* $b = d$*.* b *wird nach der Ersetzung nirgends mehr lesend verwendet. Das Modellverhalten hängt demnach nicht von der Belegung von* b *ab. Daher kann* b *aus dem Modell eliminiert werden, wenn dies nicht* ϕ *beeinflusst. Der* Konfigurationenraum *wird dabei um* 8 *Bit verkleinert. Der Einfachheit halber wird die Konfigrationenraumveränderung ohne* Interpretation *der* Parallelzuweisung*en in SPDS-*Anweisungen *genauer betrachtet. Im Details gibt es bei* 2 *globalen Variablen mit Typ* 8 *insgesamt* 65536 *globale Situationen des zu Grunde liegenden* PDS *zur Repräsentation der globalen* Variablenbelegung*en für die Variablen* a *und* c*. Die* 3 *Marken* $m1, m2$ *und* $m3$ *dienen als* Konfigurationenübergang *(davor und dahinter jeweils eine Konfiguration) und vervielfachen diese zu* $4 \cdot 65536 = 262144$ *Köpfe für Prozedur* m*. Hinzu kommen für die* 3 *lokalen Variablen mit Typ* 8 *insgesamt* 16777216 *Kellersymbole zur Repräsentation der* Variablenbelegung*en. Zusammen mit den* 4 *Marken* $p1, p2, p3, p4$ *vervielfachen diese sich zu* $65536 \cdot 5 \cdot 16777216 = 5,5 \cdot 10^{12}$ *Köpfe für Prozedur* p*. Unter Berücksichtigung, dass lediglich die Prozedur* m *die Prozedur* p *aufrufen kann, gibt es dann insgesamt* $2 \cdot 5,5 \cdot 10^{12} = 1,1 \cdot 10^{13}$ *mögliche Konfigurationen. Durch Kompression der inneren Struktur, kann die Variable* b *wie beschrieben eliminiert werden. Dies verkleinert den* Konfigurationenraum *um drei Größenordnungen auf nur noch* $4,3 \cdot 10^{10}$ *Konfigurationen.*

Im Folgenden wird gezeigt, wie derartige Redundanz allgemein für beliebige *SPDS* erkannt und anschließend eliminiert werden kann.

Lösungsidee Wie das Beispiel zeigt, wird der *Konfigurationenraum* vor allem durch die vielen möglichen *Variablenbelegung*en von *SPDS*-Variablen bestimmt. Eine Kompaktierung auf eine minimale Anzahl nötiger Variablen eliminiert (wie im Beispiel quantitativ aufgezeigt) Redundanzen besser als die Elimination von Marken. Wie im Beispiel für die Variable b gezeigt, kann dies durch Verlagerung von *SPDS-Variablenverwendungen* auf äquivalente Variablen erfolgen (Redundanzraum). Im Idealfall kann eine Variable durch eine Konstante ersetzt werden (*absolute Redundanzelimination*), wenn sie nicht bereits durch die Ausdrucksvereinfachung (siehe Abschnitt 4.3.1 ab

Seite 45) zu einer Konstanten vereinfacht wurde. Eine lesend verwendete Variable kann aber auch (wie im Beispiel geschehen) durch eine äquivalente Variable substituiert werden. Die Substitutionen werden dabei so gewählt, dass die innere Struktur des *SPDS* derart verändert (komprimiert) wird, dass nur so wenig Variablen wie möglich im Restmodell verbleiben, welches immer noch exakt das Verhalten des *SPDS* beschreibt. Eine spezielle Form einer Lebendigkeitsanalyse identifiziert dabei sog. tote SPDS-*Anweisungen* und insbesondere tote *SPDS-Zuweisung*en, um überflüssige Variablendeklaration aus der Modellbeschreibung zu entfernen.
Zusammenfassend wird eine Redundanzelimination vorgestellt, welche in folgenden drei Phasen abläuft:

1 Bestimmen des Redundanzraums

2 Komprimieren der Redundanzen

3 Eliminieren *redundant*er *SPDS*-Variablen (*Konfigurationenraum*verkleinerung).

4.3.4.2. Notwendige abstrakte Informationen

Um in **Phase 1** Variablensubstitutionen vornehmen zu können, muss bekannt sein, womit eine lesend verwendete Variable ersetzt werden kann, ohne die Semantik des *SPDS* zu ändern (Bestimmen des Redundanzraums, Phase 1). Dazu sind in einer ersten Phase *belegungsäquivalente* Ausdrücke der Größe $(0,1)$ für alle lesend verwendeten Variablen zu bestimmen.

Definition 4.3.25 (*Belegungsäquivalente* Ausdrücke τ_l^{Eq})
Die abstrakte Information $\tau_l^{Eq} : Vars \rightarrow 2^{Vars \cup \mathbb{N}'}$ *gebe die Menge der zu einer Variablen* $v \in Vars$ *an einer Marke* l *belegungsäquivalenten* Ausdrücke *der Größe* $(0,1)$ *bezüglich der realisierbaren* Variablenbelegung*en* $RENV_l$ *an (Definition 4.3.2 auf Seite 47 sowie Abschnitt 3.3.2 auf Seite 34). Im folgendem wird dafür auch kurz der Begriff* belegungsäquivalente *Ausdrücke verwendet.*

Für die **zweite Phase** werden unter den belegungsäquivalenten *Ausdrücke*n spezielle Ausdrücke ausgewählt. Diese Auswahl wird Repräsentanten genannt und soll minimal sein, um spätere Redundanzen im transformierten *SPDS* zu reduzieren (Kompression).

Definition 4.3.26 (*Repräsentanten*)
Eine Menge $R \subseteq Vars \cup \mathbb{N}'$ *als abstrakte Information eines* SPDS S *heißt* Repräsentanten *für gegebene belegungsäquivalente* Ausdrücke τ_l^{Eq}*, falls es für jede Marke* $l \in Marken$ *und jede lesend verwendete Variable* $v \in Vars$ $(used_l^{read}(v))$ *mindestens einen belegungsäquivalenten Repräsentanten gibt, d.h. (vgl. Abschnitt 4.3.2.2 auf Seite 70):*

$$\forall l \in Marken(S) : \forall v \in Vars : (used_l^{read}(v) \Rightarrow \tau_l^{Eq}(v) \cap R \neq \emptyset).$$

Durch die Wahl von Repräsentanten (gefolgt von Ersetzungen lesender *Verwendungen*) können Variablen überflüssig bzw. *redundant* werden, wie folgendes Beispiel-*SPDS* zeigt:

```
l1: x=y;
l2: if ((x>0)&&(y>0)) goto err;
```

In diesem Beispiel sind die Werte von x und y wichtig für die *Erreichbarkeitsprüfung* der Marke err. x und y sind an der Marke $l2$ belegungsäquivalente *Ausdrücke* $(y \in \tau_{l2}^{Eq}(x))$. D.h. an der Marke $l2$ gilt stets $[\![x]\!]_{env} = [\![y]\!]_{env}$ für alle $env \in RENV_{l2}$, so dass x an der Marke $l2$ durch y ersetzt werden kann, ohne den *Konfigurationenraum* oder die *Erreichbarkeitsprüfung* von err zu beeinflussen. Dadurch wird x *redundant*, war es vorher aber nicht.
In der **dritten Phase** schließlich sind für Variablen die *abstrakten* Informationen der eigentlichen Redundanz $\tau_{rdz} : Vars \rightarrow \{true, false\}$ zu bestimmen. Eine Variable $v \in Vars$ heißt *redundant* bezüglich eines *SPDS* S und einer *temporalen Formel* ϕ bzw. einer Menge von Marken $L0 \subseteq Marken(S)$, falls die Belegung von v unwichtig für die *Modellprüfung* ist.

Definition 4.3.27 (*Redundante* SPDS-*Variablen*)
Eine Variable $v \in Vars$ *heißt* redundant, *falls für jede realisierbare* Variablenbelegung $env \in RENV_l$ *an beliebiger Marke* $l \in Marken(S)$ *das* Modellprüfung*sresultat durch Wertänderung von* v *zu beliebigem* $e \in range(v)$ *nicht beeinflusst wird. Die abstrakte Information* $\tau_{rdz}(v) \in \{true, false\}$ *gibt zu einer Variablen* $v \in Vars$ *an, ob sie* redundant *ist oder nicht.*
Es ist $\tau_{rdz}(v) = true$ *gdw.* $\forall l \in Marken(S) : \forall env \in RENV_l : \forall e \in range(v) : ((env, l) \models \phi) \Leftrightarrow ((env|_v^e, l) \models \phi)$ *bzw.* $((env, l) \rightsquigarrow L0) \Leftrightarrow ((env|_v^e, l) \rightsquigarrow L0)$. *Sind die gegebene* temporale Formel ϕ *bzw. die Marken-Menge* $L0$ *aus dem Kontext klar, so wird im Folgenden kürzer verwendet:* v *ist* redundant.

Ziel der Bestimmung von Repräsentanten in Phase zwei ist dann, möglichst viele *redundante SPDS*-Variablen zu erzeugen, welche aus dem Modell eliminiert werden können. Dabei spielen tote SPDS-*Anweisungen* eine wesentliche Rolle:

Definition 4.3.28 (*Tote SPDS*-Anweisung)
Eine SPDS-Zuweisung *der Form „*$l : x = e;$*“ eines* SPDS S *heißt* tot, *wenn es von der* symbolisch*en* Konfiguration (ENV, l) *aus keine* Konfigurationenfolge *gibt, in der* x *lesend verwendet wird bevor eine* SPDS-Zuweisung *der Form „*$l' : x = e';$*“ auf tritt und wenn* x *nicht in einer gegebenen* temporalen Formel ϕ *auf tritt. Ein* SPDS-Prozeduraufruf *der Form „*$l : p(x_1, x_2, \ldots, x_n);$*“ heißt* tot, *wenn er gemäß Abschnitt 4.3.3 auf Seite 78 keinen* Seiteneffekt *besitzt. Eine* SPDS-*Verzweigung der Form „*$l : if\ (e)\ goto\ l';$*“ heißt* tot, *wenn* $\forall env \in RENV_l : [\![e]\!]_{env} = 0$. *Ein* SPDS-*Prozedurende „*$l : return;$*“ heißt* tot, *falls es nicht* erreichbar *ist.*

Obwohl es tote *SPDS*-Anweisungen im *SPDS* geben kann, sind diese u.U. eventuell nicht eliminierbar. Die Eliminierbarkeit stellt gerade im Zusammenhang mit temporalen Formeln vielmehr eine stärkere Eigenschaft dar und kann verallgemeinert auf alle *SPDS*-Modellteile wie folgt formuliert werden:

Definition 4.3.29 (Eliminierbarkeit von Teilen der *SPDS*-Beschreibung)
Sei ein SPDS S *gegeben und* $\zeta \in \{\phi, L0\}$ *eine gegebene* temporale Formel ϕ *oder gegebene auf* Erreichbarkeit *zu prüfende Marken* $L0 \subseteq Marken(S)$. *Eine SPDS*-Anweisung $l : s \in Stats(S)$ *an der Marke* $l \in Marken(S)$ *heißt* eliminierbar[39] *bezüglich* ζ *(in Zeichen* $elim_\zeta(l : s) \in \{true, false\}$[40]*), wenn die* Modelltransformation T_{elim} :

$$\frac{l \in Marken(S) \quad l : s; \in Stats(S)}{l : s; \quad \Rrightarrow \quad l : skip;}$$

das Modellprüfung*sresultat nicht beeinträchtigt. D.h. wenn gilt:*

$$(S \models \phi) \Leftrightarrow (T_{elim}(S) \models \phi)$$

bzw.

$$(S \rightsquigarrow L0) \Leftrightarrow (T_{elim}(S) \rightsquigarrow L0).$$

Eine SPDS-*Variable* $v \in Vars(S)$ *heißt* eliminierbar *(in Zeichen* $elim_\zeta(v) = true$*, wenn sie nur in eliminierbaren SPDS*-Anweisungen *verwendet wird und nicht in* ϕ *vorkommt*[41] *(*$v \notin \phi$*):*

$$elim_\phi(v) := (v \notin \phi) \wedge \bigwedge_{\substack{l:s \in Stats(S) \\ used_l(v)}} elim_\phi(l : s)$$

Analoges gilt für L0:

$$elim_{L0}(v) := \bigwedge_{\substack{l:s \in Stats(S) \\ used_l(v)}} elim_{L0}(l : s). \tag{4.20}$$

[39] Speziell für *Prozeduraufrufe* wurde hierzu in Abschnitt 4.3.3 bereits der Begriff überbrückbar eingeführt.
[40] d.h. $elim_\phi(l : s) \in \{true, false\}$ bzw. $elim_{L0}(l : s) \in \{true, false\}$
[41] nur falls auch eine *temporale Formel* gegeben ist

Eine Prozedur $m \in Prz(S)$ *heißt* eliminierbar *(in Zeichen* $elim_\zeta(m) = true$*), wenn jeder* Prozeduraufruf *an* m *eliminierbar bzw. überbrückbar ist,* m *nicht die Hauptprozedur ist und* m *nicht in der* temporalen Formel ϕ *verwendet wird:*

$$elim_\phi(m) := (m \notin \phi) \wedge (m \neq init) \wedge \bigwedge_{l:m(...);\in Stats(S)} elim_\phi(l : m(...))$$

Analoges gilt für L0:

$$elim_\phi(L0) := (m \neq init) \wedge \bigwedge_{l:m(...);\in Stats(S)} elim_{L0}(l : m(...)) \tag{4.21}$$

Dabei kann das Eliminieren ohne Berücksichtigung des *Modellprüfung*sresultats einen *Seiteneffekt* besitzen, welcher im Rahmen der *Modellprüfung* aber i.d.R. zu berücksichtigen ist. Es könnte u.U. zu zusätzlichen False Negatives kommen. Im Beispiel 4.13 auf Seite 91 an der Marke $p3$ könnte der Ausdruck $a + b$ den Wert 0 annehmen ($\exists env \in RENV^*_{p3} : [\![a + b]\!]_{env} = 0$), was zur Division mit 0 und damit zur Versackung dieses Kontrollflusses führt, weil es dann keine *Folgekonfiguration* gibt. Versackung τ_{stop} für SPDS-*Anweisungen* wurde in Abschnitt 4.3.3 auf Seite 78 in Definition 4.3.19 auf Seite 80 erklärt und wird hier nun wieder benötigt. Es verändert sich die *Erreichbarkeit* im *SPDS*, wenn versackende SPDS-*Anweisungen* durch nicht versackende SPDS-*Anweisungen* ersetzt werden. Analoges gilt für temporale Invarianzen. Wird demnach Versackung nicht beachtet, ändert sich neben der *Erreichbarkeit* im *SPDS* ggf. auch der *Wahrheitsgehalt* der gegebenen *temporalen Formel* ϕ. Diese abstrakte Information ist insbesondere für solche SPDS-*Anweisungen* wichtig, in denen *redundante* Variablen verwendet werden, die potentiell eliminiert werden.

Eigenschaften Für gängige Hochsprachen ist es unentscheidbar festzustellen, ob zwei Variablen $x, y \in Vars$ an einem *Programmpunkt* stets den gleichen Wert haben (Reduktion Postsches Korrespondenzproblem [91]). Daher sind belegungsäquivalente *Ausdrücke* (Definition 4.3.25 auf Seite 92) ebenso unentscheidbar für gängige Hochsprachen. Wie bereits für die untersuchte Äquivalenzanalyse mit Satz 4.3.6 auf Seite 63 gezeigt wurde, ist dieses Problem für *SPDS* entscheidbar, da die Äquivalenzanalyse exakt durchgeführt werden kann.

Satz 4.3.17 (Komplexität belegungsäquivalenter *Ausdrücke*)
Die Berechnung exakter belegungsäquivalenter Ausdrücke *ist mindestens so aufwändig wie die* Modellprüfung.

Beweis(Skizze) Die Behauptung wird durch Reduktion der *Modellprüfung* auf die Berechnung exakter belegungsäquivalenter *Ausdrücke* gezeigt. Sei ein *SPDS* S und eine auf *Erreichbarkeit* zu prüfende Marke $l_0 \in Marken(S)$ gegeben. Die *Modellprüfung* hat zu bestimmen, ob gilt $S \rightsquigarrow l_0$. Nun wird einen Spezialfall für belegungsäquivalente *Ausdrücke* betrachtet, ob eine Variable v an einer Marke l stets den Wert 0 hat oder nicht. Dazu wird eine neue globale (redundante) Variable $v \notin Vars(S)$ im *SPDS* S mit Typ $bits(v) = 1$ eingeführt, die per Definition beliebige Werte (0 oder 1) annehmen darf. Die SPDS-*Anweisung* an der Marke l_0 ist unwichtig für die *Erreichbarkeitsprüfung* $S \rightsquigarrow l_0$ (man betrachte O.B.d.A. nur Läufe, in denen l_0 einmalig am Ende auftritt). Daher kann die SPDS-*Anweisung* an der Marke l_0 zu „$l_0 : v = 0$;“ geändert werden, ohne den *Wahrheitsgehalt* von $S \rightsquigarrow l_0$ zu beeinträchtigen. Eingeführt wird weiter eine neue Marke $l \notin Marken(S)$ als Nachfolgemarke von l_0, so dass $(l_0, l) \in CFG$ und l nur l_0 zum intraprozeduralen Vorgänger hat. Nun werden in diesem geänderten *SPDS* exakte belegungsäquivalente *Ausdrücke* R_l bestimmt, was Aufschluss darüber gibt, ob v an der Marke l stets den Wert 0 hat oder nicht ($(v, 0) \in R_l$). Da R_l exakt ist, gilt nach Konstruktion $(v, 0) \in R_l \Leftrightarrow S \rightsquigarrow l_0$. Denn wenn $(v, 0) \in R_l$ ist, dann muss es einen Lauf über l_0 zu l geben (einziger Vorgänger von l), so dass für v der Wert 0 angenommen werden kann. Andererseits kann die Äquivalenz $(v, 0) \in R_l$ nur dann nicht realisiert werden, wenn es keinen Lauf zu l (über l_0) gibt. Somit löst die Berechnung exakter belegungsäquivalenter *Ausdrücke* das beliebig gegebene *Modellprüfung*sproblem $S \rightsquigarrow l_0$.

□

Zur Beschleunigung der *Modellprüfung* können somit exakte belegungsäquivalente *Ausdrücke* nicht genutzt werden. Als *Modellanalyse*beispiel wird daher - wie in früheren und weiteren Fällen auch - eine *konservative* Approximation verwendet.

Satz 4.3.18 (Entscheidbarkeit der Variablenredundanz)
Es ist entscheidbar (exakt berechenbar), ob eine Variable redundant *ist oder nicht.*

Beweis(Skizze) Sei eine Variable $v \in Vars$ eines *SPDS* S beliebig aber fest gewählt. Wegen $|Marken(S)| < \infty, |range(v)| < \infty$ und $|RENV_l| < \infty$ für beliebiges $l \in Marken(S)$ kann die *Modellprüfung* endlich oft für die *symbolisch*en *Konfigurationen* (env, l) bzw. $(env|_v^e, l)$ für alle $env \in RENV_l$ und alle $e \in range(v)$ durchgeführt werden. Führen alle *Modellprüfung*en zum gleichen Resultat (erfüllen ϕ oder nicht bzw. $L0$ *erreichbar* oder nicht), so ist durch Wertänderung von v keine Änderung des *Modellprüfung*sresultats feststellbar. D.h. v ist dann *redundant*. Ändert sich jedoch andererseits das *Modellprüfung*sresultat für mindestens ein $l \in Marken$ und ein $e \in range(v)$, so ist v nicht *redundant*.

□

Satz 4.3.19 (Komplexität exakter Variablenredundanz)
Die Berechnung exakter Redundanz von SPDS-*Variablen ist mindestens so aufwändig wie die* Modellprüfung.

Beweis(Skizze) Gezeigt wird die Behauptung wieder analog zum Beweis von Satz 4.3.17 durch Reduktion der *Modellprüfung* auf die Berechnung exakter Redundanz von *SPDS*-Variablen. Sei ein *SPDS* S und eine auf *Erreichbarkeit* zu prüfende Marke l_0 gegeben. Die *Modellprüfung* hat zu bestimmen, ob gilt $S \rightsquigarrow l_0$. Es sei eine neue globale Variable $v \notin Vars(S)$ im *SPDS* S mit Typ $bits(v) = 1$ eingeführt, welche per Definition beliebige Werte (0 oder 1) annehmen darf. Die SPDS-*Anweisung* an der Marke l_0 ist unwichtig für die *Erreichbarkeitsprüfung* $S \rightsquigarrow l_0$ (man betrachte O.B.d.A. nur Läufe, in denen l_0 nur einmalig am Ende auftritt). Daher kann die SPDS-*Anweisung* an der Marke l_0 zu „$l_0 : if\ (v > 0)\ skip\ false$;“ geändert werden, ohne den *Wahrheitsgehalt* von $S \rightsquigarrow l_0$ zu beeinträchtigen. Sei weiter eine neue Marke $l \notin Marken(S)$ als Nachfolgemarke von l_0 eingeführt, so dass $(l_0, l) \in CFG$ und l nur l_0 zum intraprozeduralen Vorgänger hat. Der Wert von v entscheidet dann über die *Erreichbarkeit* der Marke l. Daher kann v nicht *redundant* sein, wenn l_0 *erreichbar* ist. Nun wird in diesem geändertem *SPDS* S' die exakte Redundanz der Variablen v (welche nur an der Marke l_0 verwendet wird) bestimmt. Ist v *redundant* für die *Erreichbarkeitsprüfung* $S' \rightsquigarrow l$, so ist l_0 in S nicht erreichbar. Ist andererseits v nicht *redundant* für die *Erreichbarkeitsprüfung* $S' \rightsquigarrow l$, so ist l_0 in S erreichbar. Zusammen ergibt dies die Äquivalenz: $\tau_{rdz}(v) = true$ für die *Erreichbarkeitsprüfung* $S' \rightsquigarrow l$ gdw. $S \rightsquigarrow l_0$. Somit löst die Berechnung exakter Redundanz von *SPDS*-Variablen das beliebig gegebene *Modellprüfung*sproblem $S \rightsquigarrow l_0$.

□

Damit ist auch für die Variablenredundanz eine *konservative* Approximation nötig.

Satz 4.3.20 (Entscheidbarkeit der Eliminierbarkeit)
Es ist entscheidbar (exakt berechenbar), ob Teile der Modellbeschreibung (Definition 4.3.29 auf Seite 93) eliminierbar sind oder nicht.

Beweis(Skizze) Führe die *Modellprüfung* einmal mit und einmal ohne eliminierten Teil der Modellbeschreibung aus. Ändert sich das Modellprüfungsresultat, so ist dieser Teil der Modellbeschreibung nicht eliminierbar.

□

Wie auch die exakte Variablenredundanz ist die Eliminierbarkeit sehr aufwändig:

Satz 4.3.21 (Komplexität der Eliminierbarkeit)
Die Berechnung exakter Eliminierbarkeit von Teilen der Modellbeschreibung ist mindestens so aufwändig wie die Modellprüfung.

Abbildung 4.8.: Reduktion vom Überdeckungsproblem auf *optimale* Repräsentantenwahl $\{m_1, m_3\}$

A	x_1	x_2	x_3	x_4
m_1	1	1		1
m_2		1		1
m_3	1		1	1
m_4			1	1

```
  # {x1=m1=m3}      use(int v(8), int p(8)) {
L1: use(x1,1);        print(v);
  # {x2=m1=m2}      xi=undef, mi=undef; # i in [1..4]
L2: use(x2,2);        if
  # {x3=m3=m4}        :: p==1 -> m1=x2,m2=x2;
L3: use(x3,3);        :: p==2 -> m3=x3,m4=x3;
  # {x4=m1=m2=m3=m4}  :: p==3 -> m1=x4,m2=x4,m3=x4,m4=x4;
L4: use(x4,4);        fi; return; }
```

Beweis(Skizze) Analog zum Beweis von Satz 4.3.19.

□

Auch die Repräsentanten können *optimal* bestimmt werden.

Definition 4.3.30 (*Optimale Variablen-Repräsentanten*)
Eine Menge $R_{var} \subseteq Vars$ *heißt* optimale *Variablen-Repräsentanten bezüglich einer (endlichen) Menge* $R_{fix} \subseteq Vars \cup \mathbb{N}'$*, falls* $R = R_{var} \cup R_{fix}$ *Repräsentanten für ein* SPDS S *und belegungsäquivalente* Ausdrücken τ_l^{Eq} *sind und es keine Menge* $R'_{var} \subseteq Vars$ *mit entsprechenden Repräsentanten* $R' = R'_{var} \cup R_{fix}$ *gibt, so dass* $|R'_{var}| < |R_{var}|$.

Satz 4.3.22 (Entscheidbarkeit *optimale*r Variablen-Repräsentanten)
Optimale *Variablen-Repräsentanten sind entscheidbar (exakt berechenbar).*

Beweis(Skizze) Da $|Vars \cup \mathbb{N}'| < \infty$, können prinzipiell alle möglichen Teilmengen von $Vars \cup \mathbb{N}'$ untersucht werden (z.B. mittels einer quasilexikographischen Aufzählung). Unter jenen Teilmengen, welche eine gültige Wahl an Repräsentanten darstellen, wird eine kleinste gewählt.

□

Die Komplexität zur Berechnung *optimale*r Variablen-Repräsentanten ist im Gegensatz zur Variablenredundanz, belegungsäquivalenter *Ausdrücke* oder Eliminierbarkeit deutlich einfacher. Insbesondere bei Interpretation von Reihungen entstehen viele einzelne Variablen im Modell, welche es einem Lösungsverfahren erschweren, optimale Repräsentanten zu wählen.

Satz 4.3.23 (Komplexität *optimale*r Variablen-Repräsentanten)
Die optimale *Variablen-Repräsentantenwahl*[42] *ist NP-hart.*

Beweis(Skizze) Die Behauptung wird gezeigt durch Reduktion des NP-vollständigen Überdeckungsproblems [200] auf die *optimale* Repräsentantenwahl. Sei eine beliebige Überdeckungsmatrix $A = (a_{ij}) \in \{0,1\}^{m,n}$ gegeben. Gesucht ist eine minimale Auswahl an Zeilen, so dass deren logisches ODER den 1-Vektor $(1, 1, 1, ..., 1)$ bilden (set cover). Es wird ein *SPDS* wie jenes in Abbildung 4.8 konstruiert. Dann wird durch die *optimale* Repräsentantenwahl für $L1..Ln$ das Überdeckungsproblem A gelöst.

□

Die Entscheidbarkeit der Versackung von Prozeduren aus Abschnitt 4.3.3 auf Seite 78 überträgt sich mit den Eigenschaften auf die Versackung τ_{stop} für SPDS-*Anweisungen*, da SPDS-*Anweisungen* in Prozeduren auftreten. τ_{stop} ist für *SPDS-Anweisungen* daher exakt berechenbar mit gleicher Komplexität wie für Prozeduren (siehe Satz 4.3.13).

[42] finden einer minimalen Auswahl an Variablen-Repräsentanten

4.3.4.3. Einsetzbare Programmanalysen

Wie im vorigen Abschnitt erläutert, sollten wegen der Komplexität gewisse *abstrakten* Informationen *konservativ* durch *Programm-* bzw. *Modellanalyse*n approximiert werden.

Belegungsäquivalente Ausdrücke Eine sehr einfache Form, äquivalente Variablen zu identifizieren, ist das Verfolgen von *Kopieranweisungen* $x = y$; für zwei Variablen $x, y \in Vars$ entlang des Kontroll- bzw. Datenflusses (Copy Propagation [221]). Für *SPDS* sind jedoch *präzise Modellanalyse*n besser, da diese potentiell den *Konfigurationenraum* besser analysieren und damit mehr Redundanzen finden. Damit wird der *Konfigurationenraum* stärker verkleinert und die aufwändige *Modellprüfung* besser beschleunigt. Zum Finden von belegungsäquivalenten *Ausdrücke*n (der Größe $(0, 1)$) kann daher auch die bereits im Abschnitt 4.3.1.4 auf Seite 56 in Definition 4.3.11 auf Seite 58 erläuterte Äquivalenzanalyse eingesetzt werden. Die Ergebnisse dieser *Modellanalyse* sind dann ähnlich zu Zeigeranalysen wie z.B. [142, 138]. Zeiger auf Objekte gibt es allerdings in *SPDS* nicht. Zeiger können simuliert werden durch *SPDS*-Variablen, welche in den globalen Speicher verweisen. Der globale Speicher kann in *SPDS* als eine große *Reihung heap* konstruiert werden, so dass Zeiger aus der Hochsprache auf einen Index (Startposition eines Objekts in der *Reihung heap*) verweisen. Auf diese Weise wird die in Definition 4.3.11 auf Seite 58 erläuterte Äquivalenzanalyse zu einer Zeigeranalyse. Die vorgestellte Äquivalenzanalyse ist in diesem Sinn daher eine Erweiterung von Zeigeranalysen, welche ebenfalls zum Ziel haben, äquivalente Variablen bzw. Zeiger zu identifizieren. Der Zusammenhang zu Zeigeranalysen wird deutlicher, wenn (wie in C++) Zeigerarithmetik in der Hochsprache erlaubt ist. Die in dieser Arbeit definierte Äquivalenzanalyse unterteilt sich jedoch nicht wie z.B. [142] in zwei Phasen, weswegen die Analyseergebnisse durch entsprechende Fixpunktberechnung in diesem Fall besser über Prozedurgrenzen hinweg gelangen. Die vorgestellte Äquivalenzanalyse ist zudem auch nicht (wie [142]) auf Zeiger beschränkt, sondern erlaubt auch Zeigerarithmetik (Rückinterpretation von *SPDS*-Variablen). Zeigeranalysen in Hochsprachen basieren entweder auf Kontroll- bzw. Datenflussanalysen (z.B. [13]), Typsystemen (z.B. [216, 141]) oder Partitionierungsverfahren (z.B. [74]). Die Analysen basierend auf Kontroll- und Datenfluss (wie auch die definierte Äquivalenzanalyse in Definition 4.3.11) liefern i.d.R. *präzise*re Analyseergebnisse [112].

Repräsentantenwahl Die Repräsentantenwahl (Phase zwei) besteht aus einem Optimierungsproblem (Wahl geeigneter Repräsentanten). Es ist vergleichbar zur Registerzuteilung bei Übersetzern, wenn hinreichend viele Register zur Verfügung stehen (ohne Auslagerung von Werten) [221, 134, 98, 46, 38]. Aufgrund der Komplexität der Repräsentantenwahl (insbesondere bei vielen *SPDS*-Variablen z.B. *Interpretation* von *Reihungen*, siehe Satz 4.3.23) können die Repräsentanten heuristisch gewählt werden. Es können prinzipiell die Techniken zur Registerzuteilung für *SPDS* angepasst werden, indem zur Minimierung des Registerdrucks entsprechend bekannte Heuristiken genutzt werden (z.B. die Heuristik von Poletto [186] oder Belady [103] oder Graphfärbungen nach Briggs [38], etc). Als Register werden dann *SPDS*-Variablen verwendet. Allerdings führen diese Heuristiken für *SPDS* nicht immer zu einer guten Repräsentantenwahl. Denn sie betrachten i.d.R. nur die lokale Registerzuteilung, so dass sich insgesamt möglicherweise ein größerer *Konfigurationenraum* für das *SPDS* ergibt, als wenn in einigen Fällen lokal eben nicht die *optimale* Registerzuteilung gewählt wird. Werden die Repräsentanten R z.B. trivial als $R = Vars$ gewählt (da genügend Register zur Verfügung stehen), so entstehen dadurch keine weiteren Redundanzen, welche in der Folgephase eliminiert werden können. In einer eigenen Heuristik könnten z.B. diejenigen belegungsäquivalenten Variablen als Repräsentanten gewählt werden, welche besonders häufig in Äquivalenzklassen auftreten (abhängig von den Marken), da diese eine besonders hohe Wahrscheinlichkeit besitzen, global andere Variablen zu ersetzen bzw. abzudecken. So kann durch eine Heuristik auf den Variablen eine totale Ordnung bzw. Priorität definiert werden, um zu garantieren, dass aus einer Menge von äquivalenten Variablen stets ein kleinstes Element exisitert, welches als eindeutiger Repräsentant dieser Menge verwendet werden kann. Mit folgender heuristischer Priorität könnten die Äquivalenzinformationen geordnet werden, um später möglichst gute *Modelltransformationen* durchzuführen, indem möglichst viele *SPDS*-Variablen *redundant* werden:

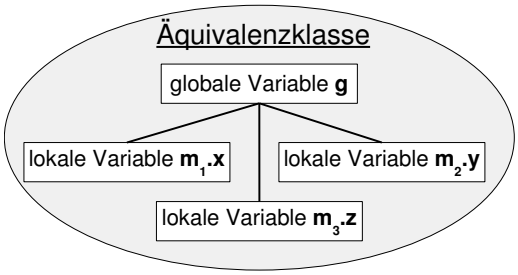

Abbildung 4.9.: Beispiel von Äquivalenzen bei verschiedenen Konfigurationen

1 ... Konstanten erhalten die höchste Priorität, falls möglich[43].

2 ... dann die kleinste Parametervariable aus der zugehörigen Prozedur, falls möglich.

3 ... dann die kleinste globale Variable, falls möglich.

4 ... dann die kleinste lokale Variable (Spätestens dies ist dann immer möglich, da es keine andere Art von Variablen gibt).

Die Begründung der Reihenfolge ist folgende. Es ist für den *Konfigurationenraum* besser, Konstanten statt Variablen zu verwenden, da überflüssige Variablen unnötigerweise den *Konfigurationenraum* vergrößern. Parametervariablen besitzen nur Priorität 2, da sie sich nicht vermeiden lassen, ohne die Signatur zu ändern. Zudem (so die Hoffnung) werden möglichst viele *Kettenzuweisung*en ausgehend von Prozedurparametern auf diese Weise vermieden. Gibt es nicht die Möglichkeit, eine Konstante oder einen Prozedurparamter als Repräsentanten zu wählen, so sollte lieber versucht werden eine globale Variable statt einer lokalen Variable als Repräsentant zu wählen. Der Grund hierfür ist verdeutlicht in Abbildung 4.9, dass mehrere Prozeduren die globalen Variablen wiederverwenden können und sich so den *Konfigurationenraum* teilen. Prozedur m_1 kann nicht auf die äquivalente Variable einer anderen Prozedur (z.B. m_2) zugreifen. Daher gibt es in der Abbildung auch keine Verbindung dazwischen. Im Idealfall werden auf diese Weise sogar alle Variablen x, y, z aus Abbildung 4.9 *redundant*. Dann ist lediglich die globale Variable g nötig. Werden andererseits als Repräsentanten x, y, z gewählt, so wird ggf. nur g *redundant*. Der *Konfigurationenraum* verkleinert sich dann lediglich um eine Variable statt um drei Variablen.
Andererseits können die Repräsentanten aber auch dank effizienter Lösungswerkzeuge für Lineare *Programme optimal* gewählt werden, sollte die Anzahl der Repräsentanten eine gewisse vorgegebene Grenze nicht übersteigen (experimentell bestimmbar).

Redundante Variablen und Eliminierbarkeit Wie gezeigt, ist es sehr aufwändig, *redundant*e Variablen gemäß Definition 4.3.27 auf Seite 93 oder die Eliminierbarkeit gemäß Definition 4.3.29 auf Seite 93 zu bestimmen. Daher werden diese Eigenschaften durch *konservative Modellanalyse*n approximiert, um die *Modellprüfung* zu beschleunigen. D.h. es werden hinreichende Eigenschaften genutzt, um *redundant*e Variablen und eliminierbare Teile der Modellbeschreibung zu identifizieren. Z.B. sind tote Variablen [221] bereits *redundant*, da deren *Variablenbelegungen* keinen Beitrag zum prinzipiellen Modellverhalten leisten, wenn sie nicht in einer gegebenen *temporalen Formel* verwendet werden oder das Modell anders beeinflussen. Ähnliches gilt für toten Code [221]. Muchnick [221, S. 443 ff] definiert eine Variable x an einem *Programmpunkt* p als lebendig (Gegenteil von tot), wenn es von p aus einen Pfad zu einem *Finalprogrammpunkt* gibt, auf welchem x verwendet wird ohne dass x bis zu dieser *Verwendung* ein neuer Wert zugewiesen wird. Diese Form von Lebendigkeit muss für *SPDS* entsprechend auf *Konfigurationen* übertragen und auf *temporale Formel*n erweitert werden. Es scheint zunächst klar, dass eine tote Variable $x \in Vars$ im Punkt p

[43] Wegen der Unabhängigkeit der eingesetzten Modellanalyse zählen hierzu insbesondere auch jene Konstanten, welche nicht durch eine Konstanten-Propagation oder -Faltung identifiziert werden können.

bzw. an einer Marke l aus dem Modell entfernt werden kann, wenn sie nicht in einer *temporalen Formel* verwendet wird. Dies ist nicht immer der Fall. Es sind für *SPDS* weitere Anpassungen bzw. die Berücksichtigung weiterer Nebenbedingungen wichtig, wie bei der Division mit 0 am Beispiel 4.3.5 auf Seite 91 gezeigt, welche von der Redundanz zur Eliminierbarkeit schließen lassen.

Versackende Anweisungen Zum *konservativen* Erkennen versackender SPDS-*Anweisungen* können die gleichen Modell- und *Programmanalysen* wie bei der Prozedurüberbrückung im Abschnitt 4.3.3 auf Seite 78 verwendet werden.

4.3.4.4. Modellanalysebeispiele

Nun werden ausgewählte Techniken der vorgeschlagenen für eine Realisierung näher betrachtet.

Idee Zur Bestimmung von τ_l^{Eq} (belegungsäquivalente *Ausdrücke* der Größe $(0, 1)$) wird die bereits im Abschnitt 4.3.1.4 auf Seite 56 in Definition 4.3.11 auf Seite 58 erläuterte Äquivalenzanalyse eingesetzt. Aus der berechneten Äquivalenzrelation Eq_l sind dann nur noch die entsprechenden Äquivalenzklassen für τ_l^{Eq} zu extrahieren.
Die Repräsentanten sollten in der zweiten Phase derart gewählt werden, dass möglichst viele Variablen *redundant* werden und aus dem Modell eliminiert werden können. Da die Komplexität der *optimal*en Repräsentantenwahl bei weitem nicht so groß ist, wie das Erkennen von Redundanten Variablen, werden in dieser Phase die Repräsentanten *optimal* bestimmt mittels des ILP[44]-Lösers *GLPK* (GNU Linear *Programming* Kit [145]). Auf eine Heuristik wird an dieser Stelle verzichtet, da *optimal*e Repräsentanten die *Modellprüfung* besser beschleunigen und der ILP-Löser GLPK die auftretenden ILP-Probleme hinreichend effizient löst.
Um *redundant*e Variablen zu identifizieren, werden mittels Kontroll- und Datenflussanalysen tote Variablen gesucht, welche zusätzlich nicht in einer gegebenen *temporalen Formel* verwendet werden. Mit Hilfe einer *Seiteneffektanalyse* (vgl. Abschnitt 4.3.3 auf Seite 78) bzw. der *Wertebereichsanalyse* aus Definition 4.3.5 auf Seite 53 können Variablen aus dem *SPDS* eliminiert werden, ohne das Ergebnis der *Modellprüfung* zu verändern. Wird eine tote *Zuweisung* aus dem *SPDS* eliminiert, so beeinflusst dies potentiell die *Erreichbarkeit*, wenn diese SPDS-*Anweisung* z.B. versackend ist (analog zu versackenden Prozeduren in Definition 4.3.20 auf Seite 80). Eliminierbare SPDS-*Anweisungen* dürfen daher nicht versacken. Andernfalls wird es (wie am Beispiel erläutert) nach der Elimination der potentiell versackenden SPDS-*Anweisung* mehr Modellpfade geben als vorher und damit potentiell auch neue Fehlalarme (False Negatives). Dies ist aber nach Zielsetzung der Dissertation nicht erwünscht.

Konkretisierung

Lemma 4.3.17 (*Konservative* Belegungsäquivalenz τ_l^{Eq})
Seien Eq_l eine Äquivalenzrelation einer Äquivalenzanalyse gemäß Definition 4.3.11 auf Seite 58 und $x \in Vars$ eines SPDS *S. Dann sind*

$$\tau_l^{Eq}(x) := \{y \mid (x, y) \in Eq_l\} \tag{4.22}$$

belegungsäquivalente Ausdrücke *für die Variable x (gemäß Definition 4.3.25 auf Seite 92).*

Beweis(Skizze) Folgerung aus der *Konservativität* der Äquivalenzanalyse.

□

Die Wahl der Repräsentanten R verläuft in zwei Phasen. In der ersten Phase werden Repräsentanten $R_{const} \subset \mathbb{N}'$ und R_ϕ gewählt und in der zweiten Phase Repräsentanten $R_{var} \subset Vars$ bestimmt, so dass $R = R_{const} \cup R_\phi \cup R_{var}$. Ist keine *temporale Formel* ϕ gegeben für die *Modellprüfung*, so ist $R_\phi = \emptyset$. Im Folgendem sei $R_{fix} := R_{const} \cup R_\phi$. Wie bereits bei der Ausdrucksvereinfachung

[44] integer linear program

im Abschnitt 4.3.1 auf Seite 45 und bei den einsetzbaren *Programmanalysen* zur Repräsentantenwahl erläutert, ist es für das *SPDS* besser, wenn lesende *Verwendungen* von Variablen komplett entfallen, statt sie durch andere zu ersetzten. Dies gelingt z.B. indem Konstanten als Repräsentanten gewählt werden. Daher werden in der ersten Phase zuerst so viele Konstanten wie möglich als Repräsentanten gewählt, da dies zusätzlich das Lineare *Programm* verkleinert. Damit für eine lesende *Verwendung* einer Variable $v \in Vars$ an der Marke $l \in Marken$ ($used_l^{read}(v) = true$) eine Konstante als Repräsentant gewählt werden kann, muss v an der Marke l belegungsäquivalent zu einer Konstanten $c \in \mathbb{N}'$ sein, d.h. $c \in \tau_l^{Eq}(v)$. Da $\tau_l^{Eq}(v)$ eine Äquivalenzklasse der Äquivalenzrelation Eq_l ist, kann diese nur über höchstens eine Konstante verfügen, da verschiedene Konstanten untereinander nicht äquivalent sind. D.h. es gilt $\tau_l^{Eq}(v) \cap \mathbb{N}' \leq 1$. Für den Fall $\tau_l^{Eq}(v) \cap \mathbb{N}' = \{c\}$ wird dann c als Repräsentant gewählt. Für den Fall $\tau_l^{Eq}(v) \cap \mathbb{N}' = \emptyset$ wird keine Konstante als Repräsentant gewählt, da die verwendete Äquivalenzanalyse für die Variable v an der Marke l keine belegungsäquivalente Konstante identifiziert hat. Dann werden Variablen v als Repräsentanten gewählt, welche in einer ggf. gegebenen *temporalen Formel* ϕ vorkommen, d.h. $R_\phi = \{v \mid v \in \phi\}$. Dann wird in Phase zwei ein ILP gelöst. Dazu wird folgendes Ganzahlige Lineare *Programm* (ILP) konstruiert:

Definition 4.3.31 (*Exakte Variablen-Repräsentanten* R_{var})
Sei die Menge der noch nicht ausdrückbaren lesenden Variablenverwendungen *in einem* SPDS S *gegeben als*

$$reads(S) := \{L_x \mid L \in Marken(S), x \in Vars(S) \setminus R_{fix}, used_L^{read}(x)\}.$$

ILP-Variablen $\alpha_v \in \{0, 1\}$ *mit* $v \in Vars$ *werden gewählt, so dass* $\alpha_v = 1 \Leftrightarrow v \in R_{var}$. *Dann ist für das ILP die Anzahl der Repräsentanten zu minimieren:*

$$\sum_{v \in Vars} \alpha_v \rightarrow min,$$

mit der Nebenbedingung, dass jede Variablenverwendung $L_x \in reads(S)$ *durch mindestens einen Variablen-Repräsentanten* v *(mit* $\alpha_v = 1$*) ausgedrückt werden kann:*

$$\forall L_x \in reads(S) : \sum_{v \in \tau_L^{Eq}(x) \cap Vars} \alpha_v \geq 1.$$

Die Lösung des ILPs (Belegung der α_v*) bestimmt dann die Menge der Variablenrepräsentanten*[45] R_{var}.

Dieses ILP wird dann durch den ILP-Löser GLPK gelöst, welcher die global minimale Anzahl an Variablen-Repräsentanten R_{var} liefert, mit denen alle lesenden *Variablenverwendungen* ausgedrückt werden können. Aus den möglichen Repräsentanten R einer Variable v an einer Marke l kann ein beliebiger ausgewählt werden, da für Repräsentanten stets gilt: $\tau_l^{Eq}(v) \cap R \neq \emptyset$.

Definition 4.3.32 (*Repräsentantenwahl* τ_l^R)
Die Repräsentantenwahl $\tau_l^R : Vars(S) \rightarrow \mathbb{N}' \cup Vars(S)$ *ist eine beliebige Funktion (Zuordnung der Variablen zu Repräsentanten), so dass gilt:* $\forall l \in Marken(S) : \forall v \in Vars(S) : (used_l^{read}(v) \Rightarrow \tau_l^R(v) \in R)$.

Sind die Repräsentanten bestimmt, werden Redundanzen von *SPDS*-Variablen gesucht. Es ist klar, dass nie verwendete Variablen automatisch *redundant* sind. Allerdings kann es SPDS-*Anweisungen* geben, welche *redundante* Variablen verwenden. Eine *Zuweisung* an eine redundante Variable z.B. ist dann ebenso *redundant*. Vergleichbare Eigenschaften sind für Höhere *Programmiersprachen* als Lebendigkeit bzw. toter *Quellcode* bekannt [221]. Der Unterschied von totem *Quellcode* zur Redundanz besteht in der Einflussnahme auf die *Modellprüfung*. So berücksichtigt einerseits toter Code keine temporalen Bedingungen, was zu *False Positives* führen kann. Andererseits können aber auch

[45] vgl. min set cover [45]

nicht tote Teile der Modellbeschreibung *redundant* sein, wenn diese keinen Einfluss auf die *Modellprüfung* hat. Wie tote bzw. *redundante* bzw. *Seiteneffekt* freie *SPDS-Prozeduraufrufe* überbrückt werden können, wurde bereits in Abschnitt 4.3.3 auf Seite 78 gezeigt. Ein totes bzw. *redundantes* *SPDS*-Prozedurende wird mittels Slicing aus Abschnitt 4.3.2 auf Seite 68 erkannt. An dieser Stelle wird daher nicht näher darauf eingegangen, wie *SPDS-Prozeduraufrufe* und *SPDS*-Prozedurenden eliminiert werden können. Zur weiteren Untersuchung werden daher nur *SPDS-Zuweisungen* und *SPDS*-Verzweigungen näher betrachtet. Tote *SPDS*-Verzweigungen werden mittels der in Abschnitt 4.3.1 auf Seite 45 erläuterten Ausdrucksvereinfachung (u.a. unter *Verwendung* der *Wertebereichsanalyse* aus Definition 4.3.5 auf Seite 53 mittels Lemma 4.3.1 auf Seite 53) bestimmt[46]. Sie können dann im Rahmen einer Verzweigungsreduktion aus dem *SPDS* eliminiert werden, da die Bedingung e in jeder realisierbaren *Variablenbelegung* unerfüllbar ist. Es ist viel wichtiger für den *Konfigurationenraum*, Variablen zu eliminieren. Dazu wurde bisher erläutert, wie lesende *Verwendungen* durch Repräsentanten ausgedrückt werden können. Schreibend werden diese Variablen allerdings immer noch verwendet in Form von *SPDS-Zuweisungen*. Diese *SPDS*-Zuweisungen können tot sein oder tot werden. Tote *SPDS-Zuweisungen* könnten dann über den Kontrollfluss mittels *konservativ* approximierter *Variablenbelegungen* identifiziert werden[47]. Ist irgend eine *SPDS-Zuweisung* „$l : x = e$;“ im *SPDS* S nicht tot, so muss die Redundanz der *SPDS*-Variablen $x \in Vars(S)$ mittels zusätzlicher *Modellanalysen* überprüft werden, wenn x aus dem *SPDS* eliminiert werden soll. Die Variable x ist aber andererseits bereits *redundant*, wenn alle *SPDS-Zuweisungen* an die Variable x tot sind und wenn sie nicht in einer gegebenen *temporalen Formel* ϕ verwendet wird. Tot sind alle *SPDS-Zuweisungen* an x dann, wenn x nicht lesend im *SPDS* verwendet wird. Dies wird als hinreichende Bedingung für Redundanz verwendet:

Lemma 4.3.18 (*Konservative* Redundanz τ_{rdz})
Eine Variable $v \in Vars$ ist redundant *bezüglich eines* SPDS S *und einer gegebenen* temporalen Formel ϕ *bzw. bezüglich einer gegebenen Menge an Marken $L0 \subseteq Marken(S)$, falls v in S nie* ***lesend*** *verwendet wird und v nicht in ϕ verwendet wird:*

$$\tau_{rdz}(v) := v \notin \phi \wedge \bigwedge_{l \in Marken} \neg used_l^{read}(v).$$

Beweis(Skizze) Sei eine nie lesend verwendete Variable v ($\forall l \in Marken(S) : used_l^{read}(v) = false$) gegeben, so dass $v \notin \phi$. Da v nie lesend verwendet wird, sind alle *SPDS-Zuweisungen* der Form „$l : v = e$“ nach Definition 4.3.28 auf Seite 93 tot. Wegen $v \notin \phi$ sind diese dann auch *redundant*. Daher sind in S alle *SPDS-Zuweisungen* an v redundant, weswegen v selbst redundant ist.

□

Entsprechend kann es Variablen geben, welche zwar *redundant* sind, dies aber nicht erkannt wird durch das *konservative* τ_{rdz}. Ebenso *konservativ* wird die Versackung τ_{stop} von SPDS-*Anweisungen* bestimmt.

Lemma 4.3.19 (*Konservative* Versackungsfreiheit τ_{stop})
Eine SPDS-Anweisung $l : z \in Stats(S)$ *eines* SPDS S *ist für beliebige* Variablenbelegungen $W \subseteq ENV$ *versackungsfrei $\tau_{stop}(W, z) = false$, wenn z keinen arithmetischen Überlauf und keine Division mit* 0 *sowie keinen Zyklus*[48] *enthält.*

Der Beweis findet sich im Anhang B ab Seite 145. Analog zu Prozeduren in Abschnitt 4.3.3 ab Seite 78 werden mögliche arithmetische Überläufe und Divisionen mit 0 *konservativ* mittels *Wertebereichsanalyse* ermittelt (siehe Lemma 4.3.15 auf Seite 84). Die Eliminierbarkeit wird in der Modellreduktion *konservativ* mittels der konservativen Redundanz τ_{rdz} und der konservativen Versackungsfreiheit τ_{stop} bestimmt.

[46] Eine *SPDS*-Verzweigung ist z.B. bereits dann tot, wenn $\forall env \in RENV_l^* : [\![e]\!]_{env} = 0$ (Folgerung aus $RENV_l \subseteq RENV_l^*$).
[47] Z.B. Überapproximation realisierbarer Pfade mittels sämtlicher möglicher Kontrollflusspfade
[48] $l : z$ enthält einen Zyklus, wenn z.B. z ein Sprung an l enthält oder l die Anfangsmarke einer Prozedur q ist und z ein *Prozeduraufruf* an q

Tabelle 4.1.: Belegungsäquivalenz τ_l^{Eq} für Beispiel 4.3.5

τ_l^{Eq}	a	b	c	d	e
$m1$	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$
$m2$	$\{2, a\}$	$\{b\}$	$\{1, c\}$	$\{d\}$	$\{e\}$
$m3$	$\{1, a\}$	$\{b\}$	$\{2, c\}$	$\{d\}$	$\{e\}$
$p1$	$\{a, d\}$	$\{b\}$	$\{c, e\}$	$\{a, d\}$	$\{c, e\}$
$p2$	$\{a, e\}$	$\{b\}$	$\{c, d\}$	$\{c, d\}$	$\{a, e\}$
$p3$	$\{a, c, e\}$	$\{b, d\}$	$\{a, c, e\}$	$\{b, d\}$	$\{a, c, e\}$
$p4$	$\{a, b, d\}$	$\{a, b, d\}$	$\{c\}$	$\{a, b, d\}$	$\{e\}$

Eigenschaften der Modellanalysebeispiele Für die *Modellanalyse*beispiele τ_l^{Eq} und τ_{stop} übertragen sich die Eigenschaften und Komplexitäten der verwendeten Äquivalenzanalyse bzw. der *Wertebereichsanalyse*. So sind beide *Modellanalyse*beispiele flusssensitiv, interprozedural, *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv. Die konkrete Komplexität ergibt sich wiederum durch das eingesetzte Fixpunktverfahren. Im Fall von τ_l^{Eq} kommt der Aufwand für die Bildung der Äquivalenzklassen hinzu: $O(n \cdot m)$, wobei $n = |Marken(S)|$ und $m = |Vars(S)|$. Im Fall von τ_{stop} kommt der wesentlich geringere Aufwand zur Prüfung der Eigenschaften aus Lemma 4.3.19 hinzu, so dass der komplexitätstheoretische Aufwand unverändert bleibt.
Für das *Modellanalyse*beispiel τ_{rdz} wird das *SPDS* auf lesende *Variablenverwendungen* untersucht. Dies gelingt in linearer Zeit in der Größe der Modellbeschreibung für jede Variable, d.h. insgesamt in $O(m \cdot n)$. Die *Modellanalyse* ist dabei flussinsensitiv, interprozedural, *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv.
Die Bestimmung der Repräsentanten R ist prinzipiell NP-vollständig in der Anzahl lesender *Variablenverwendungen* $reads(S)$ im *SPDS* S und der Größe der Äquivalenzklassen τ_l^{Eq}. Da jedoch die Äquivalenzklassen i.d.R. entweder klein gegenüber der Gesamtanzahl der *SPDS*-Variablen ist (wenig ist miteinander äquivalent) oder es nur wenige (dann aber große Äquivalenzklassen) gibt, gestaltet sich die Problemstruktur derart gutartig, dass in der Praxis das auftretende ILP-Problem effizient gelöst werden kann. Zum konkreten Worst-Case-Laufzeitverhalten vom SMT-Löser YICES wurden leider keine Informationen gefunden. Allgemein können sämtlich mögliche Teilmengen $R_{var} \subseteq Vars$ überprüft werden ob es sich um eine gültige Repräsentantenmenge handelt. In diesem Fall würde der Aufwand $O(|2^{Vars}| \cdot |reads(S)|)$ entstehen. Es muss abschließend noch gezeigt werden, dass die Lösung des ILP auch einer *optimal*en Wahl an Variablen-Repräsentanten entspricht:

Lemma 4.3.20 (Korrektheit der Repräsentantenwahl)
Die Variablen-Repräsentanten R_{var} des ILPs aus Definition 4.3.31 auf Seite 100 sind wohldefinierte optimale *Variablen-Repräsentanten für ein gegebenes* SPDS *S.*

Der Beweis findet sich im Anhang B ab Seite 145.

Abstrakte Informationen des Motivationsbeispiels Die Ergebnisse der Äquivalenzanalyse (die Äquivalenzklassen) sind in *Quellcode* 4.14 in den Kommentaren als Eq_l abgebildet. So hat die Äquivalenzanalyse an der Marke $m2$ z.B. die Äquivalenzen $c = 0$ und $a = 1$ bestimmt und an der Marke $p1$ die Äquivalenzen $c = e$ sowie $a = d$. Entsprechend ergibt sich die Belegungsäquivalenz τ_l^{Eq} aus der Tabelle 4.1.
Für die Repräsentanten R werden im Beispiel 4.3.5 auf Seite 91 zunächst die Konstanten-Repräsentanten R_{const} bestimmt. Wegen der Konstanten in den Äquivalenzklassen in der Prozedur m ergeben sich gemäß Tabelle 4.1 die Konstanten-Repräsentanten $R_{const} = \{1, 2\}$. Bei Prüfung auf *Erreichbarkeit* von $p4$ ist $R_\phi = \emptyset$, d.h. $R_{fix} = R_{const}$. Zur Bestimmung der Variablen-Repräsentanten R_{var} werden die noch nicht ausdrückbaren lesenden *Variablenverwendungen* $reads(S) =$

Quellcode 4.14: abstrakte Informationen Eq_l und τ_l^R (R_l) von Beispielcode 4.13 .

```
int a(8),c(8);
init m;

void m() {
m1: c=1, a=2;
    # Eq_m2: c=1 a=2, R_m1(a)=2, R_m1(c)=1
m2: p(c,a), c=a, a=c;
    # Eq_m3: c=2 a=1, R_m2(c)=2, R_m2(a)=1
m3: p(c,a); }

void p(int e(8), int d(8)) {
int b(8);
    # Eq_p1: c=e a=d, R_p1(c)=e, R_p1(d)=d
p1: a=c, c=d;
    # Eq_p2: a=e c=d, R_p2(d)=d, R_p2(e)=e
p2: b=d, c=e;
    # Eq_p3: a=c=e b=d, R_p3(b)=d, R_p3(a)=e, R_p3(d)=d
    # Wenn a+d=0, dann versackt hier der aktuelle Lauf.
p3: a=b, c=a/(a+d);
    # Eq_p4: a=b=d
p4: return; }
```

$\{p1_c, p1_d, p2_d, p2_e, p3_b, p3_a, p3_d\}$ bestimmt. Daraus wird das ILP mit folgender Gestalt erstellt (wobei zur Übersicht die Werte 0 nicht dargestellt sind):

	$p1_c$	$p1_d$	$p2_d$	$p2_e$	$p3_b$	$p3_a$	$p3_d$
α_a		1		1		**1**	
α_b					**1**		1
α_c	**1**		1			1	
α_d		**1**	**1**		1		**1**
α_e	1			**1**		1	

Eine Zeile beschreibt hierin eine Variable $v \in \mathit{Vars}(S)$, welche als Repräsentant gewählt werden kann ($\alpha_v = 1$) oder nicht ($\alpha_v = 0$). Die Einträge innerhalb der Zeile werden daher mit α_v multipliziert. Eine Spalte charakterisiert die Überdeckungsmöglichkeiten einer lesenden *Variablenverwendung* durch belegungsäquivalente Variablen. So hat ein Tabelleneintrag in der Zeile α_v und der Spalte l_x den Wert 1 gdw. die lesende *Variablenverwendung* l_x einer Variablen x an der Marke l belegungsäquivalent zur Variable v ist. Für die Lösung des ILPs sind eine Teilmenge der Zeilen so auszuwählen ($\alpha_v = 1$), dass sich in jede Spalte mindestens eine 1 befindet, d.h. jede *Variablenverwendung* durch einen belegungsäquivalenten Repräsentanten abgedeckt ist. Eine Lösung dieses ILP ist z.B. $R_{var} = \{d, e\}$ (d.h. $\alpha_d = 1, \alpha_e = 1, \alpha_a = 0, \alpha_b = 0, \alpha_c = 0$,; dann steht in jeder Spalte der ausgewählten Zeilen mindestens eine 1). Damit ergeben sich zusammen die Repräsentanten $R = R_{fix} \cup R_{var} = \{1, 2, d, e\}$.
Die Repräsentantenwahl $\tau_l^R : \mathit{Vars}(S) \to \mathbb{N}' \cup \mathit{Vars}(S)$ kann damit gewählt werden als (dabei ist die Belegung der freien Felder gleichgültig, da dort $\mathit{used}_l^{read}(v) = \mathit{false}$ gilt):

τ^R	$m1$	$m2$	$m3$	$p1$	$p2$	$p3$	$p4$
a		2	1			e	
b						d	
c		1	2	e			
d				d	d	d	
e					e		

Die Redundanz τ_{rdz} und die Versackungsfreiheit τ_{stop} sind einerseits abhängig von der zu prüfenden *temporalen Formel* ϕ. Andererseits werden beide Analysen erst benötigt, wenn die ersten *Modelltransformationen* erfolgt sind. Dann werden z.B. *SPDS*-Variablen insbesondere durch die *Modelltransformationen redundant*. Im Beispiel 4.3.5 auf Seite 91 ist $\tau_{rdz}(v) = false$ für jede Variable $v \in Vars(S)$. Abgesehen von „$c = a/(a + d)$“ ist jede SPDS-*Anweisung* des zu Grunde liegenden *SPDS* S offensichtlich versackungsfrei ($\tau_{stop}(RENV_l^*, l : s) = false$). „$c = a/(a+d)$“ ist hingegen potentiell versackend, wenn der approximierte realisierbare Wertebereich $RENV_{p3}^*$ die *Variablenbelegung* $a = 0$ und $c = 0$ enthält. Diese Variablenbelegung ist jedoch nicht realisierbar und auch die approximierten *Wertebereiche* $RENV_{p3}^*$ erkennen dies, so dass auch $\tau_{stop}(RENV_{p3}^*, c = a/(a + d)) = false$. Würde statt der Konstanten 1 und 2 in der Prozedur p die Konstanten 0 und 1 verwendet werden, so würden die approximierten *Wertebereiche* $RENV_{p3}^*$ dies nicht erkennen und *konservativ* die Möglichkeit $a + d = 0$ an der Marke $p3$ im Modell vorhersagen. Dies wäre aber dennoch nicht realisierbar (*Konservativität* von $RENV_{p3}^*$ führt dann zur Konservativität von τ_{stop}).

4.3.4.5. Redundanzelimination $T_{omit}(T_{elim}^*(T_R(S)))$

Sind die *abstrakten* Informationen τ_l^{Eq} und τ_l^R bestimmt, so kann nun die innere Struktur des *Konfigurationenraum*s mittels *Modelltransformationen* komprimiert werden.

Idee Durch die Repräsentantenwahl τ_l^R wurden minimal nötige belegungsäquivalente Repräsentanten gewählt mit denen nun die lesend verwendeten *SPDS*-Variablen substituiert werden (*Konfigurationenraum*verlagerung). Danach verbleiben allerdings viele nicht (lesend) verwendete Variablen in der *SPDS*-Beschreibung. Diese und daran beteiligte SPDS-*Anweisungen* sind dann ggf. *redundant* und werden in einem zweiten bzw. dritten Schritt eliminiert (*Konfigurationenraum*verkleinerung).

Konkretisierung Lesende *Variablenverwendungen* (erster Schritt) können durch folgende *Modelltransformation* ersetzt werden:

Definition 4.3.33 (*Repräsentantenprägung* T_R)
Sei $Z \subseteq Stats(S)$ *die Menge aller* SPDS-Zuweisung*en* „$l : x = e$“ *in* S. *Die Repräsentantenprägung* T_R *für ein* SPDS S *besteht aus folgenden* Modelltransformationen*:*

$$\frac{\text{„}l : x = e;\text{“} \in Z \quad v \in Vars(S) \quad used_l^{read}(v) \quad v \neq \tau_l^R(v)}{l : x = e; \;\; \Rrightarrow \;\; l : x = (e[\tau_l^R(v)/v]);} \tag{4.23}$$

$$\frac{\text{„}l : s;\text{“} \in Stats(S) \setminus Z \quad v \in Vars(S) \quad used_l^{read}(v) \quad v \neq \tau_l^R(v)}{l : s; \;\; \Rrightarrow \;\; l : s[\tau_l^R(v)/v];} \tag{4.24}$$

Diese ersetzen auftretende lesende SPDS-Variablenverwendungen *von* v *in der SPDS*-Anweisung s *durch* $\tau_l^R(v)$ *und werden so lange angewendet wie möglich.*

Bei der Repräsentantenprägung verbleiben die definierten *SPDS*-Variablen in der Modellbeschreibung, so dass sich der *Konfigurationenraum* nicht ändert (wird im Anschluss gezeigt). Für die *Konfigurationenraum*verkleinerung (Variablenelimination im zweiten Schritt), sind *redundant*e und nicht versackende *SPDS*-Variablen zu identifizieren. Nichtrepräsentanten $x \in Vars \setminus R$ werden dabei z.B. ggf. nicht länger in der *SPDS*-Beschreibung benötigt. Wegen potentiellen Versackungen im Falle von arithmetischen Überläufen oder Division mit Null, werden danach die Modell-Eigenschaften τ_{rdz} und τ_{stop} bestimmt, um neue Fehlalarme (False Negatives) zu verhindern und somit das sichere (*konservative*) Eliminieren von *redundant*en Variablen zu ermöglichen. In einem zweiten Schritt werden daher *SPDS-Zuweisung*en eliminiert, um schreibende *Variablenverwendungen* von *redundant*en Variablen zu unterbinden:

Definition 4.3.34 (*Eliminierung von* SPDS-Zuweisung*en* T^*_{elim})
Sei T^*_{elim} *jene Modellreduktion, welche die* Modelltransformationen*:*

$$\frac{„l : x = e; “ \in Stats(S) \quad \tau_{rdz}(x) \quad \neg\tau_{stop}(RENV^*_l, l : x = e)}{l : x = e; \quad \Rrightarrow \quad l : skip;} \tag{4.25}$$

sowie

$$\frac{„l : x = e; “ \in Stats(S) \quad \tau_{rdz}(x) \quad \tau_{stop}(RENV^*_l, l : x = e)}{l : x = e; \quad \Rrightarrow \quad l : if \ ((e < 0)||(e > x_{max})) \ goto \ l;} \tag{4.26}$$

sooft auf $T_R(S)$ *anwendet wie möglich, wobei* $x_{max} := 2^{Bits(x)} - 1$.

Dabei werden *Zuweisung*en an *redundante* Variablen $x \in Vars$ eliminiert. Mittels *Modelltransformationsregel* 4.25 werden nicht versackende *Zuweisung*en komplett eliminiert. Und mit Modelltranformationsregel 4.26 wird für potentiell versackende *Zuweisung*en zumindest weiterhin der Ausdruck e ausgewertet, was auch weiterhin zu einer möglichen Versackung führt bei z.B. Division mit 0. Ist der Typ von x zu klein und damit $\exists env \in RENV_l : [\![e]\!]_{env} \notin range(x)$, so terminiert der Lauf nicht, verfängt sich aber in einer Endlosschleife. Gemäß Betrachtungen in Abschnitt 3.3.3 ab Seite 35 ist dies für temporale Formeln gleichbedeutend mit Terminierung, da im Falle der Terminierung eines Laufs der letzte Zustand unendlich oft wiederholt wird. Wichtig ist, dass dadurch die *redundante* Variable x nun nicht mehr schreibend verwendet wird und es damit nur noch lesende *Verwendungen* redundanter Variablen gibt. Sind alle unnötigen *SPDS-Zuweisung*en eliminiert, so können auch die Variablen eliminiert werden, welche dann nicht mehr verwendet werden. Sämtliche Nichtrepräsentanten $x \in Vars \setminus R$ werden nach der Repräsentantenprägung T_R nicht mehr lesend verwendet und können daher ggf. eliminiert werden. Aber auch Repräsentanten könnten durch die voran gegangenen Transformationen *redundant* geworden sein. Im dritten Schritt werden demnach alle *redundant*en *SPDS*-Variablen eliminiert und damit der *Konfigurationenraum* verkleinert:

Definition 4.3.35 (*Eliminierung von* SPDS-*Variablen* T_{omit})
Sei T_{omit} *jene Modellreduktion, welche auf das* SPDS $T^*_{elim}(T_R(S))$ *so lange wie möglich die* Modelltransformationen

$$\frac{v \in Vgbl \setminus R_{fix} \quad \tau_{rdz}(v)}{\begin{array}{ccc} Vgbl & \Rrightarrow & Vgbl \setminus v \\ S & \Rrightarrow & S[0/v] \end{array}} \tag{4.27}$$

zum Eliminieren globaler redundant*er Variablen,*

$$\frac{p \in Prz(S) \quad v \in Vlcl_p \setminus R_{fix} \quad \tau_{rdz}(v)}{\begin{array}{ccc} Vlcl_p & \Rrightarrow & Vlcl_p \setminus v \\ S & \Rrightarrow & S[0/v] \end{array}} \tag{4.28}$$

zum Eliminieren lokaler redundant*er Variablen, sowie*

$$\frac{p \in Prz(S) \quad v_i \in Param_p \setminus R_{fix} \quad \tau_{rdz}(v_i)}{\begin{array}{ccc} Param_p & \Rrightarrow & Param_p \setminus v_i \\ l : p(x_1, x_2, \ldots, x_m) & \Rrightarrow & l : p(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_m) \\ S & \Rrightarrow & S[0/v_i] \end{array}} \tag{4.29}$$

zum Eliminieren redundant*er Parametervariablen anwendet, wobei* $Param_p = [v_1, v_2, \ldots, v_m]$ *ist.*

Bemerkung 4.3.4 *Es gilt* $Prz(S) = Prz(T^*_{elim}(T_R(S)))$.

Repräsentanten werden im ILP so bestimmt, dass sie nicht *redundant* sind. Das Verhalten des *SPDS* hängt dann inhärent von den *Variablenbelegung*en der minimal nötigen Repräsentanten ab.

Quellcode 4.15: Repräesentantenprägung $T_R(S)$ mittels Repräsentanten τ_l^R für Beispiel 4.13 .

```
int a(8),c(8);
init m;

void m() {
m1: c=1, a=2;
m2: p(1,2), c=2, a=1;
m3: p(2,1); }

void p(int e(8), int d(8)) {
int b(8);
p1: a=e, c=d;
p2: b=d, c=e;
p3: a=d, c=e/(e+d);
p4: return; }
```

Quellcode 4.16: Eliminierung von *SPDS-Zuweisung*en T^*_{elim} in $T_R(S)$ für Beispiel 4.13.

```
int a(8),c(8);
init m;

void m() {
m1: skip;
m2: p(1,2);
m3: p(2,1); }

void p(int e(8), int d(8)) {
int b(8);
p1: skip;
p2: skip;
p3: skip;
p4: return; }
```

Modelloptimierung des Motivationsbeispiels Für Beispiel 4.13 auf Seite 91 wurden bereits Repräsentanten R bestimmt, obgleich es sich hierbei um ein *SPDS*' mit *SPDS*-Erweiterungen handelt. Die Konzepte übertragen sich in natürlicher Weise (wie bisher auch). Die Repräsentanten werden bei der Repräsentantenprägung T_R genutzt, um lesende *Variablenverwendungen* in der Prozedur m durch Konstanten $R_{const} = \{1, 2\}$ zu ersetzen und in der Prozedur p durch die beiden Repräsentanten $R_{var} = \{e, d\}$. Das Ergebnis $T_R(S)$ ist als *SPDS*' in *Quellcode* 4.15 abgebildet. Darin sind nun genau sämtliche Nichtrepräsentanten *konservativ redundant*. Nachdem die Repräsentantenprägung erfolgt ist, wird die Versackungsfreiheit τ_{stop} für $T_R(S)$ berechnet. Für die Versackungsfreiheit sind zunächst wieder die *konservativ* approximierten realisierbaren *Wertebereiche* $RENV_l^*$ wichtig, um daraus die Versackung τ_{stop} zu bestimmen. Im Beispiel ist die SPDS-*Anweisung* „$c = e/(e + d)$“ die einzige, welche möglicherweise nach der Repräsentantenprägung potentiell versackend sein könnte, wenn $e + d = 0$ realisierbar ist[49]. Die *konservativ* approximierten realisierbaren *Wertebereiche* erkennen aber, dass es nicht zu einer Division mit 0 kommen kann, so dass stets gilt: $\forall$„$l : s$“ $\in Stats(S) : \tau_{stop}(RENV_l^*, l : s) = false$. So entsteht (nach Rückinterpretation als *SPDS*') durch die Eliminierung von *SPDS-Zuweisung*en T^*_{elim} das *SPDS*' aus *Quellcode* 4.16. Dadurch werden nun die *SPDS*-Variablen a, b, c, d und e *redundant* (insbesondere auch *konservativ* redundant ($\forall v \in Vars(T^*_{elim}(T_R(S))) : \tau_{rdz}(v) = true$) und können mittels der *Modelltransformation* T_{omit}

[49] z.B. $e = 0$ und $d = 0$

Quellcode 4.17: Eliminierung von *SPDS*-Variablen T_{omit} in $T^*_{elim}(T_R(S))$ für Beispiel 4.13.

```
init m;

void m() {
m1: skip;
m2: p();
m3: p(); }

void p() {
p1: skip;
p2: skip;
p3: skip;
p4: return; }
```

aus dem Modell eliminiert werden. Das Ergebnis dieser *Modelltransformation* T_{omit} ist in *Quellcode* 4.17 auf Seite 107 abgebildet.
Eine anschließende Prozedurüberbrückung kann dann die *Prozeduraufrufe* $m2 : p()$; sowie $m3 : p()$; überbrücken, so dass die gesamte Prozedur p aus dem Modell entfernt werden kann (siehe Abschnitt 4.3.3 ab Seite 78). Der *Konfigurationenraum* verkleinert sich dann auf fünf *Konfiguration*[50] (abhängig von einer gegebenen *temporalen Formel*). Es gibt dann keine *Variablenbelegung*en, welche zu berücksichtigen sind und nur noch *skip*-Anweisungen.

Wohldefiniertheit Die Transformationsregeln 4.23 auf Seite 104 und 4.24 auf Seite 104 ersetzten auftretende lesende *Verwendungen* von Variablen durch Repräsentanten. Die Regeln unterscheiden nach *SPDS-Zuweisung*en zur Berücksichtigung von schreibenden *Variablenverwendungen* und anderen SPDS-*Anweisungen*. Da es nur endlich viele *Variablenverwendungen* gibt und diese nur durch Repräsentanten ersetzt werden, wenn sie selbst nicht schon Repräsentant sind, können beide Regeln nur endlich oft angewendet werden.
Der Wert *redundant*er Variablen ist unwichtig für die *Modellprüfung*, so dass *SPDS-Zuweisung*en an diese unterbunden werden können, ohne das *Modellprüfung*sresultat zu beeinträchtigen. Durch Eliminieren von *SPDS-Zuweisung*en T^*_{elim} ändern sich daher auftretende *SPDS*-Zuweisungen an *redundant*e Variablen entweder in *skip-Anweisungen* (bei Nichtversackung) oder in einen bedingten Sprung (an die gleiche Marke mit einer unerfüllbaren Bedingung bei möglicher Versackung). Da es nur endlich viele *SPDS-Zuweisung*en gibt, können auch diese Regeln nur endlich oft angewendet werden.
Sämtliche Nichtrepräsentanten $v \in Vars \setminus R$ sind z.B. bereits stets *redundant*, wie folgendes Lemma zeigt. Sie sind damit geeignete Kandidaten zum Eliminieren:

Satz 4.3.24 (Redundanz der Nichtrepräsentanten)
Seien die Repräsentanten R für ein SPDS S *bestimmt. Jede Variable $v \in Vars \setminus R$ ist* redundant *in $T_R(S)$.*

Beweis(Skizze) Sei eine Variable $v \in Vars(S) \setminus R$ beliebig aber fest gewählt. Wegen $R_\phi \subseteq R$ ist $v \notin \phi$ (falls überhaupt eine *temporale Formel* ϕ gegeben ist). Nach Konstruktion des ILPs resp. von R gibt es damit keine lesende *Variablenverwendung* $l_v \in reads(S)$, da diese im Rahmen der Repräsentantenprägung durch Repräsentanten substituiert wurden. Somit kommt v höchstens schreibend, d.h. links in *SPDS-Zuweisung*en vor. Die konkrete *Variablenbelegung* ist dann aber unerheblich, da dieser Wert nie wieder gelesen wird (alle lesenden *Variablenverwendungen* von v wurden durch Repräsentanten substituiert). Somit ist v *redundant*.

□

[50] Man beachte, dass die Prozedur m implizit durch ein *return* abgeschlossen ist.

Anderseits sind Variablen-Repräsentanten i.d.R. nicht *redundant*, da sie zur minimal nötigen Variablenauswahl gehören, um das Verhalten des *SPDS* wiederzuspiegeln. Wie folgendes Lemma zeigt, ist Redundanz nur eine Voraussetzung für die Eliminierbarkeit und nicht hinreichend, weswegen für T_{elim} neben der Redundanz auch die Versackungsfreiheit τ_{stop} betrachtet wurde.

Lemma 4.3.21 (Eliminierbarkeit ⇒ Redundanz)
Ist eine Variable $v \in Vars(S)$ eliminierbar, so ist sie auch redundant*. Die Umkehrung gilt nicht.*

Der Beweis findet sich im Anhang B ab Seite 145. Ein Nichtrepräsentant $x \in Vars \setminus R$ kommt in $T_R(S)$ nicht lesend vor, da alle lesenden *Verwendungen* durch Repräsentanten durch T_R substituiert wurden. Allerdings kann es noch schreibende *Verwendungen* geben. Diese können dann nur in *SPDS-Zuweisung*en auftreten, da nur in diesem Fall Variablenwerte verändert werden. Sie werden durch T^*_{elim} schließlich eliminiert, da nach Lemma 4.3.24 Nichtrepräsentanten automatisch redundant sind. Daher sind alle im *SPDS* $T^*_{elim}(T_R(S))$ auftretenden *Variablenverwendungen* von *redundant*en Variablen höchstens lesend. Da diese andererseits *redundant* sind, ist ihr Wert unwichtig für die *Modellprüfung*, so dass ihr Auftreten durch den Wert 0 ersetzt werden kann in der Modellreduktion T_{omit}. Andererseits wird durch jede der Regeln mindestens eine Variable der endlich vielen *SPDS*-Variablen eliminiert. Daher können diese Regeln nur endlich oft angewendet werden.
Damit sind die Modellreduktionen T_R, T^*_{elim} sowie T_{omit} wohldefiniert.

Korrektheit (Invarianz Temporaler Formeln und Erreichbarkeit) Es ist klar, dass die Aussagen von *temporalen Formel*n unverändert bleiben, solange lediglich lesende *Verwendungen* von Variablen durch andere (belegungsäquivalente) ersetzt werden wie bei T_R, da sich dadurch Variablenwerte insgesamt nicht verändern. Die folgenden Sätze 4.3.25, 4.3.26 sowie 4.3.27 gelten für ein beliebiges $\alpha \in$ {ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X, ACTL, ACTL*, CTL, CTL*, LTL}. Die Beweise folgender Sätze und Lemmas finden sich im Anhang B ab Seite 145.

Satz 4.3.25 (Korrektheit von T_R)
Die Repräsentantenprägung T_R ist α-invariant für gegebenes $\phi \in \alpha$.

Werden nun durch die Repräsentantenprägung T_R weitere Variablen als überflüssig bzw. *redundant* erkannt, so könnten diese ggf. aus dem Modell entfernt werden. Wenn davon ausgegangen wird, dass nur das Modell geändert und die *temporale Formel* unverändert bleibt, so geht dies i.A. nur, sofern die gegebene *temporale Formel* nicht über diese 'wegoptimierten' Variablen spricht. Denn andernfalls wäre das Vorkommen der Variablen innerhalb der *temporalen Formel* unwichtig. Wie in Lemma 4.3.21 gezeigt, ist Redundanz nur eine hinreichende Bedingung. Wird wie in Definition 4.3.34 auf Seite 105 zusätzlich die Versackung betrachtet, so werden beide Eigenschaften zusammen hinreichend:

Lemma 4.3.22 (Hinreichende Eliminierbarkeit)
Ist die SPDS-*Variable $x \in Vars(S)$* redundant *$\tau_{rdz}(x) = true$ und „$l : x = e;$“ $\in Stats(S)$ versackungsfrei $\tau_{stop}(RENV_l$, „$l : x = e;$“$) = false$, so ist „$l : x = e;$“ $\in Stats(S)$ eliminierbar $elim_\zeta($„$l : x = e;$“$) = true$ im* SPDS *S, wobei $\zeta \in \{\phi, L0\}$.*

Hieraus ergibt sich als Folgerung die Korrektheit für die Transformationsregel 4.25 auf Seite 105. Die Korrektheit der Transformationsregel 4.25 auf Seite 105 wird ähnlich gezeigt, was zu folgendem Satz führt:

Satz 4.3.26 (Korrektheit von T^*_{elim})
Das Eliminieren von SPDS-Zuweisung*en der Form „$l : x = e;$“ mittels* Modelltransformation *T^*_{elim} ist α-invariant für gegebenes $\phi \in \alpha$.*

Gleiches gilt für die Eliminierung von *SPDS*-Variablen T_{omit} (Definition 4.3.35 auf Seite 105).

Satz 4.3.27 (Korrektheit von T_{omit})
Das Eliminieren von SPDS-*Variablen mittels Modellreduktion T_{omit} ist α-invariant für das* SPDS *$T^*_{elim}(T_R(S))$ und gegebenes $\phi \in \alpha$.*

Für die Sätze 4.3.25, 4.3.26 sowie 4.3.27 ergibt sich damit auch die *Erreichbarkeits-Invarianz* für a priori gegebene Marken $L0 \in Marken(S)$. Insbesondere ist damit jeder im *SPDS* $T^*_{elim}(T_R(S))$ nicht verwendete Nichtrepräsentant $v \in Vars \setminus R$ eliminierbar.

4.3.4.6. Eigenschaften

Nun werden einige Eigenschaften zu den beschriebenen *Modelltransformationen* erläutert.

Entscheidbarkeit der optimalen Redundanzelimination Die Qualität der Approximationen (insbesondere für die Versackungsfreiheit) entscheidet stark über die Form und Güte der Modellreduktionen. Die *Optimal*ität der Modellreduktionen wird daher über die Optimalität der abstrakten Informationen definiert. Für die Eliminierung von SPDS-*Anweisungen* übertragen sich dabei Überlegungen aus Abschnitt 4.3.3 ab Seite 78:

Definition 4.3.36 (*Optimale Redundanzelimination*)
Eine Redundanzelimination T heißt präzise *bezüglich einer Redundanzanalyse $\tau_{rdz} : Vars(S) \to \{true, false\}$, wenn T genau die* redundant*en Variablen $Vars_{rdz} := \{v \in Vars(S) \mid \tau_{rdz}(v) = true\}$ eliminiert. T heißt* optimal*, wenn τ_{rdz}* exakt *ist.*
*Eine Eliminierung von SPDS-*Anweisungen *T' ist* präzise *bezüglich einer Menge $\Lambda \subseteq Stats(S)$ an SPDS-*Anweisungen[51]*, wenn sie das* SPDS *S auf genau die SPDS-*Anweisungen *Λ beschneidet oder die SPDS-*Anweisungen *„$l : s$“ $\notin \Lambda$ eliminiert. T' heißt* optimal*, wenn Λ genau die Menge der nicht-eliminierbaren SPDS-*Anweisungen *darstellt. D.h. wenn gilt: „$l : s$“ $\notin \Lambda \Leftrightarrow elim_\zeta(l : s)$ für $\zeta \in \{\phi, L0\}$.*
Analog ist eine Repräsentantenprägung T'' präzise, wenn sie genau die gewählten Repräsentanten R prägt. Sie ist optimal*, wenn die Repräsentanten exakt bestimmt werden.*

Die Repräsentantenprägung T_R ist demnach zwar präzise, aber erst optimal, wenn die eingesetzte Äquivalenzanalyse zur Bestimmung belegungsäquivalenter *Ausdrücke* exakt ist. Als *Modellanalyse*beispiel wurde wegen ihrer Komplexität nur eine *konservative* Approximation für belegungsäquivalenter *Ausdrücke* eingesetzt, so dass die Repräsentantenprägung T_R nicht *optimal* ist, obwohl die Repräsentanten mittels ILP-Löser bestimmt werden.
Die Modellreduktion T^*_{elim} zum Eliminieren von SPDS-*Anweisungen* ist *präzise* bezüglich der Menge an nichtversackenden *Zuweisung*en an *redundant*e Variablen. Für versackende Zuweisungen an redundante Variablen wird nicht T_{elim} verwendet, um die Versackung zu erhalten. Daher ist sie für diese SPDS-*Anweisungen* nicht *präzise*. Sie ist zudem nicht *optimal*, da lediglich *SPDS-Zuweisung*en eliminiert werden und auch nur solche, welche an *redundant*e Variablen Werte zuweisen. Sie wird nur eingesetzt, um das Eliminieren der *redundant*en Variablen zu ermöglichen bzw. zu vereinfachen.
Die Redundanzelimination T_{omit} schließlich ist ebenso *präzise*, wenn in der *temporalen Formel* keine *redundant*en Variablen auftreten. Im vorgestellten *Modellanalyse*beispiel τ_{rdz} ist dies der Fall, so dass T_{omit} bezüglich τ_{rdz} *präzise* ist. T_{omit} ist aber nicht *optimal*, da τ_{rdz} wegen ihrer Komplexität nur eine *konservative* Approximation für redundante Variablen darstellt.
Würden die entsprechenden *Modellanalyse*beispiele exakt durchgeführt, was prinzipiell möglich ist, so wäre die vorgestellte Redundanzelimination $T_{omit}(T^*_{elim}(T_R(S)))$ *optimal*. Demnach ist die optimale Redundanzelimination entscheidbar.

Fehlalarme Durch die Redundanzelimination $T_{omit}(T^*_{elim}(T_R(S)))$ entstehen wegen α-Invarianz keine neuen Fehlalarme (weder *False Negatives* noch *False Positives*). Das Verhalten des *SPDS* sowie das *Modellprüfung*sresultat bleiben erhalten.

Konfigurationenraumveränderung Die Modellreduktionen T^*_{elim} und T_R verändern die Größe des *Konfigurationenraum*s nicht. Zwar werden *Konfigurationenübergänge* mittels T_{elim} eliminiert,

[51] z.B. ein Slice aus Abschnitt 4.3.3 ab Seite 78

Tabelle 4.2.: Aufwand der *Modellanalyse*beispiele (vgl. Aufwand für Definitionen 4.3.11 und 4.3.5)

Analyse	Beschreibung	Aufwand
Eq_l	Belegungsäquivalente *Ausdrücke*	$O(j \cdot m \cdot k_{max} \cdot n2^2)$
τ_l^E	*konservative* Belegungsäquivalenz	$O(m \cdot n)$
R	*optimale* Repräsentanten	$O(\lvert 2^{Vars}\rvert \cdot \lvert reads(S)\rvert)$
τ_l^R	Repräsentantenwahl	$O(n)$
τ_{rdz}	*konservative* Redundanz	$O(m \cdot n)$
τ_{stop}	*konservative* Versackungsfreiheit	$O(m \cdot n \cdot \lvert ENV\rvert \cdot j \cdot k_{max} \cdot r \cdot t)$

$m = |Marken(S)|,\ n = |Vars|,\ n2 = |Vars \cup \mathbb{N}'|^2,\ k_{max} = max.\ Verzweigungsgrad,\ j = Anzahl\ Fixpunktiterationen,\ t = Vereinigung\ Variablenbelegungsmengen,\ r = O(E|_A^B)$.

dabei werden Marken allerdings nicht aus dem *SPDS* entfernt[52]. Beim Eliminieren von *SPDS*-Variablen mittels T_{omit} wird der *Konfigurationenraum* kleiner, wie am Beispiel 4.3.5 auf Seite 91 gezeigt wurde. Wie sehr sich der *Konfigurationenraum* verändert ist stark vom *SPDS* und den eingesetzten *Modellanalyse*beispielen abhängig. Wird z.B. eine globale Variable v mit Typ $bits(v) = 32$ eliminiert, so verkleinert sich der *Konfigurationenraum* um den Faktor 2^{32}. Auf diese Weise konnten z.B. die $1,1 \cdot 10^{13}$ *Konfigurationen* aus Beispiel 4.3.5 auf Seite 91 durch Eliminierung sämtlicher *SPDS*-Variablen auf lediglich $2 \cdot 8 + 5 = 21$ *Konfigurationen* reduziert werden (siehe *Quellcode* 4.17 auf Seite 107).

Komplexität Für die Redundanzelimination werden verschiedene *Modellanalyse*beispiele mit unterschiedlichem Aufwand zur Bestimmung von *abstrakten* Informationen eingesetzt. Der jeweilige Aufwand wurde bereits erörtert und ist in Tabelle 4.2 zusammengefasst (jeweils für das *SPDS* S, welches sich nur unwesentlich in den entscheidenden Einflussfaktoren von $T_R(S)$ bzw. $T^*_{elim}(S)$ unterscheidet). Damit wird für sämtliche *Modellanalyse*beispiele insgesamt ein Aufwand benötigt von $O(j_1 \cdot m \cdot k_{max} \cdot n2^2 + m \cdot n + |2^{Vars}| \cdot |reads(S)| + n + m \cdot n + m \cdot n \cdot |ENV| \cdot j_2 \cdot k_{max} \cdot r \cdot t) = O(|2^{Vars}| \cdot |reads(S)| + m \cdot n2^2 \cdot |ENV| \cdot j_{max} \cdot k_{max} \cdot r \cdot t)$, wobei j_1 bzw. j_2 die Anzahl der Fixpunktiterationen der Wertebereichsanalyse bzw. Äquivalenzanalyse und $j_{max} := \max(j_1, j_2)$ sind.
Die verschiedenen eingesetzten *Modelltransformationen* besitzen jeweils den deutlich geringeren Aufwand von ...

- $O(|S|)$ für T_R (da höchstens alle lesenden *Variablenverwendungen* ersetzt werden),
- $O(m)$ für T^*_{elim} (da höchstens jede *SPDS-Zuweisung* ersetzt wird) und
- $O(|S| \cdot n)$ für T_{omit} (da höchstens n Variablen eliminiert werden mit jeweiliger Modifikation des *SPDS* S).

Insgesamt führt dies zu einem Transformationsaufwand von $O(|S| + m + |S| \cdot n) = O(|S| \cdot n)$.

Aufwand vs. Nutzen Einerseits kann durch *präzise*re abstrakte Informationen (hierzu zählen Belegungsäquivalenz, Redundanz sowie Versackungsfreiheit) der innere *Konfigurationenraum* i.d.R. stärker komprimiert werden. Andererseits kann durch *Wiederverwendung* von Analyseergebnissen anderer Modellreduktionen der Aufwand zur *Modellanalyse* reduziert werden. Für die Versackungsfreiheit τ_{stop} und die *konservativ* approximierten realisierbaren *Wertebereiche* $RENV_l^*$ ergeben sich z.B. die gleichen Analyseergebnisse, wenn statt des *SPDS* $T_R(S)$ das *SPDS* S verwendet wird. Die Ergebnisse bzw. *abstrakten* Informationen werden dann den Substitutionen durch T_R entsprechend angepasst. Diese abstrakten Informationen erneut auch für $T_R(S)$ zu berechnen ist daher nicht nötig und kann von anderen *Modellanalyse*n wiederverwendet werden. Für die Praxis zeigt sich eine deutliche Reduktion des *Konfigurationenraum*s mittels nicht exakter Analyseergebnissen. Wie

[52] Wie beschrieben zielt die Redundanzelimination auf Minimierung verwendeter Variablen ab, so dass überflüssige *Konfigurationenübergänge* in späteren Schritten durch eine Stotterreduktion [67] oder durch Slicing (siehe Abschnitt 4.3.2 ab Seite 68) eliminiert werden.

Beispiel 4.3.5 auf Seite 91 zeigt, kann durch die Redundanzelimination der *Konfigurationenraum* bereits mit nicht exakten Analyseergebnissen sogar *optimal* verkleinert werden. Dies liegt u.a. an den zu *Programmanalysen* vergleichsweise *präzise*n *Modellanalyse*n mit Fixpunktberechnungen.
Die aufwändigste *Modellanalyse* für die Variablenredundanzelimination ist die *Wertebereichsanalyse* zur Bestimmung der *konservativen* Versackung. Im Rahmen dieser Arbeit sollten allerdings keine neuen *Fehlalarme* generiert werden. Kann das Auftreten zusätzlicher *False Negatives* toleriert werden, so kann auf die aufwändige Analyse zur Versackung und die *Wertebereichsanalyse* verzichtet werden. Dann werden sämtliche SPDS-*Anweisungen* als nicht versackend angenommen und eigentlich versackende *SPDS*-Läufe würden nach Anwendung der Modelltranformationsregel T_{elim} nicht mehr versacken. D.h. die α-Invarianz ist dann nicht mehr erfüllt und die *Erreichbarkeit* wird *konservativ* überapproximiert. Andererseits ist die Versackung aber auch u.U. nur durch unsachgemäße *Modellabstraktion* bei Beschränkung auf sehr wenige Bits in das *SPDS* gelangt (z.B. bei Modellierung der 32-Bit Addition mittels 8-Bit *SPDS*-Addition ohne vorherige Bereichsprüfung). In so einem Fall kann es durch Versackungserhaltung ggf. zu *False Positives* kommen, da z.B. Fehlermarken durch die unsachgemäße *Abstraktion* nicht *erreichbar* wurden und bei Modellreduktion dann auch weiterhin nicht erreichbar sind.
Wie Experimente gezeigt haben, beschleunigen *optimal* gewählte Repräsentanten die *Modellprüfung* besser als heuristisch gewählte. Dies liegt u.a. daran. dass die Gesamtanzahl der *SPDS*-Variablen in den untersuchten Modellen gering ist (typisch für Modellprüfung) und damit die zu bestimmenden exakten Variablenrepräsentanten effizient durch einen ILP-Löser bestimmt werden konnten (dann ist auch das ILP-Problem relativ klein).

4.3.4.7. Zusammenfassung

Rückblick Es wurde in diesem Abschnitt gezeigt, dass die Größe des Konfigurationenraums stark durch *SPDS*-Variablen beeinträchtigt wird. Ein *SPDS*-Beispiel verdeutlichte dabei, dass das nötige Kelleralphabet des *PDS* beim Eliminieren einer Variable deutlich kleiner wird als beim Eliminieren einer Marke. Daher wurde ein Verfahren entwickelt, um das Verhalten eines *SPDS* mit so wenig Variablen wie möglich zu beschreiben. Dabei wurden bestehende Techniken anderer Modellprüfer auf *SPDS* erweitert und mit Techniken aus dem Übersetzerbau und Programmanalysen kombiniert, welche zur Berücksichtigung der zu prüfenden Eigenschaften entsprechend angepasst wurden. Das entwickelte Verfahren besteht aus den drei Phasen: Bestimmung des Redundanzraums, Kompression der Redundanzen und Elimination der dabei entstandenen redundanten *SPDS*-Variablen. Für die erste Phase wurden entsprechende abstrakte Informationen betrachtet, um den Redundanzraum zu charakterisieren. Dazu wurden die Konzepte belegungsäquivalenter Ausdrücke, Repräsentanten, Redundanz, tote *SPDS*-Anweisungen und die Eliminierbarkeit von *SPDS*-Modellteilen erläutert und deren Eigenschaften untersucht. Dabei zeigte sich, dass diese Eigenschaften (im Gegensatz zu Hochsprachen) für *SPDS* exakt bzw. optimal bestimmbar sind. Die Berechnungskomplexität optimaler belegungsäquivalenter Ausdrücke, exakter Redundanz sowie exakter Eliminierbarkeit übersteigt dabei die der Modellprüfung, wohingegen optimale Repräsentanten effizienter bestimmt werden können (NP-hart). Daher wurden in diesem Abschnitt einsetzbare Programmanalysen betrachtet und abgesehen von den Repräsentanten daraus konservative Approximationen als Modellanalysebeispiele entwickelt (Heuristiken). Hierzu gehören die vorgestellte konservative Belegungsäquivalenz, die konservative Redundanz sowie die konservative Versackungsfreiheit. Dazu wurden jeweils die Konservativität nachgewiesen und entsprechende Eigenschaften untersucht (flusssensitiv, interprozedural, konservativ, entscheidbar, kontextinsensitiv, Aufwand etc.). Die Repräsentanten werden hingegen im vorgestellten Verfahren exakt bestimmt mittels des ILP-Lösers GLPK. Die Korrektheit dieser Repräsentantenwahl wurde ebenso nachgewiesen. Diese Modellanalysebeispiele wurden am *SPDS*-Beispiel veranschaulicht, um dann die Variablenredundanzelimination zu erläutern. Letztere minimiert die Anzahl der verwendeten *SPDS*-Variablen, indem in drei Stufen verschiedene Modelltransformationen nacheinander angewendet werden, und komprimiert so die innere Struktur des *SPDS*. Durch die Repräsentantenprägung T_R werden in einem ersten Schritt lesend vorkommende redundante *SPDS*-Variablen durch Repräsentanten ersetzt. In einem zweiten Schritt werden durch das Eliminieren entsprechender *SPDS*-Zuweisungen durch T^*_{elim} dann auch schreibende *SPDS*-Variablenzugriffe eliminiert, so dass im dritten und letz-

ten Schritt redundante Variablen durch T_{omit} komplett aus dem Modell entfernt werden können. Nachdem diese Schritte am *SPDS*-Beispiel veranschaulicht wurden, wurden wichtige Eigenschaften erörtert und bewiesen. So sind Nichtrepäsentanten z.B. stets redundant (Satz 4.3.24 auf Seite 107), was eine zusätzliche Modellanalyse zur Redundanzanalyse erübrigt. Weiter wurde erläutert, warum die Analyse der Redundanz alleine nicht genügt für eine temporal invariante Modellreduktion. Abschließend wurden verschiedene Eigenschaften betrachtet und die Korrektheit gezeigt. Demnach gibt es nur einen Einfluss auf den Konfigurationenraum für T_{omit}. Dennoch bleiben bei der vorgestellten Redundanzelimination Aussagen temporaler Formeln erhalten. T^*_{elim} und T_R sind präzise und die Redundanzelimination $T_{omit} \circ T^*_{elim} \circ T_R$ ist nicht optimal, obgleich sie entscheidbar ist. Der Aufwand wird auch hier wieder durch die Modellanalysen bestimmt (nicht durch die Modelltransformationen) und hängt von der konkreten Realisierung (Fixpunktverfahren) der Modellanalysen ab.

Fazit Durch die vorgestellte Variablenredundanzelimination $T_{omit} \circ T^*_{elim} \circ T_R$ können Redundanzen in *SPDS* verringert werden. Dies beschleunigt die Modellprüfung und andere Modellanalysen. Betrachtete vergleichbare Arbeiten erkennen und entfernen häufig nur unwichtige (z.B. tote) Variablen [96, 195]. Wie gezeigt, ist es möglich, die innere Struktur des Zustandsraums von *SPDS* umzustrukturieren, so dass nur noch minimal viele Variablen zur Verhaltensbeschreibung benötigt werden. Vergleichbar zur Variablenredundanzelimination ist Macro Expansion [254], wo im Modellprüfprozess direkt BDD-Variablen durch äquivalente *Ausdrücke* substituiert werden. Dazu ist es jedoch zunächst nötig, erst einmal das Modell zu erzeugen. In dieser Arbeit wird die Transformation bereits direkt in der *symbolisch*en Modellbeschreibung durchgeführt. In ähnlicher Weise werden bei Promela im *Modellprüfer* SPIN nichtlesend verwendete Variablen erkannt und aus dem Modell entfernt [96], allerdings ohne Beachtung von Rekursion und ohne Berücksichtigung *temporaler Formel*n. Die entworfene Variablenredundanzelimination eliminiert selbst ohne Kompression der inneren Struktur bereits nichtlesend verwendete Variablen aus dem *SPDS*, so dass i.d.R. deutlich mehr Variablen eliminiert werden können als im [96] beschriebenen Verfahren, wenn man es direkt auf *SPDS* übertragen würde. In der betrachteten Literatur wurde keine Kompression der inneren Struktur durch Konfigurationenraumverlagerung auf belegungsäquivalente Bereiche für *SPDS* betrachtet. Dies ist in dieser Arbeit erfolgt. Ergebnisse redundanter Berechnungen werden häufig in Variablen zwischengespeichert, was durch die maschinenunabhängige Übersetzerbauoptimierung zur „Elimination redundanter Berechnungen“ [195] reduziert werden kann. Diese Variablen können in *SPDS* zu belegungsäquivalenten Variablen und dann als Repräsentanten gegeneinander substituiert werden. Wiederholte Berechnungen des gleichen Werts werden so idealerweise automatisch unterbunden, wenn nur eine der äquivalenten Variablen im Restmodell verbleibt (und zwar jene, die den Konfigurationenraum kleiner macht). Die untersuchten Arbeiten treffen keine Aussagen zur Existenz und Komplexität bezüglich optimalen Eliminierens von Variablen (nicht berechenbar in turingmächtigen Programmiersprachen) für *SPDS* wurden in den untersuchten Arbeiten nicht getroffen (abgesehen von eigenen Veröffentlichungen wie z.B. [56]).

4.3.5. Wertebereichsreduktion

In diesem Abschnitt wird der *Konfigurationenraum* verkleinert, indem der Typ von definierten Variablen minimiert wird.

4.3.5.1. Motivation

Viele Variablen in höheren *Programmiersprachen* nehmen während ihrer Laufzeit oft nur wenige Variablenwerte des *Wertebereich*s an, wie z.B. den Wert 0 oder 1 (andernfalls würden die meisten *Programme* sehr lange Laufzeit besitzen, um alle möglichen Variablenwerte zu realisieren). Diese *Variablenbelegung* überträgt sich dann während der Modellkonstruktion auch in das Modell. Zwar können mit der Variablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90 überflüssige Variablen identifiziert und eliminiert werden, doch können diese ggf. noch einen unnötig großen Typ besitzen.

Problembewusstsein Es gibt im interpretierenden *PDS* um so mehr *Konfigurationenübergänge* $s \to z$, je größer der Typ einer *SPDS*-Variable ist und je mehr mögliche Werte für v an einer Marke l ausgewertet werden können. Insbesondere wenn Modelle automatisch gewonnenen bzw. aus höheren *Programmiersprachen* generiert werden, sind Variablentypen sehr groß. Gängige Praxis zur Modellverkleinerung ist eine Einschränkung auf nur sehr wenige Bits (z.B. 5 oder 8), damit der *Modellprüfer* das Modell noch erfolgreich verifizieren kann. Erfolgt diese manuelle Einschränkung der Bitbreite ohne Berücksichtigung der Belegung (Prüfung auf Einhaltung der Bereichsgrenzen), so kann es sogar (wie in Abschnitt 4.3.4.6 auf Seite 109 beschrieben) zu *False Positives* kommen. Werden die Typen im Modell automatisch minimiert (ausgehend von hinreichend großen Typen), so können einerseits *False Positives* bei der *Modellabstraktion* vermieden werden und andererseits wird der *Konfigurationenraum* auf nur noch den nötigen Teil beschränkt bzw. beschnitten. Dies wird im Folgenden genauer erläutert. Natürlich profitieren Modellreduktionstechniken dabei wiederum von Synergieeffekten. Denn durch Anwendung anderer *Modelltransformationen* oder der *Interpretation* von SPDS*'-Anweisung* können Variablen mit unnötig großem Typ im Modell vorliegen. Insbesondere lokale und Parameter-Variablen sind dabei wichtig, da diese wegen Rekursion bei steigender Rekursionstiefe mehr und mehr den *Konfigurationenraum* vergrößern, wenn diese mehrfach in unterschiedlichen Rekursionstiefen auftreten.

Beispiel 4.3.6 *Als Beispiel diene wieder die Prozedur heapify aus* Quellcode *4.18. Zu Prüfen sei die Haldenbedingung nach* Prozeduraufruf *von heapify:*

$$\phi = G\ m2 \Rightarrow \bigwedge_{0 \leq i < 50} (a[i] \geq a[2 * i]) \wedge (a[i] \geq a[2 * i + 1]). \tag{4.30}$$

Da die Reihungseinträge *jeweils die* Variablenbelegung 0 *oder* 1 *haben können, sind sie nicht im Rahmen der Ausdrucksvereinfachung durch z.B. Konstanten ersetzbar. Auch die* Variablenverwendungen *von i und p sind nicht weiter vereinfachbar. Der* Seiteneffekt *zur Herstellung der Haldenbedingung verhindert ebenso, dass im Modell ein* Prozeduraufruf *überbrückt werden kann. Auch ist die Variablenredundanzelimination nicht in der Lage eine* SPDS-*Variable zu eliminieren, etc. Die* Reihungseinträge *haben allerdings lediglich die Variablenwerte* 0 *oder* 1*, so dass der Typ sämtlicher* Reihungselemente *auf* 1 *Bit minimiert werden kann. Dadurch wird der* Konfigurationenraum *um* 700 *Bits und damit um den Faktor* $2^{700} = 5,3 \cdot 10^{210}$ *an* Konfigurationen *kleiner. Der theoretische* Konfigurationenraum $Konf(S)$ *ist im Gegensatz zum realisierbaren unendlich. Zumindest jedoch wird die Anzahl an möglichen Köpfen (*Interpretation *von SPDS'*-Anweisungen *unberücksichtigt) von* $2^{800} * 4 + 2^{800} * 6 * 2^8 * 2^8 = 3 \cdot 10^{246}$ *auf* $2^{100} * 4 + 2^{100} * 6 * 2^8 * 2^8 = 5 \cdot 10^{35}$ *reduziert.*

Lösungsidee Analog zur *SPDS*-Beschneidung aus Abschnitt 4.3.2 ab Seite 68 können die Variablentypen beschnitten werden, so dass der Typ minimal wird. Dabei sollen formal große Wertebereiche auf kleinere abstrahiert werden, falls weg abstrahierte Werte keinen Einfluss auf das Modellprüfungsergebnis haben (präzise Abstraktion) [71]. Wegen initial willkürlich belegter globaler und lokaler *SPDS*-Variablen wird der maximal realisierbare Wert einer Variablen v stets von seinem Typ abhängen:

$$\max\{[\![v]\!]_{env} \mid l \in Marken, env \in RENV_l\} = 2^{bits(v)} - 1.$$

Somit ist der maximal realisierbare Wert ungeeignet um den Typ zu minimieren. Dies trifft im Beispiel 4.3.6 für den Parameter p bei Prozedureintritt an der Marke $l0$ zu sowie für die *Reihung a* an der Marke $m0$. Dort sind für p bzw. *Reihungseinträge* von a sämtliche Werte $0 \ldots 255$ realisierbar. Um dennoch den Typ minimieren zu können, muss das Modellverhalten genauer analysiert werden. So kommen z.B. nur die realisierbaren Werte jener Marken in Betracht, in denen die Variable auch verwendet wird (lesend oder schreibend).

4.3.5.2. Notwendige abstrakte Informationen

Um den Variablentyp zu minimieren, ist der maximal verwendete Variablenwert an gewissen Marken wichtig. Wird eine *SPDS*-Variable lesend oder schreibend verwendet, so ist deren maximaler Variablenwert entscheidend.

Quellcode 4.18: Beispiel zur Berechnung der Haldenstruktur.

```
int a(8)[100];
init m;

void m() {
m0: A j (0,99) a[j]=choose(0,1);
m1: heapify(99);
m2: return; }

void heapify(int i(8)) {
      int p(8);
l0:   if (i>0) heapify(i-1);
l1:   p = (i-1)/2;
l2:   if (a[p] >= a[i]) return;
l3:   a[p] = a[i], a[i] = a[p];
l4:   i = p;
l5:   if (i>0) goto l1; }
```

Definition 4.3.37 (*Maximalverwendung* $\tau_{max}(v)$)
Sei ein SPDS S *und eine* SPDS-*Variable* $v \in Vars(S)$ *gegeben. Dann heißt* $\tau_{max} : Vars(S) \to \mathbb{N}_{\geq 0}$ *mit*

$$\begin{aligned} \tau^*_{max}(v) \quad := \quad & \max(\{[\![v]\!]_{env} \mid l \in Marken(S), used^{read}_l(v), env \in RENV_l\} \\ & \cup \{[\![e]\!]_{env} \mid „l : v = e; “ \in Stats(S), env \in RENV_l\}) \end{aligned}$$

Maximalverwendung *von* v. *Wird* v *weder lesend noch schreibend verwendet, so ist* $\tau^*_{max}(v) := 0$. *Die abstrakte Information* $\tau_{max} : Vars(S) \to \mathbb{N}_{\geq 0}$ *ist eine* konservative Approximation *von* τ^*_{max}, *falls für jede* SPDS-*Variable* v *gilt:* $\tau_{max}(v) \geq \tau^*_{max}(v)$. τ_{max} *ist exakt, wenn für jede* SPDS-*Variable* v *gilt:* $\tau_{max}(v) = \tau^*_{max}(v)$.

O.B.d.A. sei im Folgenden stets $\tau^*_{max}(v) > 0$ (und damit auch $\tau_{max}(v) > 0$). Denn andernfalls wäre v konstant (0) und durch die Ausdrucksvereinfachung aus Abschnitt 4.3.1 ab Seite 45 und der Variablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90 eliminierbar.

Eigenschaften Eine Versackung tritt genau dann ein, wenn $\tau^*_{max}(v) \notin range(v)$. Da $\tau^*_{max}(v)$ formal definiert ist über berechenbare *Variablenbelegung*en, ist natürlich auch $\tau_{max}(v)$ berechenbar und kann damit für *SPDS* exakt bestimmt werden. Allerdings ist auch diese aufwändiger als die Modellprüfung selbst:

Satz 4.3.28 (Komplexität der Maximalverwendung)
Die Berechnung exakter Maximalverwendungen ist mindestens so aufwändig wie die Modellprüfung.

Beweis(Skizze) Analog zum Beweis von Satz 4.3.19 auf Seite 95.

□

4.3.5.3. Einsetzbare Programmanalysen

Da $\tau_{max}(v)$ auf der Abschätzung von *Wertebereich*en beruht, können hierzu die gleichen *Modellanalyse*n genutzt werden, wie bei der *Wertebereichsanalyse* (siehe Abschnitt 4.3.1.3 auf Seite 48).

4.3.5.4. Modellanalysebeispiel

Idee Zur Bestimmung der möglichen Werte, wird die Analyse $values_l$ aus Abschnitt 4.3.1.4 ab Seite 51 zur *konservativen* Bestimmung realisierbarer *Variablenbelegung*en $RENV^*_l$ wiederverwendet. Daraus wird die *Maximalverwendung* approximiert.

Quellcode 4.19: Analyse-Ergebnisse für Beispiel 4.3.6.

```
int a(8)[100];
init m;

void m() {                          #a*=(0,255)
m0: A j (0,99) a[j]=choose(0,1); #a*=(0,1)
m1: heapify(99);                    #a*=(0,1)
m2: return; }

void heapify(int i(8)) {
      int p(8);                     #i=(0,99)
l0:   if (i>0) heapify(i-1);        #i=(0,99), p=(0,255)
l1:   p = (i-1)/2;                  #p=(0,49)
l2:   if (a[p] >= a[i]) return;     #a*=(0,1)
l3:   a[p] = a[i], a[i] = a[p];     #a*=(0,1)
l4:   i = p;                        #i=(0,49)
l5:   if (i>0) goto l1; }
```

Konkretisierung Die realisierbaren *Variablenbelegungen* $RENV_l$ werden in $\tau^*_{max}(v)$ durch ihre *konservative* Approximation $RENV_l^*$ ersetzt:

Definition 4.3.38 (*Konservative* Maximalverwendung)
Sei ein SPDS S *und eine* SPDS-*Variable* $v \in Vars(S)$ *gegeben. Dann ist*

$$\begin{aligned}\tau_{max}(v) \quad := \quad & \max(\{[\![v]\!]_{env} \mid l \in Marken(S), used_l^{read}(v), env \in RENV_l^*\} \\ & \cup\{[\![e]\!]_{env} \mid \text{„}l : v = e; \text{“} \in Stats(S), env \in RENV_l^*\})\end{aligned}$$

die konservative Maximalverwendung *von* v.

Eigenschaften des Modellanalysebeispiels Für das *Modellanalyse*beispiel $\tau_{max}(v)$ übertragen sich wegen der verwendeten *Wertebereichsanalyse* die Eigenschaften und Komplexitäten. So ist sie flusssensitiv, interprozedural, *konservativ*, entscheidbar, kontextinsensitiv sowie pfadinsensitiv. Zumindest die *Konservativität* bedarf dabei besonderer Beachtung:

Lemma 4.3.23 (*Konservativität* von $\tau_{max}(v)$)
Sei ein SPDS S *gegeben. Dann gilt* $\forall v \in Vars(S) : \tau_{max}(v) \geq \tau^*_{max}(v)$.

Beweis Folgerung aus $RENV_l \subseteq RENV_l^*$.

□

Daher wird im Folgenden für τ^*_{max} die *konservative* Approximation τ_{max} eingesetzt.

Abstrakte Informationen des Motivationsbeispiels In *Quellcode* 4.19 sind die *Modellanalyse*-ergebnisse von $values_l$ in den Kommentaren zu sehen. Die *Reihungsvariablen* sind zu $a*$ zusammengefasst. Unter Beachtung der schreibenden und lesenden *SPDS*-Variablen-*Verwendungen* ergibt sich dann $\tau^*_{max}(a*) = \tau_{max}(a*) = 1, \tau^*_{max}(i) = \tau_{max}(i) = 99$ und $\tau^*_{max}(p) = \tau_{max}(p) = 49$. Da *choose* an der Marke $m0$ nicht stets den Wert 0 liefert, sind alle diese *Maximalverwendungen* im Beispiel auch realisierbar. Daher ist τ_{max} für das Beispiel exakt ($\tau_{max} = \tau^*_{max}$). Der Parameter j ist dabei keine *SPDS*-Variable, weswegen es keine abstrakte Information zur *konservativen Maximalverwendung* für diesen gibt.

4.3.5.5. Wertebereichsreduktion T_{min}

Nun wird die eigentliche Modellreduktion beschrieben.

Quellcode 4.20: Typminimierung mittels *Modelltransformation* 4.31.

```
int a(1)[100];
init m;

void m() {
m0: A j (0,99) a[j]=choose(0,1);
m1: heapify(99);
m2: return; }

void heapify(int i(7)) {
      int p(6);
l0:   if (i>0) heapify(i-1);
l1:   p = (i-1)/2;
l2:   if (a[p] >= a[i]) return;
l3:   a[p] = a[i], a[i] = a[p];
l4:   i = p;
l5:   if (i>0) goto l1; }
```

Idee Die im *SPDS* definierten Typen werden so weit minimiert, wie es die bestimmte *konservative Maximalverwendung* zulässt.

Konkretisierung Da nur $\tau_{max}(v) > 0$ vereinfachend betrachtet wird, müssen (falls nötig) die Ausdrucksvereinfachung aus Abschnitt 4.3.1 ab Seite 45 und die Variablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90 zuvor angewendet werden. Für eine Variable $v \in Vars(S)$ eines *SPDS* S ist dann $bits(\tau_{max}(v)) = \lceil log_2(1 + \tau_{max}(v)) \rceil$ der kleinste mögliche Typ für den Ausdruck $\tau_{max}(v) \in \mathbb{N}_{\geq 0}$ (siehe Typ eines Ausdrucks in Definition 3.2.3 auf Seite 26).

Definition 4.3.39 (*Wertebereichsreduktion* T_{min})
Sei $V = Vgbl$ oder $V = Vlcl_p$ oder $V = Param_p$ die Definitionsliste globaler, lokaler oder Parametervariablen des SPDS *S und sei eine* temporale Formel *ϕ oder eine Menge an Marken $L0$ gegeben. Dann besteht die* Wertebereichsreduktion *T_{min} aus folgender* Modelltransformation*, welche so oft wie möglich für jede Wahl von V angewendet wird und dabei die Typen jeder Variablen $v \in V$ durch den minimalen Typen $b_{min,v} := bits(\tau_{max}(v))$ ersetzt, falls möglich:*

$$\frac{v = (\alpha, b) \in V \quad b_{min,v} < b \quad \alpha \notin \phi}{V \;\Rrightarrow\; V[(\alpha, b_{min,v})/v]} \tag{4.31}$$

Wobei $(\alpha \notin \phi) \equiv true$ ist, wenn statt einer temporalen Formel *ϕ eine Menge an Marken $L0$ gegeben ist.*

Modelloptimierung des Motivationsbeispiels Wegen $\tau_{max}(a*) = 1, \tau_{max}(i) = 99$ bzw. $\tau_{max}(p) = 49$ sind die minimalen Typen $b_{min,v}$ jeweils $1, 7$ bzw. 6. Die *Modelltransformation* 4.31 ändert daher die Typen entsprechend, wie *Quellcode* 4.20 veranschaulicht.
Der *Konfigurationenraum* gemessen in der Anzahl der Köpfe reduziert sich dabei von $2^{800} * 4 + 2^{800} * 6 * 2^8 * 2^8 = 3 \cdot 10^{246}$ auf $2^{100} * 4 + 2^{100} * 6 * 2^6 * 2^7 = 6 \cdot 10^{34}$.

Bemerkung 4.3.5 *Insbesondere die Forderung $\alpha \notin \phi$ ist wichtig in Definition 4.3.39, wie die LTL-X Formel $\psi =$ „$F\neg(p = 255)$“ zeigt. So ist ψ in heapify-Beispiel S nicht erfüllt, weil es einen Pfad gibt, welcher zu uninitialisiertem p bei Prozedureintritt von $heapify$ an der Marke $l0$ führt. Dort hat p in S möglicherweise den Wert* 255 *und somit ist $S \not\models \psi$. In $T_{min}(S)$ kann der Wert* 255 *hingegen nicht mehr realisiert werden, da der Typ von p nicht ausreicht zur Darstellung dieses Wertes. Entsprechend gilt dann $T_{min}(S) \models \psi$. Die* Wertebereichsreduktion *T_{min} wäre ohne diese Forderung nicht einmal LTL-X-invariant (und damit auch nicht invariant bezüglich der anderen aufgeführten Logiken).*

Wohldefiniertheit Eine Typminimierung gemäß *Modelltransformationsregel* 4.31 kann höchstens einmal je definierter *SPDS*-Variable v aus der *SPDS*-Beschreibung (global oder lokal) erfolgen, da der Minimaltyp $b_{min,v}$ durch $\tau_{max}(v)$ eindeutig bestimmt und fest ist. Aus der Endlichkeit der *SPDS*-Variablenmenge folgt dann die höchstens endliche Anwendbarkeit von *Modelltransformationsregel* 4.31. Die *Wertebereichsreduktion* T_{min} aus Definition 4.3.39 ist daher wohldefiniert.

Korrektheit (Invarianz Temporaler Formeln und Erreichbarkeit)

Satz 4.3.29 (Korrektheit der *Wertebereichsreduktion* T_{min})
Die Wertebereichsreduktion T_{min} *ist CTL*-invariant für gegebenes* $\phi \in CTL^*$ *und somit auch* α*-invariant für gegebenes* $\phi \in \alpha$ *und ein beliebiges* $\alpha \in \{ACTL, ACTL\text{-}X, ACTL^*, ACTL^*\text{-}X, CTL, CTL\text{-}X, CTL^*, CTL^*\text{-}X, LTL, LTL\text{-}X\}$ *sowie* Erreichbarkeits-invariant *für a priori gegebene Marken* $L0$.

Der Beweis findet sich im Anhang B ab Seite 145.

4.3.5.6. Eigenschaften

Obwohl die *Wertebereichsreduktion* T_{min} nur minimale Änderungen am *SPDS* S durchführt, ist ihr Einfluss auf den *Konfigurationenraum* groß, wie das Beispiel verdeutlicht. Eine wichtige Beobachtung ist folgende. Variablen, welche verwendet werden bevor ihnen ein Wert zugewiesen wird, können mittels τ_{max} nicht optimiert werden, wie dieses Beispiel zeigt:

```
void test() {
    int i(8);
    l: if (i < 10) goto l; }
```

Da die Variable i an der Marke l uninitialisiert verwendet wird, ist $\tau_{max}(i) = 255$. Würde die Variable i dennoch auf z.B. 2 Bit im Typ minimiert, so würde dies die *Erreichbarkeit* beeinflussen indem i nur noch Werte < 10 realisieren würde. Dabei kann die Konstante auch z.B. durch ein *choose* ausgedrückt sein, was für statische *Modellanalyse*n das Treffen von Aussagen erschwert.

Berechenbarkeit einer optimalen Wertebereichsreduktion Für *SPDS* ist eine *optimale* Wertebereichsreduktion berechenbar. Dabei kann diese Optimalität wie folgt formuliert werden:

Definition 4.3.40 (*Optimale Wertebereichsreduktion*)
Eine Wertebereichsreduktion T *heißt* lokal präzise *bezüglich einer* Maximalverwendung $\tau_{max} : Vars(S) \to \mathbb{N}_{\geq 0}$, *wenn* T *genau dann die Typen jeder Variablen* v *durch den minimalen Typen* $b_{min,v} := bits(\tau_{max}(v))$ *ersetzt, falls dies keinen Einfluss auf das* Modellprüfung*sresultat hat.* T *heißt* global präzise, *wenn* T *lokal präzise und* $\sum_{v \in Vars} b_{min,v}$ *minimal ist. Ist* T *global* präzise *und* τ_{max} *exakt, so heißt* T optimal.

Zur Verdeutlichung des Unterschieds zwischen lokal und global *präzise*r *Wertebereichsreduktion* diene folgendes Beispiel:

Beispiel 4.3.7 *Zu folgendem* SPDS S *mit Hauptprozedur main sei die Formel* $\phi = F(x = 3 \vee y = 6)$ *gegeben:*

```
void main() {
      int x(8);
      int y(8);
  m0: x=1, y=1;
  m1: return;  }
```

Es ist $S \models \phi$, *da es eine* Initialkonfiguration *gibt, so dass* x *und* y *entsprechend belegt werden können (z.B.* $x = 0$ *und* $y = 6$*). Eine exakte* Maximalverwendung *liefert* $\tau_{max}(x) = \tau_{max}(y) = 1$. *Entsprechend sind die Minimaltypen für* x *und* y *bestimmt als* $b_{min,x} = b_{min,y} = 1$. *Nun gibt es verschiedene Möglichkeiten, ohne Beeinflussung von* ϕ *die Typen zu minimieren. Einerseits ist*

dann T_1 mit $bits(x) = 1$ und $bits(y) = 3$ eine lokal präzise *Transformation, da gilt $(S \models \phi) \Leftrightarrow (T_1(S) \models \phi)$. Denn ϕ ist auch in $T_1(S)$ gültig mit der initialen* Variablenbelegung $x = 0$ *und* $y = 6$. *Andererseits ist T_2 mit $bits(x) = 2$ und $bits(y) = 1$ ebenfalls eine lokal* präzise *Transformation, da auch hier für die initialen* Variablenbelegung $x = 3$ *und* $y = 0$ *die Formel ϕ erfüllt ist. Dennoch ist die Summe der verwendeten Bits bei T_2 letztlich geringer, so dass T_1 nur lokal* präzise*, nicht jedoch global präzise ist. Da es in T_2 nicht möglich ist, den Typ von x weiter zu verringern ohne das weiterhin ϕ gilt, ist T_2 auch global* präzise *(im Gegensatz zu T_1).*

Die vorgestellte *Wertebereichsreduktion T_{min}* ist demnach genau dann lokal *präzise*, wenn die gegebene *temporale Formel ϕ* keine *SPDS*-Variable $v = (\alpha, b) \in Vars$ enthält, für die $b_{min} < b$ gilt. Somit stellt die *Wertebereichsreduktion T_{min}* eine *konservative* Approximation dar und ist insbesondere i.A. nicht *optimal*. Dennoch ist sie automatisch global *präzise*, wenn sie bereits lokal *präzise* ist:

Lemma 4.3.24 (Lokal *präzise* $\Rightarrow$ global präzise)
Die Wertebereichsreduktion *T_{min} ist global* präzise*, wenn sie lokal präzise ist.*

Beweis(Skizze) Dies gilt wegen Abwesenheit von temporalen Einflüssen bei lokaler Präzision, denn es ist $\sum_{v \in Vars} b_{min,v}$ bereits bei lokaler Präzision minimal.

□

Obgleich die vorgestellte *Wertebereichsreduktion T_{min}* nicht *optimal* ist, kann sie optimal durchgeführt werden, wie folgender Satz zeigt.

Satz 4.3.30 (Berechnenbarkeit einer *optimal*en *Wertebereichsreduktion*)
Eine optimal*e* Wertebereichsreduktion *ist berechenbar.*

Beweis (Skizze) Sei τ_{max} exakt (dies ist möglich wegen der Berechenbarkeit von $RENV_l$) und $b_{min,v} < bits(v) = b$. Dann ist die *Modelltransformation* $T_{b^*,v}$ jene, die den Typ von $v = (\alpha, b)$ auf $b^* \in \mathbb{N}_{\geq 0}$ ändert: $V \Rrightarrow V[(\alpha, b^*)/v]$ für $V = Vgbl$ bzw. $V = Vlcl_p$ bzw. $V = Param_p$ je nachdem wo v definiert ist ($v \in Vgbl$ bzw. $v \in Vlcl_p$ bzw. $v \in Param_p$). Nun werden sämtliche mögliche neue Typen $B = \{\beta \in \mathbb{N}_{\geq 0} \mid b_{min,v} \geq \beta < b\}$ für v betrachtet. Für jeden dieser Typen kann dann eine *Modellprüfung* durchgeführt werden und unter jenen Typen, wo sich das Modellprüfungsresultat nicht ändert, wird der kleinste gewählt: $b^* := \min\{\beta \in B \mid (S \models \phi) \Leftrightarrow (T_{b^*,v}(S) \models \phi)\}$. Bei Betrachtung aller *SPDS*-Variablen auf diese Weise kann $\sum_{v \in Vars} b_{min,v}$ minimiert werden und es folgt die Behauptung. So werden alle möglichen *SPDS*-Variablen auf ihre insgesamt minimalen Typen reduziert ohne Beeinflussung des *Modellprüfung*sresultats, was bedeutet, dass diese *Wertebereichsreduktion optimal* ist. Daher ist sie berechenbar (und damit auch entscheidbar).

□

Satz 4.3.31 (Komplexität einer *optimal*en *Wertebereichsreduktion*)
Die Komplexität einer optimal*en* Wertebereichsreduktion *übersteigt die der* Modellprüfung *auf* Erreichbarkeit*.*

Beweis Sei ein beliebiges *SPDS S* auf *Erreichbarkeit* einer Marke $l0 \in Marken(S)$ zu prüfen. Dann wird eine neue globale Variable $r \notin Vars(S)$ in das *SPDS* eingeführt mit Typ $bits(r) = 1$ ein und ändere den *Konfigurationenübergang* an der Marke $l0$ ab zu „$l0 : r = 1;$“ zu einem *SPDS* S'. Dies ändert die *Erreichbarkeit* der Marke $l0$ in S nicht, so dass $S \rightsquigarrow l0 \Leftrightarrow S' \rightsquigarrow l0$. Nun gibt es zwei Möglichkeiten, wie eine *optimal*e *Wertebereichsreduktion* T in S' den neuen minimalen Typ für r wählt. Einerseits kann r den Typ 0 besitzen. Dann ist aber die *SPDS*-Marke $l0$ nicht *erreichbar*, da es andernfalls zu einer Versackung durch zu kleinem Typ bei der *Zuweisung* $r = 1$ kommt und die Transformation $T_{0,r}$ ist *optimal*. Andererseits kann r weiterhin den Typ 1 besitzen. Dann war $l0$ in S' *erreichbar* und die Transformation $T_{1,r}$ ist *optimal*. Insgesamt gilt damit $(S \rightsquigarrow l0) \Leftrightarrow (S' \rightsquigarrow l0) \Leftrightarrow (T_{1,r}$ ist *optimal*$)$. Somit entscheidet die *optimal*e *Wertebereichsreduktion* über die *Erreichbarkeit* im *SPDS S* (Reduktion).

□

Fehlalarme Wegen temporaler Invarianz der vorgestellten *Wertebereichsreduktion* T_{min} sind Fehlalarme ausgeschlossen.

Konfigurationenraumveränderung Der *Konfigurationenraum* wird stark beeinflusst und stets kleiner. Dies ist abhängig vom *SPDS* und reduziert diesen im Falle der Typverringerung von b auf $b' < b$ Bits einer ...

- globalen Variable v um den Faktor $2^{b'-b}$,
- Parameter oder lokalen Variable $v \in Vlcl_p \cup Param_p$ um den Faktor $(2^{b'-b})^k$,

wobei k die absolute Häufigkeit der *Prozeduraufrufe* p im Keller einer *Konfiguration* darstellt. Die Kellertiefe ist unbeschränkt, so dass es im zweiten Fall theoretisch um eine Reduktion um einen unbeschränkt großen Faktor handelt. Die Anzahl der Köpfe ändert sich dennoch nur um den endlichen Faktor $2^{b'-b}$.

Komplexität Das *Modellanalyse*beispiel zur *konservativen* Bestimmung von *Maximalverwendungen* τ_{max} basiert auf der *Wertebereichsanalyse*. Diese kann in Zeit $O(|Marken(S)| \cdot |Vars| \cdot |ENV| \cdot j \cdot k_{max} \cdot r \cdot t)$ durchgeführt werden (Parameter siehe Abschnitt 4.3.2.2 auf Seite 70)
Die Transformationskomplexität beträgt $O(|Vars| \cdot |\phi|)$, da maximal so oft der Typ von Variablen verändert wird, wie es Variablen gibt und die Vorbedingungen der Eigenschaften einfach zu prüfen sind (letzteres geht in $O(|\phi|)$).

Aufwand vs. Nutzen Der Aufwand wird durch die *konservativ* Approximierten Wertebreiche $RENV_l^*$ bestimmt. Diese *Modellanalyse*ergebnisse können allerdings aus anderen *Modelltransformationen* wie z.B. der Variablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90 wieder verwendet werden. Diese verändern das Modell nicht stark, so dass die Analyseergebnisse auch hier nicht erneut berechnet werden müssen. Somit reduziert sich die *Wertebereichsreduktion* T_{min} auf die Bestimmung und Ersetzung der Minimaltypen, was effizient implementiert werden kann.

4.3.5.7. Zusammenfassung

Rückblick Im vorigen Abschnitt wurde gezeigt, dass die Größe des Konfigurationenraums stark durch *SPDS*-Variablen beeinträchtigt wird. Allerdings betrachtet die dort beschriebene Variablenelimination nicht die Typen auftretender *SPDS*-Variablen. In der Praxis sind diese im gewählten Modellprüfungsprozess häufig sehr homogen (einheitlich), so dass dies kein Problem für die Variablenelimination darstellt. In diesem Abschnitt wurden die Typen der im Modell verbliebenen *SPDS*-Variablen genauer betrachtet und ein Verfahren entwickelt, um nicht benötigte Bits der *SPDS*-Variablen zu identifizieren und zu eliminieren. An einem *SPDS*-Beispiel wurde verdeutlicht, dass Variablentypen verkleinert werden können, um damit auch den Konfigurationenraum zu verkleinern. Die dazu nötige abstrakte Information der Maximalverwendung ist exakt berechenbar und aufwändiger als die Modellprüfung, so das eine Heuristik mit Wertebereichsanalyse zur Approximation vorgestellt wurde. Eigenschaften und einsetzbare Programmanalysen waren daher für diesen Teil nicht zu untersuchen, da die Wertebereichsanalyse bereits in Abschnitt 4.3.1 auf Seite 45 untersucht wurde und die Maximalverwendung einen Spezialfall davon darstellt (Intervallanalyse genügt). Nachdem die Konservativität des Modellanalysebeispiels gezeigt wurde, wurde diese am *SPDS*-Beispiel veranschaulicht und dann die Wertebereichsreduktion T_{min} formal erklärt. Dabei werden die Typen von *SPDS*-Variablen so minimiert, dass die zu prüfenden temporalen Eigenschaften sich nicht ändern, falls die höheren Bits von *SPDS*-Variablen im *SPDS* nicht benötigt werden. Dieses Verfahren wurde ebenfalls am *SPDS*-Beispiel verdeutlicht. Bei Betrachtung der Optimalität der Wertebereichsreduktion wurde zwischen lokaler und globaler Präzision unterschieden und an einem weiteren *SPDS*-Beispiel veranschaulicht. So folgt aus lokaler Präzision auch die globale für die vorgestellte Wertebereichsreduktion T_{min} und die Optimalität überträgt sich aus der optimalen Maximalverwendung. Analoges ergab sich für die Entscheidbarkeit, Komplexität sowie Eigenschaften wie flusssensitiv, interprozedural, konservativ, entscheidbar, kontextinsensitiv und den Aufwand

etc. Aussagen temporaler Formeln bleiben bei der vorgestellten Wertebereichsreduktion T_{min} erhalten, obgleich sich der Konfigurationenraum in diesem Fall verändert. Der Aufwand wird auch hier wieder durch die Wertebereichanalyse bestimmt und hängt von der konkreten Realisierung (Fixpunktverfahren) ab.

Fazit Mit der vorgestellten Wertebereichsreduktion T_{min} ist es nun auch möglich, Variablentypen in *SPDS* zu verkleinern. Dadurch werden große Wertebereiche von *SPDS*-Variablen auf kleinere abstrahiert mittels einer präzisen Abstraktion. Dies beschleunigt die Modellprüfung durch Verkleinerung des Zustandsraums und die Präzision anderer Modellanalysen wie z.B. die Wertebereichanalyse (da dieser konservativen Approximation engere Grenzen durch kleinere Variablentypen gesetzt sind). Im F-Soft-Projekt gibt es für einfachere Modelle einen ähnlichen Ansatz [6]. Dieser operiert jedoch auf *Quellcode*ebene und nicht im Modell und erlaubt nur die Analyse von einfachen Intervallgrenzen (Minimum sowie Maximum) statt des gesamten *Wertebereich*s wie in der vorliegenden Arbeit. Neben der vorliegenden Arbeit ist in der betrachteten Literatur F-Soft das einzige Projekt [6], wo eine *Wertebereichsreduktion* mittels einer *Wertebereichsanalyse* (engl. Range-Analysis) in Modellen vorgenommen wird. Insbesondere für *SPDS* bei Berücksichtigung temporaler Formeln ist dies in der betrachteten Literatur nicht untersucht. Erst in dieser Arbeit wurde ein entsprechendes Verfahren entwickelt sowie Aussagen zu Existenz und Komplexität einer optimalen Wertebereichsreduktion getroffen.

4.4. Modellreduktionsreihenfolge

Im folgenden Abschnitt wird der Modellreduktionsprozess erläutert. Die dabei gewählte Reihenfolge der Modellreduktionen wird erläutert und begründet, da diese sich teilweise gegenseitig beeinflussen. Zur Auswahl stehen die Modellreduktionen:

- **A**usdrucksvereinfachung aus Abschnitt 4.3.1 ab Seite 45
- **P**rozedurüberbrückung aus Abschnitt 4.3.3 ab Seite 78
- **M**odellbeschneidung aus Abschnitt 4.3.2 ab Seite 68
- **V**ariablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90 und
- **W**ertebereichsreduktion aus Abschnitt 4.3.5 ab Seite 112.

Zur Abkürzung einer Modellreduktion wird im Folgenden jeweils der Anfangsbuchstabe verwendet. Eine Überschrift X < Y bedeute dann, dass die Modellanalyse mit Anfangsbuchstabe X vor der Modellanalyse mit Anfangsbuchstabe Y durchgeführt wird. Im jeweils darauf folgenden Abschnitt einer solchen Überschrift wird diese Wahl begründet, obgleich nicht auf jeden Aspekt der gegenseitigen Beeinflussung an dieser Stelle Rücksicht genommen werden konnte. Eine gravierende Einschränkung stellt das allerdings nicht dar, da die Modellreduktionen wiederholt angewendet werden, solange sich noch Optimierungen am *SPDS* vornehmen lassen.

A < M Die Ausdrucksvereinfachung sollte vor der Modellbeschneidung erfolgen, damit das Slicing-Ergebnis besser wird. Je einfacher Ausdrücke sind, desto wahrscheinlicher ist es, dass komplette Verzweigungen im Kontrollfluss für das Slicing nicht berücksichtigt werden müssen. Dies ist der Fall, wenn Bedingungen in bedingten Verzweigungen stark zu 0 (bzw. false) oder 1 (bzw. true) ausgewertet werden konnten. Dann ist z.B. das Vorwärts-Slicing in der Lage, den nicht relevanten Teil der Verzweigung zu ignorieren, was zu präziserem Slicing führt und der Modellbeschneidung ggf. ermöglicht, das *SPDS*-Modell besser zu beschneiden. Vereinfachte *Ausdrücke* wirken sich nicht nur positiv auf die *Modellprüfung* aus, sondern auch auf die anderen *Modellanalyse*n. So sind *Modellanalyse*ergebnisse auch im Allgemeinen potentiell um so *präzise*r, je einfacher auftretende *Ausdrücke* sind. Andererseits ermöglichen die am wenigsten gravierenden Eingriffe am Anfang der Reihenfolge eine bessere Wiederverwendung der Modellanalyseergebnisse. So greift die Ausdrucksvereinfachung gar nicht in den Konfigurationenraum ein und ermöglicht damit eine sehr gute Wiederverwendung

der Modellanalyseergebnisse, was Analysezeit für die Modellanalysen spart. Daher wurde die Ausdrucksvereinfachung als erste Modellreduktion gewählt (A < M, A < P, A < V, A < W).

P < M Auch die Prozedurüberbrückung sollte vor der Modellbeschneidung erfolgen, weil überbrückte Prozeduren ggf. tot bzw. unerreichbar werden. Slicing kann dies erkennen und die Prozedur ggf. komplett aus dem *SPDS* eliminieren. Dies ist z.B. häufig bei Bibliotheksaufrufen der Fall, wenn unnötige Bibliotheksfunktionen in das *SPDS*-Modell überführt werden. Nach der Modellbeschneidung kann es allerdings sinnvoll sein, die Prozedurüberbrückung erneut einzusetzen, weil durch die Beschneidung eine Prozedur ggf. erst überbrückbar wird, wenn Teile einer anderen Prozedur „weggeschnitten“ wurden. Für die Qualität des Slicings ist es sinnvoller, weniger in Prozeduren zu verzweigen. Damit ist dann die Modellbeschneidung in der Lage, das *SPDS* mehr zu beschneiden, als wenn die Prozedurüberbrückung vor der Modellbeschneidung durchgeführt wird. Um nach der Modellbeschneidung trotzdem wieder überbrückbare Prozeduren zu finden, werden alle Modellreduktionen nach einem Durchlauf so lange wiederholt, wie das *SPDS* verbessert werden kann. Beim Überbrücken einer Prozedur ist ein Seiteneffekt ausgeschlossen, so dass Modellanalyseergebnisse konservativ wiederverwendet werden können (bei einem erneuten Durchlauf werden diese dann ggf. präziser).

M < V Die Variablenredundanzelimination sollte erst nach der Modellbeschneidung erfolgen (Reihenfolge MV), da es wenig Sinn macht, den Redundanzraum für unwichtige (beschneidbare) *SPDS*-Variablen zu betrachten. Würde die Variablenredundanzelimination vor der Modellbeschneidung (Reihenfolge VM) erfolgen, so würden einerseits zusätzliche unnötige *SPDS*-Variablen das ILP vergrößern und damit das Lösen des ILPs erschweren. Andererseits könnten die zusätzlichen beschneidbaren *SPDS*-Variablen im *SPDS* zu einer ungünstigen Verlagerung des Redundanzraums führen, so dass zuvor beschneidbare *SPDS*-Variablen nicht mehr beschneidbar werden. Dies ist z.B. dann der Fall, wenn die Minimallösung (optimale Repräsentanten) durch eigentlich beschneidbare *SPDS*-Variablen derart beeinträchtigt wird, dass sie sich vergrößert. Dann verbleiben bei der Reihenfolge VM mehr *SPDS*-Variablen im Restmodell als bei der Reihenfolge MV. Die Reihenfolge MV ist dann besser geeignet, um den Konfigurationenraum für *SPDS* zu verkleinern, und wurde daher für den Modellreduktionsprozess gewählt.

V < W Für die Variablenredundanzelimination sind homogene Typen besser, da *SPDS*-Variablen bei ihrer Verwendung untereinander dann besser ausgetauscht (Repräsentantenprägung) und Äquivalenzen einfacher erkannt werden können. Im gewählten Modellprüfungsprozess mit JMoped und Moped z.B. sind auftretende Typen oft sehr homogen, da diese durch einen Benutzer als Modellierungsparameter vorgegeben werden. Die Äquivalenzanalyse ist auf einheitlichen Typen einfacher und muss bei gleichem Typ keine Typkonflikte und mögliche Anpassungen beachten. Würde die Wertebereichsreduktion vor der Variablenredundanzelimination erfolgen (Reihenfolge WV), so hätten manche Variablen einen deutlich geringeren Typ als andere. *SPDS*-Variablen kleineren Typs können dann nicht als Repräsentanten für größere *SPDS*-Variablenwerte verwendet werden. D.h. dass sie dann nicht belegungsäquivalent gegeneinander ausgetauscht werden könnten, was zu potentiell weniger erkennbarer Belegungsäquivalenz für die Äquivalenzanalyse führt. Dies wiederum beeinträchtigt potentiell die Wahl der Repräsentanten für die Variablenredundanzelimination und könnte u.U. zu mehr Repräsentanten führen, als wenn erst die Variablenredundanzelimination und dann die Wertebereichsreduktion (Reihenfolge VW) durchgeführt wird. Aus diesem Grund wurde für den Modellreduktionsprozess die Reihenfolge VW gewählt.

Fazit Somit ergibt sich aus den voran gegangenen Betrachtungen die Modellreduktionsreihenfolge A < P < M < V < W, wie in der Aufzählung zu Beginn das Abschnitts aufgeführt.

5. Experimente

In diesem Kapitel wird der Nutzen der vorgestellten Methoden in der Praxis untersucht. Im Rahmen der gewählten Modellprüfkette[1] mit dem Modellprüfer Moped können neben temporalen Bedingungen[2] z.B. auch (nicht korrekt behandelte) *Ausnahmen* (*Exceptions*), Speicherüberläufe, *Zusicherungen* (*Assertions*) und andere Fehler **automatisch** überprüft werden. Beispielhaft für die Experimente wurde das Prüfen von Zusicherungen betrachtet. Denn der Gebrauch von Zusicherungen gewinnt in der Praxis höherer Programmiersprachen stetig an Bedeutung, da Zusicherungen in der Praxis gut geeignet sind, um Fehler in Programmen zu finden [202] (sie weisen gezielt auf konkrete Stellen im Programm hin, wo eine Zusicherung fehlschlägt).

5.1. Eingriff in die gewählte Modellprüfkette von Moped

Wie Abbildung 5.1 zeigt, wurde im Rahmen dieser Arbeit für *SPDS*' (welche durch die Modellbeschreibungssprache *Remopla* [219, 218] beschrieben werden) das Werkzeug *HalSPSI* entwickelt, um Modelle zu optimieren. Dabei wurde nicht der Modellprüfer geändert, sondern nur die Modelle analysiert und transformiert. Es handelt sich daher um einen transparenten Eingriff. Die Modellreduktionen für *SPDS* wurden auf Remopla entsprechend übertragen, da sich die Modellbeschreibungssprache Remopla von der aus dieser Arbeit leicht unterscheidet. So gibt es in Remopla z.B. keine Funktion *choose* und keine einfache bedingte Verzweigung. Dafür werden in jedem von JMoped [119, 65, 64] generierten Remopla-Beispiel z.B. interprozedurale Sprünge und Parallelanweisungen verwendet. Die Modellanalysen und Transformationen wurden darauf angepasst und geringfügig modifiziert. Es wurden dazu im Anhang A ab Seite 133 *SPDS*'-Anweisungen (*SPDS*-Abkürzungen) definiert, um möglichst viele Remopla-Sprachkonzepte auf die in dieser Arbeit definierte *SPDS*-Kernsprache abzubilden[3]. Aus bereits erwähnten Vorteilen wurde statt der Interpretation der Nicht-*SPDS*-Kernsprache (z.B. Reihungen) eine direkte Unterstützung für Remopla-Sprachkonzepte implementiert (z.B. Reihungen direkt in den Modellanalysen behandelt).

5.2. Remopla - Beschreibungssprache für SPDS

Für die Experimente wurde zur Beschreibung von *SPDS* die Modellierungsprache Remopla des *Modellprüfer*s Moped in einer bisher unveröffentlichten Entwicklerversion[4] basierend auf Version 2 verwendet. Damit können Java Programme bis zur Version 6[5] verifiziert werden. Seit JMoped 2.0 (aktuell 2.0.2) ist auch beschränkte Parallelität in Java-Programmen mittels Kontextbeschränkung erlaubt [223]. Leider gibt es keine formale Semantikbeschreibung für Remopla (Nachfrage bei den

[1] siehe Anlage auf CD
[2] LTL auf *PDS*mit Moped Version 1.0
[3] Dazu gehört u.a. auch die Erweiterung um interprozedurale Verzweigungen (siehe Anlage auf beiliegender CD)
[4] Die Versionen der Werkzeuge befinden sich auf der beiliegenden CD. Sie wurden in Zusammenarbeit mit den Entwicklern der Werkzeuge JMoped und Moped zur Verfügung gestellt.
[5] Die offiziellen Veröffentlichungen unterstützen Java bis zur Version 5.0.

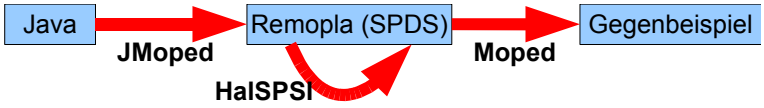

Abbildung 5.1.: Eingriff in die Modellprüfkette von Moped mittels *HalSPSI*

Entwicklern). Für einen formalen Korrektheitsnachweis der Modellanalysen und -Reduktionen war eine informale und unvollständige Semantikbeschreibung [219] unzureichend. Daher wurde eine formale Semantik für *SPDS* definiert (siehe Abschnitt 3.2 ab Seite 23). Eine vollständige Syntaxbeschreibung der Sprache Remopla in Form einer Grammatik findet sich als Anlage auf beiliegender CD. Die Unvollständigkeit und die Unklarheiten der bisherigen informalen Beschreibung für Remopla [219] liegen u.a. in folgenden Punkten:

- Schlüsselwort **init** (anders erwartetes unspezifiziertes Verhalten bei Angabe mehrerer *Remoplaprogrammpunkte*)
- Bedeutung von Variablen mit gefolgtem ' und " (Unterscheidung verschiedener Stackrahmen bei *Prozeduraufruf*/Rückkehr)
- negative Variablenbelegungen (in Doku nicht erwähnt; in Syntax erlaubt; durch Moped derzeit nicht unterstützt)[6]
- Prozedurdefinition innerhalb anderer Prozeduren (Speicherzugriffsfehler, sobald innere Prozeduren verwendet werden, die laut Syntax erlaubt sind)
- Verhalten bei Überlauf/Unterlauf sowie Division durch Null usw.

Die folgenden Einschränkungen in Remopla wurden daher zur Vereinfachung getroffen und damit in den Experimenten nur eine Teilsprache von Remopla berücksichtigt (im Folgenden mit ***Remopla2*** bezeichnet).

Einschränkung	**Grund**
keine Strukturen	nicht verwendet in den Benchmarks, triviale Simulation via Variablen möglich und bisher nur globale Typdeklarationen von Moped unterstützt
keine Prozedurdefinitionen innerhalb von Prozeduren	nicht von Moped unterstützt
nur Variablenbelegungen ≥ 0	nicht von Moped unterstützt
keine ENUMs	nicht verwendet in den Benchmarks, triviale Simulation mittels Werte möglich und bisher nur globale Typdeklarationen von Moped unterstützt

Eine Grammatik zu der von mir untersuchten Teilsprache Remopla2 findet sich ebenfalls als Anlage auf der beiliegenden CD.

5.3. Rahmenbedingungen

Als Grundlage für die Experimente dienten 150 Remopla-Modelle. Diese wurden mittels JMoped aus 30 Java-Beispielen gewonnen, welche zum größten Teil zu den Benchmark-Instanzen der Werkzeuge JMoped und Moped gehören. Zur Modellgenerierung der Remopla-Beispiele wurden jeweils 5 unterschiedliche Bitbreiten (4 bis 8 Bit) zur Modellierung von Ganzzahlen (Integer) verwendet, so dass sich aus den 30 Java-Beispielen 150 Remopla-Modelle ergeben. Aufgabe für den Modellprüfer Moped war es, zu diesen Modellen jeweils vor und nach der Modellreduktion das Fehlschlagen von Java-Zusicherungen (Assertions) zu prüfen. Formal bedeutet dies, dass der Wahrheitsgehalt der temporalen Formel „$\neg EF\ assert(false)$“ zu bestimmen ist[7].
Durchgeführt wurden die Experimente mit einem AMD 64 X2 4200+ mit 2 GB Hauptspeicher unter Linux (Kernel 2.6.27). Die in dieser Arbeit beschriebenen Modelltransformationen wurden auf Remopla übertragen und in dem Werkzeug *HalSPSI* implementiert. Dabei wurden die beiden Modellanalysebeispiele Wertebereichsanalyse und Äquivalenzanalyse aus der Anlage auf beiliegender CD verwendet, um folgende Modellreduktionen zu realisieren:

[6] Man beachte an dieser Stelle, dass Zwischenergebnisse beim Berechnen einer *Nachfolgekonfiguration* sehr wohl negativ sein dürfen.

[7] Dies ist eine Kurzschreibweise für „$\neg EF\ (l_{assert} \Rightarrow (x == 0))$“ für die Prozedur „$void\ assert(int(1)\ x)$“ mit Anfangsmarke $l_{assert} \in Marken(assert)$ und wurde in den Experimenten auf die Prüfung von Erreichbarkeit zurückgeführt.

- Ausdrucksvereinfachung aus Abschnitt 4.3.1 ab Seite 45
- Prozedurüberbrückung aus Abschnitt 4.3.3 ab Seite 78
- Modellbeschneidung aus Abschnitt 4.3.2 ab Seite 68
- Variablenredundanzelimination aus Abschnitt 4.3.4 ab Seite 90
- Wertebereichsreduktion aus Abschnitt 4.3.5 ab Seite 112
- Stotterreduktion[8] aus [67].

Diese Modellreduktionen werden in genau dieser Reihenfolge angewendet (Begründung siehe Abschnitt 4.4 auf Seite 120) und so lange wiederholt, wie das *SPDS* verbessert werden kann. Um dabei das grundlegende Prinzip und den praktischen Nutzen der beschriebenen Modelltransformationen darzustellen, wurde das Seiteneffektanalysebeispiel der Prozedurüberbrückung konservativ implementiert. Die beschriebene Modelltransformation ist davon funktionell nicht betroffen und verdeutlicht weiterhin das Resultat der Modellreduktion. Bessere Analyseergebnisse der Modellanalysebeispiele führen lediglich zu potentiell mehr Modelltransformationen.
So wurde für das Seiteneffektanalysebeispiel der Prozedurüberbrückung ein Seiteneffekt dann schon konservativ angenommen, wenn es in der Prozedur einen interprozeduralen oder intraprozeduralen Sprung gibt oder einen Prozeduraufruf oder eine Zuweisung an eine globale Variable. Nach ersten Experimenten wurde eine Verfeinerung dieser Modellanalyse zu der im Kapitel 4.3.3 ab Seite 78 vorgeschlagenen Modellanalysen zu Gunsten anderer (bedeutsameren) Implementierungen nicht weiter betrachtet.
Nach einem Update des Modellprüfers Moped zur Unterstützung von Java-Ringarithmetik wurden sehr häufig Modulo-Operatoren in den automatisch generierten Remopla-Modellen verwendet. Diese erschweren oder verhindern die Ausdrucksvereinfachung mit Maxima deutlich wegen der Abbildung auf die Division und damit verbundenen Problemen beim Kürzen von Brüchen (siehe Erläuterungen zu Abschnitt 4.3.1.4 auf Seite 54). Die Verwendung des Computeralgebrasystems Maxima in der Ausdrucksvereinfachung wurde daher deaktiviert, da sich der SMT-Löser YICES als besser geeignet und deutlich effizienter erwiesen hat. Auch hier ist die beschriebene Modelltransformation davon funktionell nicht betroffen.

5.4. Ergebnisse

In Tabelle 5.1 wurde die Modellprüfzeit*[9] für den Modellprüfer Moped für unoptimierte und optimierte Modelle miteinander verglichen. Die Zeilen stellen jeweils das Java-Beispiel dar und die Spalten jeweils das Remoplamodell mit Modellierung unterschiedlicher Bitbreiten für ganze Zahlen (4 bis 8 Bit). Die Spaltenmarkierung $^-$ stellt darin die Laufzeit für das unveränderte Originalverfahren mit Moped und $^+$ mit sämtlichen Modellreduktionen dar. Es zeigt sich, dass die Modellreduktionen die Modellprüfzeiten signifikant verringern. Als beste Modellreduktion hat sich die Wertebereichsreduktion herausgestellt. Wie auch die Variablenredundanzelimination reduziert sie den Konfigurationenraum auch aus theoretischer Sicht stark. Daher wurde die Wertebereichsreduktion in den Spalten mit der Markierung „ ’ “ gesondert betrachtet.
Die Zeit zur Analyse und Transformation der Modelle liegt in der Größenordnung von wenigen Sekunden (< 6 s) je Remopla-Modell und beträgt damit nur einen Bruchteil der Modellprüfungszeiten. Die *Modellprüfung* wird durch präzisere *Modellanalyse*n i.d.R. stärker beschleunigt, da sich der *Konfigurationenraum* potentiell stärker verkleinert und die *Modellprüfung* i.d.R. wesentlich aufwändiger ist als die *Modellreduktion*en[10].
Die Gesamtlaufzeit zur Prüfung des ganzen Benchmarks (150 Remopla-Beispiele) beträgt unoptimiert (ohne *HalSPSI*) 49 Stunden (einschließlich Speicherüberläufe und Zeitüberschreitungen;

[8] um in Remopla auch Parallelanweisungen zu nutzen, welche mit *SPDS*-Kernanweisungen nicht möglich sind
[9] * ... einschließlich Modellanalyse und -Reduktion
[10] So führt z.B. der Einsatz des SMT-Lösers *YICES* in den Experimenten insgesamt trotz erhöhter Worst-Case Laufzeit (siehe Abschnitt 4.3.1 auf Seite 45) zu einer Verbesserung des Laufzeitverhaltens vom *Modellprüfer*. Der Gewinn durch bessere *Konfigurationenraum*verkleinerung kompensiert den zusätzlichen Aufwand für *YICES*.

im Folgenden mit *out+* gekennzeichnet) bzw. 5 Stunden (ohne Speicherüberläufe und Zeitüberschreitungen; im Folgenden mit *out−* gekennzeichnet). Bei Einsatz aller Modellreduktionen reduziert sich dies auf $1,95$ Stunden*(*out+*) bzw. $0,8$ Stunden*(*out−*), obwohl dank vermiedener Speicherüberläufe mehr Modellprüfungen durchgeführt werden konnten als zuvor. Dies stellt eine Reduktion der Laufzeit von 96 % (*out+*) bzw. von 84 % (out-) dar. Wird lediglich die Wertebereichsreduktion eingesetzt, so reduziert sich die Gesamtlaufzeit des Benchmarks auf $8,5$ Stunden*(*out+*) bzw. $2,4$ Stunden*(*out−*). Dies stellt eine Reduktion der Laufzeit von 83 % (*out+*) bzw. von 52 % (out-) dar.
Ohne Modellreduktionen treten in 38 von 150 Fällen (25%) Speicherüberläufe auf. D.h. in diesen Fällen ist der Modellprüfer Moped wegen zu großem Konfigurationenraum nicht in der Lage, das Remopla-Modell zu analysieren. Durch Verwendung aller Modellreduktionen reduzieren sich diese um 79 % auf nur noch 8 Speicherüberläufe[11]. Wird lediglich die Wertebereichsreduktion genutzt, so reduzieren sich diese auf 6. Dafür ist das Beispiel $IntBufferTest$ verantwortlich. In diesem Beispiel wirken sich die anderen Modellreduktionen störend auf die Wertebereichsreduktion aus. In den Fällen Fibonacci und RecFib verbessert sich aus gleichem Grund die Laufzeit sogar leicht, wenn nur die Wertebereichsreduktion genutzt wird. Diese 3 Beispiele vergrößern die Gesamtlaufzeit des Benchmarks um 18 Minuten (für alle Modellreduktionen im Vergleich zur Wertebereichsreduktion). Dennoch liefert der Einsatz aller Modellreduktionen im Vergleich zur Wertebereichsreduktion eine Verringerung der Gesamtlaufzeit um weitere 77 % (von $8,5$ auf $1,95$ Stunden; siehe oben). Dies liegt vor allem daran, dass die Laufzeiten dieser 3 Beispiele im Vergleich zu den anderen deutlich geringer sind. Womit die zusätzliche Verwendung aller Reduktionen sinnvoll ist. So ist es besser, bei kleinen Laufzeiten (siehe Gesamtlaufzeit für 4 Bits) lediglich die Wertebereichsreduktion zu verwenden. Bei größeren Modellprüflaufzeiten (vgl. Gesamtlaufzeiten ab 5 Bits) lohnt sich der Einsatz der anderen Modellanalysen, da durch diese zusätzlichen Modellanalysen und -transformationen dann wieder mehr Laufzeit der Modellprüfung eingespart wird. Für sehr kleine Laufzeiten des Modellprüfers (wenige Sekunden) kann es auch vorkommen, dass die Zeit für die Analyse und Transformation der Modelle die eigentliche Zeit für die Modellprüfung übersteigt. Die Effekte der Transformation und insbesondere deren reduzierende Auswirkung auf die Modellgröße führen in den Fällen von großen Konfigurationenräumen i.d.R. zu deutlich kleineren Modellprüfzeiten.
In 31 % (46 von 150) der Fälle kann durch Verwendung aller Modellanalysen bereits das Modellprüfungsergebnis vor der Modellprüfung bestimmt werden. In der Mehrheit liegt dies daran, dass es keine erreichbaren Zusicherungen mehr im Remopla-Restmodell gibt oder bestimmte nötige Variablenbelegungen nicht realisierbar sind (konservativ approximiert). In diesen Fällen erübrigt sich die Anwendung des Modellprüfers Moped, was die Gesamtlaufzeit senkt. In der Tabelle 5.1 sind diese Fälle mit einem $*$ gekennzeichnet. In 26 % (39 von 150) der Fälle kann durch Verwendung der Wertebereichsreduktion bereits das Modellprüfungsergebnis vor der Modellprüfung bestimmt werden.
Hieran wird deutlich, dass mehr und bessere Modellreduktionen die Modelle derart vereinfachen, dass weniger aufwändige Modellprüfungen nötig sind. Wie die Tabelle 5.1 zeigt, ist dies vor allem für größere Bits günstig, da in diesen Fällen der Aufwand für die Modellprüfung stark ansteigt, wohingegen der Aufwand für die Modellanalyse klein ist (wenige Sekunden).
Aus Platzgründen musste leider von einer genauen Darstellung von Einzelergebnissen für die einzelnen Modellreduktionen abgesehen werden. Einzeln betrachtet liefert die Variablenredundanzelimination die zweitbeste Reduktion der Gesamtlaufeit des Benchmarks, die Modellbeschneidung die drittbeste und die Ausdrucksvereinfachung die viertbeste. Durch Kombination der Modellreduktionen ergeben sich allerdings Synergieeffekte, wonach die a priori Anwendung der Ausdrucksvereinfachung und der Prozedurüberbrückung z.B. zu einem deutlich besseren Slicing für die Modellbeschneidung führt. Allerdings sind starke beispielabhängige Unterschiede zu bemerken, wie bereits einige Beispiele in der Tabelle 5.1 andeuten.
Zusammenfassend kann gefolgert werden, dass die vorgestellten Modellreduktionen die Modellprüfung deutlich beschleunigen und in einigen Fällen erst ermöglichen (insbesondere bei größeren Bitbreiten oder komplexeren Modellen).

[11] Wie das Beispiel BubbleSort zeigt, können (wenn auch selten) neue Speicherüberläufe entstehen wegen z.B. ungünstiger OBDD-Variablenreihenfolge.

Tabelle 5.1.: Modellprüfzeiten für Moped (inklusive Laufzeit für Modellanalyse und -Reduktion).

Beispiel	**4Bit-**	**4Bit+**	**4Bit'**	**5Bit-**	**5Bit+**	**5Bit'**	**6Bit-**	**6Bit+**	**6Bit'**	**7Bit-**	**7Bit+**	**7Bit'**	**8Bit-**	**8Bit+**	**8Bit'**
ShortEval	347 s	< 1 s*	< 1 s*	mout	< 1 s*	< 1 s*	mout	< 1 s*	< 1 s*	mout	< 1 s*	< 1 s*	mout	< 1 s*	< 1 s*
While	< 1 s	< 1 s	< 1 s	1 s	< 1 s	< 1 s	2 s	< 1 s	1 s	2 s	1 s	1 s	4 s	1 s	2 s
assertfalse	< 1 s	< 1 s	< 1 s	1 s	< 1 s	2 s	2 s	< 1 s	3 s	5 s	1 s	8 s	26 s	3 s	21 s
deadlock_abstraction	< 1 s	< 1 s*	< 1 s*	1 s	< 1 s*	< 1 s*	1 s	< 1 s*	1 s*	1 s	1 s*	1 s*	4 s	1 s*	2 s
erb	1 s	< 1 s*	< 1 s	1 s	2 s	1 s	1 s	2 s	2 s	1 s	2 s	3 s	1 s	1 s	3 s
erb2	335 s	2 s	6 s	mout	1 s	23 s	mout	2 s	246 s	mout	2 s	3661 s	mout	2 s	tout
false_neg_bits	417 s	1 s*	< 1 s*	mout	1 s*	< 1 s*	mout	2 s*	1 s*	mout	4 s*	3 s*	mout	4 s*	4 s*
false_pos_bits	1 s	1 s	1 s	1 s	2 s	2 s	2 s	3 s	3 s	5 s	8 s	8 s	26 s	30 s	22 s
interfacetest	1 s	< 1 s*	< 1 s*	8 s	1 s*	< 1 s*	34 s	1 s*	< 1 s*	107 s	2 s*	1 s*	315 s	2 s*	2 s*
max_fn	< 1 s	2 s	2 s	1 s	2 s	1 s	1 s	2 s	2 s	1 s	1 s	2 s	1 s	2 s	2 s
native_fp	387 s	1 s*	2 s*	mout	4 s*	4 s*	mout	5 s*	5 s*	mout	5 s*	4 s*	mout	6 s*	6 s*
parseint	1 s	1 s	2 s	9 s	2 s	2 s	42 s	2 s	2 s	114 s	2 s	2 s	328 s	2 s	1 s
lsq	< 1 s	< 1 s	< 1 s	< 1 s	1 s	2 s	4 s	1 s	3 s	31 s	4 s	9 s	357 s	23 s	37 s
LinkedList	2 s	1 s	1 s	13 s	1 s	1 s	62 s	1 s	1 s	217 s	2 s	1 s	712 s	2 s	2 s
Lock1	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	1 s	< 1 s	< 1 s	1 s	1 s	1 s	1 s	1 s	1 s
QuickSort	13 s	1 s	2 s	97 s	2 s	3 s	mout	6 s	7 s	mout	14 s	21 s	mout	65 s	88 s
RBTree	2 s	< 1 s*	< 1 s*	14 s	< 1 s*	< 1 s*	70 s	2 s*	1 s*	269 s	2 s*	2 s*	1046 s	3 s*	2 s*
Dispatching	341 s	1 s*	1 s*	mout	2 s*	1 s*	mout	2 s*	2 s*	mout	2 s*	2 s*	mout	4 s*	4 s*
ArrayFib	5 s	2 s	1 s	71 s	5 s	6 s	335 s	20 s	54 s	1463 s	89 s	239 s	mout	799 s	1286 s
Fibonacci	< 1 s	1 s	1 s	7 s	2 s	1 s	38 s	7 s	1 s	114 s	18 s	2 s	333 s	63 s	6 s
MemoFib	5 s	1 s	1 s	57 s	6 s	5 s	231 s	55 s	53 s	873 s	233 s	232 s	mout	1275 s	1304 s
RecFib	9 s	< 1 s	< 1 s	8 s	1 s	1 s	41 s	1 s	1 s	122 s	3 s	1 s	330 s	11 s	4 s
ArrayUtils	9 s	< 1 s	< 1 s	110 s	1 s	1 s	464 s	1 s	1 s	1736 s	3 s	33 s	mout	11 s	35 s
BubbleSort	8 s	7 s	8 s	69 s	mout	mout	303 s	mout	mout	mout	mout	mout	mout	mout	mout
ConcreteFieldClass	< 1 s	1 s	< 1 s	7 s	1 s	1 s	40 s	1 s	1 s	116 s	1 s	1 s	337 s	1 s	1 s
ExceptionsInOneLine	8 s	6 s	4 s	64 s	5 s	3 s	267 s	4 s	3 s	989 s	4 s	3 s	mout	5 s	5 s
InstanceOfTest	< 1 s	1 s	1 s	13 s	1 s	1 s	68 s	1 s	1 s	215 s	1 s	1 s	649 s	1 s	1 s
ParameterRestrictions	9 s	1 s*	5 s	72 s	1 s*	3 s	340 s	3 s*	5 s	1513 s	3 s*	3 s	mout	4 s*	5 s
IntBufferTest	369 s	4 s	2 s	mout	mout	14 s	mout	mout	154 s	mout	mout	813 s	mout	mout	mout
BinaryHeap	402 s	2 s*	1 s*	mout	2 s*	2 s*	mout	2 s*	2 s*	mout	4 s*	3 s*	mout	5 s*	5 s*
Gesamt	2672 s	45 s	49 s	627 s	50 s	84 s	2349 s	129 s	558 s	7895 s	413 s	5062 s	4470 s	2328 s	2804 s
Anzahl mout+tout	0	0	0	7	2	1	8	2	1	9	2	1	14	2	3

Spalten „-“ ohne, Spalten „+“ mit allen Modellreduktionen und Spalten ’ nur mit Wertebereichsreduktion. mout=Speicherüberlauf, tout=Zeitüberschreitung (> 18000 s)

6. Zusammenfassung

Ziel der Arbeit war die Beschleunigung der Softwaremodellprüfung durch Programmanalysen. Es wurde eine Modellprüfungskette mit Moped sowie *SPDS* als Modelle und der erfolgte Eingriff in diese Kette auf Modellebene ausgewählt. Es wurden vorrangig Techniken zur Verkleinerung des *Zustandsraums* betrachtet, da *Modellprüfer* sehr sensitiv auf die Größe des *Zustandsraums* von Modellen reagieren. Die Veränderung des Zustands- bzw. Konfigurationenraums (i.d.R. unendlich) ist dann charakterisierbar als Änderung der Zustandsanzahl und der Kelleralphabetgröße des *SPDS*. Ein verringerter Konfigurationenraum führt i.d.R. zu einer geringeren Laufzeit der Modellprüfung (experimentell bestätigt). In einigen Fällen wird die Modellprüfung erst ermöglicht[1] und in anderen Fällen erübrigt sich die *Modellprüfung* sogar ganz. Dazu wurden Verfahren zur Optimierung von Modellen für die *Modellprüfung* untersucht und entwickelt (Modellreduktionen). Zu diesen Verfahren zählen die Ausdrucksvereinfachung, die Modellbeschneidung, die Prozedurüberbrückung, die Variablenredundanzelimination sowie die Wertebereichsreduktion. Tabelle 6.1 fasst die wichtigsten Eigenschaften dieser Modellreduktionen zusammen. Die Verfahren sind meist präzise, nicht optimal (bestimmt durch Güte der eingesetzten Modellanalyse), konservativ und produzieren keine neuen Fehlalarme bezüglich der jeweils invarianten temporalen Logiken. Neben praktischen Resultaten als Beleg für deren Nutzen (Experimente) wurden aber auch neue theoretische Erkenntnisse aufgeführt (Lemmas, Sätze). So wurde z.B. gezeigt, dass die exakte Durchführung eingesetzter *Modellanalyse*n im Gegensatz zu herkömmlichen *Programmier-* bzw. Hochsprachen prinzipiell möglich, wenn auch aufwändiger ist als die *Modellprüfung* selbst. Es ist bekannt, dass mittels Modellprüfung bestimmte Modellanalysen (exakt[2]) berechnet werden können [121, 36, 140, 14, 175, 34]. Dann sind diese auch entscheidbar (Existenz einer optimalen Modellanalyse). In meiner Veröffentlichung [69] wurde eine exakte Äquivalenzanalyse vorgestellt, welche ohne Verwendung temporaler Formeln, Variablenäquivalenzen in *SPDS* bestimmt. In der betrachteten Literatur wurde nicht gezeigt, **welche** exakten *Modellanalyse*n und *Transformationen* sich zur Beschleunigung der *Modellprüfung* von *SPDS* eignen und **welche nicht** (wegen ihrer Komplexität). Die vorliegende Arbeit zeigt dagegen, dass viele exakte *Modellanalyse*n mindestens so aufwändig sind wie die Modellprüfung. Dies rechtfertigt die Durchführung approximativer bzw. konservativer *Modellanalyse*n. Exakte *Modellanalyse*n können daher meist nicht genutzt werden, um die *Modellprüfung* von *SPDS* zu beschleunigen, führen aber dennoch zu verbesserten Analysen für die Ausgangssprachen [68]. Auch wurde in dieser Arbeit eine formale Semantikdefinition für *symbolische* Beschreibungen von Kellersystemen geschaffen (siehe Abschnitt 3.2 ab Seite 23), welche die Grundlage für die Korrektheitsbeweise bildete. Zusammengefasst wurde in dieser Arbeit gezeigt, **dass, welche, wie** und **wie nicht**[3] Modell- bzw. *Programmanalysen* genutzt werden können, um die symbolische kellersystembasierte *Softwaremodellprüfung* zu verbessern. Die Resultate dieser Arbeit sollen helfen, die Softwarequalität zu verbessern und einen Beitrag leisten zur Sicherheit der Gesellschaft im Alltag beim Einsatz von Software. Sicherheitskritische Systeme wie sie z.B. im Automobil- und Flugzeugbau auftreten, werden immer komplexer und damit auch ihr zu überprüfendes Verhalten.

[1] D.h. es werden zusätzlich solche *Programme* prüfbar, die es vorher nicht waren.

[2] nur bezüglich der Randbedingungen des eingesetzten Modellprüfers (endliche Modelle ⇒ keine unbeschränkte Rekursion; beschränkte Modellprüfung/bounded model checking).

[3] Hier sind die Komplexitätresultate exakter Analysen gemeint sowie die Unentscheidbarkeit von Modellanalysen wie z.B. das Minimieren der Größe reelwertiger Ausdrücke.

Tabelle 6.1.: Eigenschaften der Modellreduktionen, $\alpha \in \{$ACTL, ACTL*, CTL, CTL*, LTL$\}$

Eigenschaft je Kapitel	**4.3.1** (S. 45)	**4.3.2** (S. 68)	**4.3.3** (S. 78)	**4.3.4** (S. 90)	**4.3.5** (S. 112)
erreichbarkeits-invariant	+	+	+	+	+
α-invariant	+	+	-	+	+
α-X−invariant	+	+	+	+	+
Konfigurationenraum	unverändert	abgeschnitten	überbrückt	komprimiert	abstrahiert
entscheidbar	+	+	+	+	+
optimal	-	-	-	-	-
präzise	+	+	-	- (nur teilweise)	-*
konservativ	+	+	+	+	+
Fehlalarm möglich	-	-	- für α-X	-	-

Legende: *...weder lokal, noch global präzise

7. Ausblick

Die vorgestellten Verfahren können auf vielfältige Weise noch verbessert werden. So befinden sich bereits in den Anlagen auf der beiliegenden CD. Beschreibungen für eine verbesserte *Interpretation* von *Reihungen*, für interprozedurale Verzweigungen sowie eine Erweiterung um beschränkte Parallelität[1] für *SPDS*. So sind z.B. für Slicing noch viele weitere Verbesserungen bekannt, welche untersuchenswert für den vorgeschlagenen Ansatz sind (z.B. Verbesserung durch Laufzeitanalysen [116]).

Synergieeffekte Zusammen mit anderen Arbeiten (wie jene von Bernhard Steffen et al. [36, 35]), worin *Programmanalysen* durch den Einsatz von *Modellprüfung* verbessert werden, kann dann die erneute *Modellprüfung* von den verbesserten *Programmanalysen* profitieren, was wiederum die Programmanalysen verbessert usw...

Weitere Anwendungsbereiche. Nicht nur bei der Software-*Modellprüfung* und beim modellbasierten Testen, sondern auch bei der Codegenerierung sind die Größe und Komplexität von Modellen entscheidende Einflussfaktoren. Modell-Optimierungen verbessern daher auch Bereiche wie modellbasiertes Testen, Testdaten- und Testfallgenerierung sowie Codegenerierung.

Rekursionspräzise Programmanalysen. Werden Modellanalysen für Kellersysteme exakt durchgeführt, um Analyseergebnisse für die Ausgangssprache zu gewinnen, so bezeichnen wir diese Verfahren in eigenen Arbeiten wegen der adäquaten Nachbildung von Rekursion auch als *rekursionspräzise* Analysen [68, 69], da sie eine Rückinterpretation der gewonnenen abstrakten Informationen in die Hochsprache erlauben. So gelingen neben der rekursionspräzisen Intervallanalyse aus [68] weitere sehr *präzise Programmanalysen* für die i.d.R. turingmächtige Ausgangssprache.

Fixpunktverfahren. Die Konvergenz für die Fixpunktverfahren (*Wertebereichsanalyse*, *Äquivalenzanalyse*) lässt sich beschleunigen, indem nicht in jeder Iteration die Informationen für alle Knoten neu berechnet werden, sondern nur diejenigen Knoten erneut betrachtet, für die sich die Eingangsinformation geändert hat (*Worklist-Prinzip*, [51]). Damit lässt sich der Aufwand zur näherungsweisen Bestimmung der Wertebereiche in der Praxis nach Bedarf begrenzen. Auch ist eine Erweiterung um Kontextsensitivität vorstellbar, um die Qualität der Analyseergebnisse zu erhöhen[2]. Diese könnte wie folgt gestaltet werden. Wurde ein *Prozeduraufruf* als **nicht** realisierbar erkannt, aber die Prozedur selber (z.B. durch Aufruf an anderer Stelle) als realisierbar bzw. *erreichbar* identifiziert, so würde der *Programmteil* nach dem *Prozeduraufruf* bei Kontextinsensitivität für *Modellanalysen* *erreichbar* werden. Auch für Call-Strings [21, 126] der Länge k können immer Modelle gefunden werden, in denen der *Programmteil* nach dem *Prozeduraufruf* unnötigerweise als *erreichbar* klassifiziert werden würde. Hierfür ist eine nachträgliche Untersuchung der *Prozeduraufrufe* möglich, um solche Artefakte zu erkennen. Einerseits können solange die nachfolgenden Grundblöcke als nicht *erreichbar* markiert werden, wie es keine eingehende Kante einer anderen *erreichbaren* Stelle gibt. Wiederholung führt zu einem Fixpunkt mit z.B. kleinen *erreichbaren* Kreisen, die jedoch nicht global erreichbar sind. Durch anschließende Suche nach starken Zusammenhangskomponenten (SCC-Analyse [32, 28]) können dann auch diese identifiziert und als nicht *erreichbar* klassifiziert werden. Andererseits ist es jedoch einfacher, zu diesem Zweck direkt den Fixpunktalgorithmus zu modifizieren. Dazu werden (zunächst) keine Konfigurationen nach einer Prozedurrückkehr betrachtet, falls der zugehörige *Prozeduraufruf* (bisher) nicht *erreichbar* ist. Die *symbolische* Konfiguration wird als *wartend* markiert und erst dann betrachtet, wenn die Analyse

[1] realisierbar mit Kaktuskellern [212]
[2] bei erhöhtem Laufzeitbedarf

den zugehörigen *Prozeduraufruf* erreicht. Wird dieser Prozeduraufruf nie erreicht, so werden auch nicht die davon abhängigen wartenden Knoten bearbeitet. Allerdings zeigen Experimente beim Vergleich von konservativen und exakten Modellanalysen, dass die konservativen Realisierungen dieser Arbeit bereits relativ genaue Ergebnisse liefern [69].

Wertebereichsanalyse. Die *Wertebereichsanalyse* kann statt der gesamten Wertemenge lediglich ein einzelnes Intervall betrachten. Dies führt zwar zu unpräziseren Analyseergebnissen, aber andererseits konvergiert sie dann schneller, da sie zu viel schnelleren *Auswertungen* führt (wegen durch abstrakte Interpretation [178] auf Intervallmengen ausgeweiteten arithmetischen Operationen).

Wertebereichsreduktion. T_{min} kann durch Analyse der Formel ϕ *präziser* durchgeführt werden. Z.B. kann der Einfluss einer Typminimierung für die Variable v auf das *Modellprüfungs*resultat genauer bestimmt werden. In der vorgestellten Modellreduktion wird dazu lediglich *konservativ* $v \notin \phi$ verwendet.

Versackung. Ein einfaches Typsystem kann für *SPDS* eingesetzt werden, um z.B. *unsichere* SPDS-*Anweisungen* zu finden. Unsicher sind solche SPDS-*Anweisungen*, welche Variablen v vom Typ $bits(v)$ an andere Variablen w vom Typ $bits(w) < bits(v)$ zuweisen („w = v“). Derartige *SPDS-Zuweisung*en führen potentiell zu terminierenden Läufen (Versackung), wenn die *SPDS*-Variable w einen Lauf mit einer *Variablenbelegung* env besitzt, so dass $[\![v]\!]_{env} \notin range(w)$[3]. Ein Typsystem kann diese Stellen konservativ vorhersagen.

Prozedurüberbrückung. Eine mögliche Verallgemeinerung der Prozedurüberbrückung ist die Überbrückung von SPDS-*Anweisungen* und *SPDS*-Anweisungsblöcken (aufeinanderfolgende *Anweisungen* ohne Verzweigungen, vgl. Grundblock bei *Programmiersprachen*). Dann werden statt Prozeduren SPDS-*Anweisungsblöcke* überbrückt, wenn diese keinen Effekt besitzen. Allerdings muss auch der *Seiteneffekt* auf lokale Variablen berücksichtigt werden, was die Einsetzbarkeit vermutlich stark begrenzt. Durch Inlining [62] (übertragen auf *SPDS*) können *Prozeduraufruf*e ebenfalls eliminiert werden. Ein positiver Nebeneffekt dabei ist die Erhöhung der Kontextsensivität für verschiedene *Modellanalyse*n. Negativ ist jedoch die *Konfigurationenraum*vergrößerung des Modells.

Nebenläufigkeit. Für SPIN sind Erweiterungen auf Multicores verfügbar [94, 29, 25]. Naheliegenderweise kann so auch z.B. der aufwändige Teil der Automatenkonstruktion ($pre^*, post^*$) während der *SPDS-Modellprüfung* bei Moped parallelisiert werden, um auch für die *SPDS*-Modell-prüfung oder die Berechnung exakter Analyseergebnisse Laufzeit einzusparen. Veröffentlichungen im Projekt DIVINE (Distributed Verification Environment [24]) bieten dazu eine Reihe parallelisierter *Modellprüfungs*-Algorithmen und -Datenstrukturen [30, 28, 26, 33, 242, 27, 31, 23].

Missachten der Kriterien dieser Arbeit. Der *Konfigurationenraum* kann noch stärker beschnitten bzw. vereinfacht werden, wenn temporale Formeln ihren Wahrheitsgehalt konservativ verändern dürfen. Auf diese Weise können allerdings neue False Negatives entstehen, was wir in [71] genauer beschrieben haben. Insbesondere in Kombination mit *Programmpfad* und Programmzustands-Partitionierung [177] werden somit weitere deutliche Reduktionen des *Konfigurationensraums* denkbar. Hierzu können z.B. Nebenbedingungen (sog. Constrains) formuliert und mit Hilfe eines SMT-Lösers analysiert werden.

Jede Maßnahme zur Qualitätssicherung hat ihre Vor- und Nachteile. Vlt. können mit den vorgestellten Techniken dieser Arbeit in Zukunft Software-Fehler[4] früher erkannt und behoben werden.

[3] Dies entspricht einem dynamischen Typfehler in einer Hochsprache.

[4] z.B. der Pentium-Bug: Ende 1994 wurde ein Fehler im Intel-Pentium-Prozessor bekannt. Der Fehler verursachte bei Divisionen mit Gleitkommazahlen gelegentlich Rundungsfehler und kostete Intel ca 464 Millionen Dollar. http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/pentiumbug/website/

A. Abkürzende Schreibweisen für SPDS

Hier werden abkürzende Schreibweisen für *SPDS* definiert, welche einerseits die Beschreibung von *SPDS* erleichtern und andererseits die Experimente mit der Modellsprache Remopla des gewählten Modellprüfers Moped erlauben. Diese Abkürzungen erweitern die SPDS-*Anweisungen* zu SPDS'-*Anweisungen*. Sie erlauben eine noch kompaktere Beschreibung von *SPDS*. Zudem können für Sprachkonstrukte aus Hochsprachen *SPDS*-Entsprechungen definiert werden, um die *Modellabstraktion* und gleichzeitig das Modell zu vereinfachen. Für die *Interpretation* von SPDS-*Anweisungen* in *PDS*-Transitionen wird die Interpretationsfunktion $\gamma[\![\cdot]\!] : SPDS \to PDS$ verwendet. Abkürzungen (SPDS'-*Anweisungen*) werden hingegen zunächst als SPDS-*Anweisungen* interpretiert. Dazu dient die *Interpretations*funktion $\delta[\![\cdot]\!] : SPDS' \to SPDS$. Die Komposition beider *Interpretations*funktionen $(\gamma \circ \delta)[\![\cdot]\!] : SPDS' \to PDS$ führt auch für die abkürzenden Schreibweisen daher zu einem *PDS*. Sei hierzu P ein *SPDS*'. Die Schlüsselwörter *true* bzw. $false$ werden als 1 bzw. 0 interpretiert.

Prozedurausdrücke Ein *Prozeduraufruf* mit *Ausdrücke*n hat die Form „$L : p(e_1, e_2, \ldots, e_m);$“ $\in$ P mit $e_j \in Expr$ für eine Prozedur p mit $n \geq m$ Parametern. Für weggelassene Argumente e_j mit $m < j \leq n$ gilt $e_j := 0$. Den Parametervariablen $Param_p = [p_1, p_2, \ldots, p_n]$ werden beim Prozedureintritt bei gegebener *Variablenbelegung env* die Variablenwerte $\forall j \leq n : [\![p_j]\!]_{env} = [\![e_j]\!]_{env}$ zugeteilt.
Um derartige *Prozeduraufruf*e zu interpretieren, werden neue Hilfsvariablen $t_1, t_2, \ldots, t_{n_{max}} \notin Vars$ eingeführt, wobei $n_{max} := max_{p \in Prz}|Param_p|$ die maximale Parameteranzahl ist. In diesen Variablen werden die *Ausdrücke* „zwischengespeichert“ bis zum *Prozeduraufruf*. Der Typ dieser neuen Hilfsvariablen t_i ist der größte Typ, welcher an der i-ten Stelle auftreten kann: $bits(t_i) := max_{p \in Prz} bits(p_i)$, wobei $Param_p = [p_1, p_2, \ldots, p_n]$. Dann ist die *Interpretation* formal definiert als:

$$\begin{aligned} \delta[\![L : p(e_1, e_2, \ldots, e_n);]\!] \quad := \quad & L : t_1 = \delta[\![e_1]\!]; l_2 : t_2 = \delta[\![e_2]\!]; \ldots; l_n : t_n = \delta[\![e_n]\!]; \\ & l_1 : p(t_1, t_2, \ldots, t_n); \end{aligned}$$

Mit neuen Marken $l_1, l_2, ..., l_n \notin Marken(P)$. Folgendes Beispiel veranschaulicht dies.

Beispiel A.0.1 (Prozedurausdrücke)
Folgendes SPDS' *sei gegeben:*

```
int a(6);
void p(int p1(7), int p2(8)) {...}
...
L1: a=5;
L2: p(a+6,a*a);
```

Es enthält eine Prozedur p mit zwei Argumenten p1 *und* p2 *(mit Typ 7 bzw. 8 Bit) sowie eine Variable a mit 6 Bit. Die* SPDS'*-Variable a wird an der Marke* L1 *mit dem Wert* 5 *belegt und für einen* Prozeduraufruf *an der Marke* L2 *mit den* Ausdrücken $a + 6$ *und* $a * a$ *verwendet. Der* Prozeduraufruf *wird dann interpretiert zu:*

```
int a(6), t1(7), t2(8);
void p(int p1(7), int p2(8)) {...}
...
L1: a=5;
L2: t1=a+6; l2: t2=a*a; l1: p(t1,t2);
```

Bei *Interpretation* der SPDS'*-Anweisung* für Prozedurausdrücke wird wegen *Verwendung* der SPDS-*Anweisungen* der *Konfigurationenraum* vergrößert. Einerseits werden zusätzliche globale Variablen t_i eingeführt, welche durch Anzahl und Typ durch die im *SPDS*' definierten Prozeduren bestimmt sind. Andererseits sind zusätzliche *Konfigurationenübergänge* und Marken $l_1, l_2, ..., l_n$ nötig, was ebenso den *Konfigurationenraum* vergrößert und Läufe verlängert.

Parallelzuweisung Eine *Parallelzuweisung* $Pass_n$ hat die Form „$L : x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n;$“ $\in P$. Die *Parallelzuweisung* $Pass_n$ weist gleichzeitig (synchron) den paarweise verschiedenen Variablen x_j bei gegebener *Variablenbelegung env* die Werte $[\![e_j]\!]_{env}$ zu, falls $\forall j : [\![e_j]\!]_{env} \in range(x_j)$. Die *Parallelzuweisung* beschreibt demnach synchrone Parallelität durch einen einzigen Konfigurationsübergang.
Zur *Interpretation* von *Parallelzuweisungen* werden neue Variablen $t_1, t_2, \ldots, t_{n_{max}} \notin Vars$ eingeführt, wobei $n_{max} := max_{Pass_n} n$ die Länge einer längsten *Parallelzuweisung* ist. Mit diesen neuen (temporären) Variablen werden die ausgewerteten *Ausdrücke* zwischengespeichert, um das Ergebnis dann den Ziel-Variablen nacheinander zu zuweisen. Für den Typ der neu eingeführten Variablen t_k mit $k \in \{1, 2, \ldots, \max\{n \mid Pass_n \in P\}\}$ gilt: $bits(t_k) := \max\{bits(x_k) \mid Pass_n =$ „$L : x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n;$“ $\in P\}$. D.h. die längste *Parallelzuweisung* in P bestimmt die Anzahl der t_i und der größte Typ einer Variable x_i an der Stelle i irgend einer *Parallelzuweisung* $Pass_n$ bestimmt den Typ, d.h. die Größe von t_i. Es ist

$$\begin{aligned}\delta[\![L : x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n;]\!] \quad := \quad & L : \delta[\![t_1 = e_1]\!]; l_2 : \delta[\![t_2 = e_2]\!]; \ldots; l_n : \delta[\![t_n = e_n]\!]; \\ & l'_1 : \delta[\![x_1 = t_1]\!]; l'_2 : \delta[\![x_2 = t_2]\!]; \ldots; l'_n : \delta[\![x_n = t_n]\!];\end{aligned}$$

Mit neuen Marken $l_2, l_3, ..., l_n, l'_1, l'_2, ..., l'_n \notin Marken(P)$. Zur Veranschaulichung dieses Sachverhaltes diene folgendes Beispiel.

Beispiel A.0.2 (*Parallelzuweisung*)
Es werden wie folgt 3 Variablen unterschiedlicher Größe definiert und wie folgt verwendet.

```
int a(6), b(1), c(2);
L1: a=7, b=0, c=2;
L2: b=!b, a=a+b, c=b+1;
```

Variable a hat den Typ $bits(a) = 6$. *Variable b hat den Typ* $bits(b) = 1$ *und Variable c hat den Typ* $bits(c) = 2$. *An der Marke L*1 *werden den Variablen a,b bzw. c jeweils gleichzeitig die Werte* $7, 0$ *bzw.* 2 *zugewiesen. Sind vor der Marke L*1 *noch alle* Variablenbelegung*en denkbar, so sind für die* Konfigurationen *nach der Marke L*1 *nur noch Vaiablenbelegungen* $env \in ENV$ *mit den Bedingungen* $env(a) = 7, env(b) = 0$ *und* $env(c) = 2$ *möglich. An der Marke L*2 *wird b schließlich sein Inverses (* 1*) zugewiesen, a wird zu* $[\![a + b]\!]_{env} = 7$ *und c erhält den Wert* $[\![b + 1]\!]_{env} = 1$. *Das Beispiel wird mittels der SPDS-*Anweisungen *interpretiert als:*

```
int t1(6), t2(6), t3(2);
int a(6), b(1), c(2);
L1: t1=7; l2: t2=0; l3: t3=2;
l1': a=t1; l2': b=t2; l3': c=t3;
L2: t1=!b; l5: t2=a+b; l6: t3=b+1;
l4': b=t1; l5': a=t2; l6': c=t3;
```

Wobei der Unterschied zwischen dem Semikolon als Anweisungsende *im Gegensatz zum Komma bei der* Parallelzuweisung *zu beachten ist und* $l1, l2, ..., l6, l1', ...l6' \notin Marken(P)$ *neue Marken sind, welche den* Konfigurationenraum *vergrößern.*

Durch *Verwendung* der SPDS-*Anweisungen* sind zusätzliche globale Variablen t_i nötig, welche durch die Art der *Parallelzuweisung*en bestimmt sind. Diese zusätzlichen Variablen vergrößern ebenfalls den *Konfigurationenraum*. Wenn *Parallelzuweisung*en ohne *Interpretation* durch *Modellanalyse*n und Transformationen behandelt werden, vergrößert sich der *Konfigurationenraum* nicht. Aus diesem Grund werden *Parallelzuweisung*en in den Experimenten auch ohne *Interpretation* berücksichtigt, so dass sich der *Konfigurationenraum* für die Modelle nicht ändert und die Effekte der Transformationen in dieser Hinsicht unverfälscht messbar werden.

Parallelzuweisung mit Prozeduraufruf Eine *Parallelzuweisung mit Prozeduraufruf* Pap_{nm} hat die Form „$L : p(e_{11}, e_{12}, ..., e_{1m}), x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n;$“ $\in P$. Pap_{nm} ruft die Prozedur p mit den Parametern e_{1j} auf und weist gleichzeitig (synchron) den paarweise verschiedenen Variablen x_j bei gegebener *Variablenbelegung env* die Werte $[\![e_j]\!]_{env}$ zu, falls $\forall j : [\![e_j]\!]_{env} \in range(x_j)$.
Zur *Interpretation* werden (wie bei der *Parallelzuweisung*) neue Variablen $t_1, t_2, \ldots, t_{n_{max}} \notin Vars$ eingeführt, wobei $n_{max} := max_{Pap_{nm}} n$ die Länge einer längsten *Parallelzuweisung* ist. Mit diesen neuen (temporären) Variablen werden die ausgewerteten *Ausdrücke* zwischengespeichert, dann die Prozedur p aufgerufen und danach das Ergebnis den Ziel-Variablen nacheinander zugewiesen. Der Typ der neu eingeführten Variablen t_k entspricht der bei Parallelzuweisung. Dann ist

$$\begin{aligned}\delta[\![L : p(e_{11}, e_{12}, ..., e_{1m}), x_2 = e_2, \ldots, x_n = e_n;]\!] \quad := \quad & L : \delta[\![t_1 = e_1]\!]; l_2 : \delta[\![t_2 = e_2]\!]; \ldots; l_n : \delta[\![t_n = e_n]\!]; \\ & l_1 : \delta[\![p(e_{11}, e_{12}, ..., e_{1m})]\!]; \\ & l'_1 : \delta[\![x_1 = t_1]\!]; l'_2 : \delta[\![x_2 = t_2]\!]; \ldots; l'_n : \delta[\![x_n = t_n]\!];\end{aligned}$$

Mit neuen Marken $l_1, l_2, l_3, ..., l_n, l'_1, l'_2, ..., l'_n \notin Marken(P)$. Wie bei der *Parallelzuweisung* ändert sich der Konfigurationenraum analog (eine zusätzliche Marke l_1).

Unbedingter Sprung Die *Anweisung* „*goto* $l;$“ $\in P$ ohne eine Bedingung wird mit der SPDS-*Anweisung* „if (1) *goto* $l;$“ interpretiert. In Zeichen: $\delta[\![goto\ l]\!] := if$ (1) *goto* l. Der *Konfigurationenraum* bleibt durch diese abkürzende Schreibweise unverändert.

Schlüsselwort skip Das Schlüsselwort „*skip* e“ $\in P$ mit $e \in Expr$ unterbricht den aktuellen Lauf, falls die Bedingung e nicht erfüllt ist. Das Verhalten ist vergleichbar mit einer Zusicherung (Assertion) bei *Programmiersprachen*, welche ebenfalls zur Laufzeit geprüft wird und ggf. die *Programmausführung* abbricht. Das Schlüsselwort „*skip*“ $\in P$ ist eine Kurzform für „*skip* 1“ und dient als leere Anweisung.
Die Abkürzung „l:*skip* e;“ $\in P$ wird interpretiert als:

$$\delta[\![l : skip\ e;]\!] := l :\ if\ (!e)\ goto\ l;$$

Die nach „*skip* e“ folgenden *Anweisungen* werden nicht ausgeführt, falls die Bedingung $[\![e]\!]_{env} \neq 1$, d.h. nicht erfüllt ist bei gegebener Belegung *env*. Weiter ist $\delta[\![l : skip;]\!] := \delta[\![l : skip\ 1;]\!]$.
Bei *Verwendung* der Abkürzung „*skip* e“ oder „*skip*“ wird der *Konfigurationenraum* nicht verändert.

Bedingter Anweisungsblock Ein bedingter *Anweisungsblock* $IF_m \in P$ (im Folgenden auch als *Mehrfachverzweigung* bezeichnet) hat die Form:

$$\begin{aligned}&„if \\ &\quad :: \quad b_1 \rightarrow l_1 : L_1 \\ &\quad :: \quad b_2 \rightarrow l_2 : L_2 \\ &\quad ... \\ &\quad :: \quad b_m \rightarrow l_m : L_m \\ &fi;“\end{aligned}$$

Der bedingte *Anweisungsblock* $IF_m \in P$ enthält Listen von *Anweisungen* (Prozedurrümpfe) L_i ($i \in \{1, 2, \ldots, m\}$) beginnend jeweils mit der Marke l_i, von denen die erste mit erfüllter Bedingung $[\![b_i]\!]_{env} = 1$ ($b_i \in Expr$ und $[\![b_i]\!]_{env} \in \{0, 1\}$) als Fortsetzung des aktuellen Laufs dient[1]. Die letzte Bedingung b_m darf durch das Schlüsselwort *else* ersetzt werden, welches erfüllt ist wenn $\forall i : [\![b_i]\!]_{env} = 0$.
Die *Interpretation* von „else“ ist $\delta[\![else]\!] :=\ !(b_1||b_2||\ldots||b_{m-1})$. D.h. die logische Bedeutung von *else* entspricht $\neg \bigvee_{i=1}^{m-1} b_i$. Der bedingte *Anweisungsblock* $IF_m \in P$ ist eine Abkürzung für folgenden

[1] In Remopla ist diese Wahl nichtdeterministisch, was durch einen zusätzlichen *choose*-Aufruf realisiert werden kann.

Quellcode A.1: Beispiel eines bedingten Anweisungsblocks.

```
int x(2);
x = choose(0,2);
if
    :: (x<1)  -> x = x+1;
    :: (x==1) -> x = 0;
fi;
```

Quellcode A.2: *Interpretation* des bedingten *Anweisungsblock*s aus *Quellcode* A.1.

```
        int x(2);
        x = choose(0,2);
        if (x<1)  goto l1;
        if (x==1) goto l2;
        skip false;
   l1 : x = x+1; goto lend;
   l2 : x = 0;   goto lend;
 lend : skip;
```

SPDS-Ausschnitt mit zusätzlichen Marken $l_1, \ldots, l_m, l_{end} \notin$ *Marken*. Es ist[2] $\delta[\![IF_m]\!] :=$

$$
\begin{array}{rcl}
 & & if\ (b_1)\ goto\ l_1 \\
 & & if\ (b_2)\ goto\ l_2 \\
 & & \ldots \\
 & & if\ (b_m)\ goto\ l_m \\
\delta[\![& & skip\ false; \\
l_1 & : & L_1;\ goto\ l_{end}; \\
l_2 & : & L_2;\ goto\ l_{end}; \\
 & & \ldots \\
l_m & : & L_m;\ goto\ l_{end}; \\
l_{end} & : & skip;]\!]
\end{array}
$$

Gilt $\forall i > 0 : [\![b_i]\!]_{env} = 0$, so sind l_i $(i > 0)$ und l_{end} nicht *erreichbar* (gewollte Endlosschleife). Gibt es allerdings eine erfüllte Bedingung $(\exists i : [\![b_i]\!]_{env} = 1)$, so ist l_{end} stets *erreichbar*, sofern auch sämtliche L_i stets endliche Läufe beschreiben und nicht terminieren.
Zur Veranschaulichung diene folgendes Beispiel.

Beispiel A.0.3 (Bedingter Anweisungsblock)
Quellcode *A.1 zeigt ein* SPDS*'-Beispiel für einen bedingten Anweisungsblock. Darin wird zunächst ein zufälliger Wert aus* $\{0, 1, 2\}$ *für die Variable* x *(mit Typ* $bits(x) = 2$*) gewählt. Ist der Wert der Variablen* x *kleiner als* 1*, so wird deren Wert um 1 erhöht. Hat* x *den Wert* 1*, so wird* x *der Wert* 0 *zugewiesen. Hat jedoch* x *den Wert* 2*, so sind* Nachfolgekonfigurationen *nach dem bedingten* Anweisungsblock *nicht* erreichbar*. Die (vorläufige)* Interpretation *des* Quellcodes *A.1 ist in Quellcode A.2 gezeigt. Dies wird dann weiter interpretiert, bis nur noch SPDS-*Anweisungen *verwendet werden. Wird für* x *der Wert* 1 *gewählt durch choose, so ergibt sich ein Lauf entlang der Marke* $l2 \rightarrow lend$*. Die anderen Fälle sind analog.*

Bei *Verwendung* von bedingten *Anweisungsblöcken* IF_{m_i} wird der *Konfigurationenraum* durch jeweils $2 + 2m_i$ zusätzliche *symbolische Konfigurationenübergänge* (Marken) vergrößert.

[2] Nicht angegebene Marken werden der Übersicht wegen implizit vergeben (siehe Bemerkung 3.2.2 auf Seite 25).

Bedingte Anweisung Als Kurzform für einen bedingten *Anweisungsblock* mit nur einer Bedingung kann eine bedingte *Anweisung* verwendet werden. Eine bedingte *Anweisung* hat die Form „$if\ (e)\ s;$“ $\in P$ mit $e \in Expr$ und einer SPDS'-*Anweisung* s. Die *SPDS'-Anweisung* s wird ausgeführt, falls die Bedingung $[\![e]\!]_{env} = 1$ erfüllt ist. Die Abkürzung „$if\ (e)\ s;$“ $\in P$ sei eine Kurzschreibweise für

```
if
    :: e -> s;
    :: else -> skip;
fi;
```

Der *Konfigurationenraum* verändert sich dann erst durch *Interpretation* dieses bedingten Anweisungsblocks.

Schleife Eine Schleife $DO_m \in P$ hat die Form:

$$\begin{array}{ll} do & \\ & :: \quad b_1 \rightarrow l_1 : L_1 \\ & :: \quad b_2 \rightarrow l_2 : L_2 \\ & ... \\ & :: \quad b_m \rightarrow l_m : L_m \\ od; & \end{array}$$

Sie enthält ebenso wie ein bedingter *Anweisungsblock* Listen von Anweisungen (Prozedurrümpfe) L_i beginnend jeweils mit der Marke l_i, von denen wiederum die erste mit erfüllter Bedingung $[\![b_i]\!]_{env} = 1$ ($b_i \in Expr$ und $[\![b_i]\!]_{env} \in \{0,1\}$) als Fortsetzung des aktuellen Laufs dient. b_m darf wieder durch das Schlüsselwort *else* ersetzt werden mit gleicher Bedeutung. Zusätzlich im Gegensatz zum bedingten *Anweisungsblock* ist innerhalb der *Anweisungslisten* das Schlüsselwort *break* erlaubt, welches das Ende der (innersten) Schleife signalisiert ($\delta[\![break]\!] := goto\ l_{end}$).
Eine Schleife DO_m ist eine Kurzschreibweise für $\delta[\![DO_m]\!] :=$

$$\begin{array}{rcl} l_{loop} & : & if\ (b_1)\ goto\ l_1 \\ & & if\ (b_2)\ goto\ l_2 \\ & & ... \\ & & if\ (b_m)\ goto\ l_m \\ \delta[\![& & skip\ false;]\!] \\ \delta[\![l_1 & : & L_1;]\!]\ if(1)\ goto\ l_{loop}; \\ \delta[\![l_2 & : & L_2;]\!]\ if(1)\ goto\ l_{loop}; \\ & & ... \\ \delta[\![l_m & : & L_m;]\!]\ if\ (1)\ goto\ l_{loop}; \\ l_{end} & : & if\ (0)\ goto\ l_{end}; \end{array}$$

Gilt $\forall i > 0 : [\![b_i]\!]_{env} = 0$, so sind auch hier l_i ($i > 0$) und l_{end} von l_{loop} aus nicht *erreichbar* ($l_{loop} \not\rightsquigarrow l_{end}$).
Die *Verwendung* von Schleifen DO_{m_i} führt zu einer Vergrößerung des *Konfigurationenraum*s durch jeweils $2 + 2m_i$ zusätzliche *symbolische Konfigurationenübergänge* (Marken).

Prozedurrückgabewerte Eine Prozedur kann bei Prozedurende einen Rückgabewert liefern. Dazu wird der SPDS-*Anweisung return* ein Ausdruck $e \in Expr$ nachgestellt: *return* e;. Ein *Prozeduraufruf* darf dann auf der rechten Seite einer *Anweisung* (RHS) verwendet werden.
Anweisungen der Form *return* e; sowie *Prozeduraufrufe* $x = p(...)$; werden interpretiert durch Einführung neuer spezieller globaler Variablen $reti \in Vgbl$ - je Rückgabetyp eine, d.h. unabhängig

von der Anzahl der Prozeduren. Seien $e, g_1, g_2, \ldots, g_m \in Expr$. Dann ist

$$\begin{aligned} \delta[\![return\ e]\!] &:= reti = e;\ return; \\ \delta[\![x = p(g_1, g_2, \ldots, g_m)]\!] &:= p(g_1, g_2, \ldots, g_m); x = reti; \end{aligned}$$

Eine Prozedur braucht nicht durch eine **Return**-Anweisung abgeschlossen zu sein. Endet eine Prozedur ohne die *Anweisung return*, so wird diese am Prozedurende während der *Interpretation* ergänzt[3]. Damit werden Prozeduren, welche via SPDS-*Anweisung* formuliert sind, auch weiterhin stets mit einer *return-Anweisung* beendet. Für mit *Prozeduraufruf*en kombinierte *Parallelzuweisung*en gilt ($e_2, \ldots, e_n \in Expr$):

$$\delta[\![return\ e, x_2 = e_2, \ldots, x_n = e_n;]\!] := \delta[\![iret = e, x_2 = e_2, \ldots, x_n = e_n;\ return;]\!]$$

Sowie beim *Prozeduraufruf*:

$$\begin{aligned} \delta[\![x_1 = p(g_1, g_2, \ldots, g_m), x_2 = e_2, \ldots, x_n = e_n]\!] := \\ \delta[\![t1 = g_1; t2 = g_2; \ldots, tm = g_m; x_2 = e_2, \ldots, x_n = e_n;]\!] \\ \delta[\![x_1 = p(t1, t2, \ldots, tm);]\!] \end{aligned}$$

Wobei $t1, t2, \ldots, tm$ Variablen mit $bits(ti) = \max\{bits(x) \mid p \in Prz(P), x \in Params_p\}$ sind[4]. Die resultierenden *Parallelzuweisung*en werden (wie bereits beschrieben) weiter interpretiert.
Der *Konfigurationenraum* wird durch diese abkürzende Schreibweise wie folgt vergrößert. Die unterschiedliche Anzahl der Prozedurrückgabe-Typen entscheidet über die Anzahl der neu eingefügten Variablen *reti*, welche den *Konfigurationenraum* vergrößern. Zusätzlich sind zwei bzw. mehr weitere *Konfigurationenübergänge* zur *Interpretation* nötig (Marken). Werden *Parallelzuweisung* mit *Prozeduraufruf*en kombiniert, so sind mehr als zwei zusätzliche *Konfigurationenübergänge* nötig und zudem auch Variablen *ti*, deren Anzahl und Typ durch die Signaturen bestimmt werden. Anschließend wird bei der *Interpretation* der entstehenden *Parallelzuweisung* der *Konfigurationenraum* weiter vergrößert.

Reihungen *Reihungen* (Arrays) sind für *SPDS* wichtig, da sie oft den größten Anteil am *Konfigurationenraum* besitzen und die in dieser Arbeit untersuchten Transformationen *Reihungen* manipulieren und damit einen sehr großen Effekt bei der *Konfigurationenraum*verkleinerung erzielen. Eine generelle Beschränkung auf Nicht-*Reihungsvariablen* wäre zwar prinzipiell möglich, aber wegen des einher gehenden starken Einflusses auf den *Konfigurationenraum* ungeeignet. Zunächst werden eindimensionale *Reihungen* erläutert und dann beschrieben, wie mehrdimensionale *Reihungen* auf diese zurück geführt werden können.
Eine (eindimensionale) *Reihung* v besitzt den Typ $bits(v) > 0$ und eine statisch definierte Länge $len(v) \in \mathbb{N}_{>0}$. Ein einzelner *Reihungseintrag* $v[i]$ an der Stelle $i \in \{0, 1, 2 \ldots, n_v\}$ der Reihung v kann die Werte 0 bis $2^{bits(v)} - 1$ annehmen, wobei $n_v := len(v) - 1$.
Sind lediglich globale *Reihungen* vorhanden, so genügt die verbesserte *Interpretation* von *Reihungen* aus der Anlage auf beiliegender CD. Im Folgenden wird eine generellere Interpretation angeführt, welche sowohl für lokale als auch für globale *Reihungen* gültig ist. $[\![v[i]]\!]_{env}$ ist undefiniert ($[\![v[i]]\!]_{env} := \bot$), falls $i \notin \{0, 1, 2 \ldots, n_v\}$. Wird eine *Reihung* mit einem derartigen Index indiziert, so gibt es keine *Folgekonfiguration* zum Fortsetzen des aktuellen Laufs nach der aktuellen Marke.
Reihungen werden mittels gewöhnlicher Variablen interpretiert. Dazu werden für jede Reihung v der Länge $len(v) = 1 + n_v$ und mit dem Typ $b_v = bits(v)$ insgesamt $len(v)$ einzelne Variablen $v0, v1, \ldots vn_v$ sämtlich mit dem Typ b_v neu in das *SPDS* eingeführt. Für eine globale *Reihung* werden entsprechend globale Variablen eingeführt und für eine lokale *Reihung* lokale Variable. Der *Konfigurationenraum* für Variablenwerte des *SPDS* P ändert sich durch diese Ersetzung (noch) nicht.
Lesende und *schreibende* Zugriffe auf *Reihungen* werden unterschiedlich interpretiert. Zuerst werden lesende *Reihungszugriffe* interpretiert mittels der *Interpretations*funktion $\delta_{read} : SPDS' \to SPDS'$

[3] Wie in anderen Fällen auch, verändern dann temporale Aussagen mit dem X-Operator evtl. ihren Wahrheitsgehalt.
[4] Eine Verfeinerung wäre z.B. die Berücksichtigung der Reihenfolge der Parameter, welche in der Praxis bei eher homogenen Parametertypen unnötig ist.

Quellcode A.3: Implementation der Prozedur *arracc* zum Ersetzen von *Reihungszugriffen*.

```
int(b) arracc(int i(k), int max(k),
              int v0(b), int v1(b), ... , int vn(b)) {
stop: if (i>max) goto stop;
      if
        :: (i==0) -> return v0;
        :: (i==1) -> return v1;
        ...
        :: (i==n) -> return vn;
      fi;
}
```

und danach werden schreibende *Reihungszugriffe* interpretiert mittels $\delta_{write} : SPDS' \to SPDS'$. Die Komposition ergibt die *Interpretation* von *Reihungen* $\delta_{write} \circ \delta_{read}$. Zur Unterstützung von Reihungen wird eine neue *Interpretation*sfunktion $\delta_{[]} := \delta \circ \delta_{write} \circ \delta_{read} : SPDS' \to SPDS$ für *SPDS*'-Modelle eingeführt, so dass zu aller erst *Reihungen* (lesend, dann schreibend) interpretiert werden und danach alle anderen abkürzenden Schreibweisen. D.h. es ist

$$\delta_{[]}[\![s]\!] = \delta[\![\delta_{write}[\![\delta_{read}[\![s]\!]]\!]]\!]. \tag{A.1}$$

Lesender Reihungszugriff Der lesende Zugriff auf einen *Reihungseintrag*[5] $v[i]$ erfolgt über einen *Prozeduraufruf*

$$int(b)\ arracc(int\ i(k),\ int\ max(k),\ int\ v^0(b),\ int\ v^1(b), \ldots,\ int\ v^n(b)),$$

wobei $arracc \notin Prz, b = \max\{bits(v) \mid v \text{ ist } Reihung \text{ in } P\}$ die maximale Bitbreite von Reihungsvariablen ist[6], $n = \max\{len(v) - 1 \mid v \text{ ist } Reihung \text{ in } P\}$ der maximal zugreifbare Index über alle *Reihungen*[7] und $k = \max(1, \lceil log_2(n+1) \rceil)$ die nötigen Bits zur Adressierung der Indizes in *Reihungen*. Die Implementierung der Prozedur *arracc* findet sich in *Quellcode* A.3.
Ein **lesender** Zugriff auf eine *Reihung* innerhalb einer *Anweisung* $s \in Stats(P)$ bzw. innerhalb eines Ausdrucks $e \in Expr$ wird wie folgt durch *Verwendung* von *arracc* ersetzt. Sei dazu $s \in Stats(P)$ eine *Anweisung* und $e_j \in Expr(s)$ $(j = 1, 2, \ldots, z)$ in s enthaltene *Ausdrücke* sowie $v_i[p_i] \in e_j$ $(i = 1, 2 \ldots, m)$ darin enthaltene lesende Zugriffe auf *Reihungen* v_i mit $p_i \in Expr$. Diese *Reihungszugriffe* $v_i[p_i]$ seien o.B.d.A. wegen möglichen geschachtelten Reihungszugriffen derart geordnet, dass zuerst die innersten *Reihungszugriffe* auftreten und danach erst Reihungszugriffe, welche die *Auswertung* anderer Reihungszugriffe benötigen[8]. Dadurch sind zur *Auswertung* eines *Reihungszugriffs* $v_i[p_i]$ bereits alle inneren Reihungszugriffe $v_j[p_j] \in p_i$ mit $j < i$ ausgewertet. Dies ist stets möglich, da die *Ausdrücke* endlich sind. Zur *Interpretation* lesender *Reihungszugriffe* werden wie im Beispiel neue globale Hilfsvariablen a_i $(i = 1, 2, \ldots, m)$ eingeführt, wobei m die maximale Anzahl *Reihungsvorkommen* $v_i[p_i] \in Stats(s)$ an einer SPDS-*Anweisung* s bezeichne. D.h. es ist

$$\begin{aligned} m &:= \max_{s \in Stats(P)} |\{v_i[p_i] \mid „v_i[p_i]“ \in e,\ e \in Expr(s)\}| \\ bits(a_i) &:= b. \end{aligned}$$

Weiter bezeichne $e[x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots]$ die syntaktische Ersetzung von x_i durch y_i in $e \in Expr$ sowie

$$p'_{i+1} := p_{i+1}[v_1[p'_1] \mapsto a_1, v_2[p'_2] \mapsto a_2, \ldots, v_i[p'_i] \mapsto a_i],$$

[5] Dieser kann auch geschachtelt sein z.B. $v[v[j]]$. Dann ist erst $v[j]$ auszuwerten zu $a = [\![v[j]]\!]_{env}$ und damit dann $v[a]$.
[6] Es ist $b > 0$, da *Reihungen* vom Typ $bits(v) = 0$ nicht erlaubt sind.
[7] Es ist $n \geq 0$, da leere Reihungen nicht erlaubt sind.
[8] Im Beispiel $v[v[j]]$ wird der innere *Reihungszugriff* „$v[j]$“ vor „$v[v[j]]$“ geordnet und damit zuerst ausgewertet.

Quellcode A.4: *SPDS*-Beispiel für *Reihungen*.

```
int h[5](8);
init main;

void main() {
  int x[3](2);
  x[0] = 2;
  x[1] = 3;
  h[x[0]] = x[0] + x[x[1]-2];
}
```

die rekursive Ersetzung von bisherigen *Reihungszugriffen* in auftretenden Teilausdrücken, wobei $p_1 = p'_1$. Damit können lesende Zugriffe innerhalb einer *Anweisung* s interpretiert werden als:

$$\begin{aligned}
\delta_{read}[\![s]\!] \quad := \quad & a_1 = arracc(p'_1, n_{v_1}, v_1^0, v_1^1, \ldots v_1^{n_{v_1}}); \\
& a_2 = arracc(p'_2, n_{v_2}, v_2^0, v_2^1, \ldots v_2^{n_{v_2}}); \\
& a_3 = arracc(p'_3, n_{v_2}, v_3^0, v_3^1, \ldots v_3^{n_{v_3}}); \\
& \ldots \\
& a_{m-1} = arracc(p'_{m-1}, n_{v_{m-1}}, v_{m-1}^0, v_{m-1}^1, \ldots v_{m-1}^{n_{v_{m-1}}}); \\
& a_m = arracc(p'_m, n_{v_m}, v_m^0, v_m^1, \ldots v_m^{n_{v_m}}); \\
& s[v_1[p'_1] \mapsto a_1, v_2[p'_2] \mapsto a_2, \ldots, v_m[p'_m] \mapsto a_m].
\end{aligned}$$

Man beachte, dass fehlende Parameter per Definition von Prozedurausdrücken als 0 aufgefasst werden. Enthält s keine lesenden *Reihungszugriffe*, so ist $\delta_{read}[\![s]\!] = s$. Beispiel A.0.4 veranschaulicht dies.

Schreibender Reihungszugriff Ein **schreibender** Zugriff auf eine *Reihung* „$v[x] = y$“ $\in Stats(P)$ mit $x, y \in Expr$ der Länge $len(v) = n + 1$ erfolgt ebenso durch eine große bedingte Verzweigung, jedoch ohne einen *Prozeduraufruf*. Zur *Interpretation* der *Anweisung* $v[x] = y$ sind zunächst lesende *Verwendungen* von *Reihungen* innerhalb der Anweisung zu interpretieren, so dass x und y keine *Reihungszugriffe* mehr enthalten (siehe Formel A.1). Dann ist

$$\begin{aligned}
\delta_{write}[\![v[x] = y]\!] := \quad & if \\
& :: \quad (x == 0) -> \; v0 = y; \\
& :: \quad (x == 1) -> \; v1 = y; \\
& \ldots \\
& :: \quad (x == n) -> \; vn = y; \\
& fi;
\end{aligned}$$

Enthält s keine schreibenden *Reihungszugriffe*, so ist $\delta_{write}[\![s]\!] = s$.

Beispiel A.0.4 (*Reihung*)
Zur Veranschaulichung sei das SPDS *aus* Quellcode *A.4 gegeben. Darin wird eine globale* Reihung h *mit Länge* $len(h) = 5$ *und Typ* $bits(h) = 8$ *definiert sowie eine lokale Reihung* x *mit Länge* $len(x) = 3$ *und Typ* $bits(x) = 2$. *Es gibt eine Prozedur main, die als* Anfangskonfiguration *des* SPDS *dient (spezifiziert durch „init main“). Der* Reihung x *werden an den Stellen* 0 *bzw.* 1 *die Werte* 2 *bzw.* 3 *zugewiesen. Der* Reihung h *wird an der Stelle* $x[0] = 2$ *der Wert* $x[0] + x[x[1] - 2] = 2 + x[3 - 2] = 2 + 3 = 5$ *zugewiesen. Das Beispiel aus* Quellcode *A.4 wird (zunächst) interpretiert gemäß Quellcode A.5. Anschließend wird es weiter interpretiert, bis nur noch SPDS-*Anweisungen verwendung *finden.*

Quellcode A.5: Interpretiertes *SPDS*-Beispiel für *Reihungen* aus *Quellcode* A.4.

```
int h0(8), h1(8), h2(8), h3(8), h4(8);
init main;

int(8) arracc(int i(3), int max(3), int v0(8),
  int v1(8), int v2(8), int v3(8), int v4(8)) {
stop: if(i>max) goto stop;
      if
        :: (i==0) -> return v0;
        :: (i==1) -> return v1;
        :: (i==2) -> return v2;
        :: (i==3) -> return v3;
        :: (i==4) -> return v4;
    fi;
}

void main() {
  int x0(2), x1(2), x2(2);
  int a1(8), a2(8), a3(8);

  if
    :: (0==0) -> x0 = 2;
    :: (0==1) -> x1 = 2;
    :: (0==2) -> x2 = 2;
  fi;
  if
    :: (1==0) -> x0 = 3;
    :: (1==1) -> x1 = 3;
    :: (1==2) -> x2 = 3;
  fi;
  a1 = arracc(0, 2, x0, x1, x2);
  a2 = arracc(1, 2, x0, x1, x2);
  a3 = arracc(a2-2, 2, x0, x1, x2);
  if
    :: (a1 == 0) -> h0 = a1 + a3;
    :: (a1 == 1) -> h1 = a1 + a3;
    :: (a1 == 2) -> h2 = a1 + a3;
    :: (a1 == 3) -> h3 = a1 + a3;
    :: (a1 == 4) -> h4 = a1 + a3;
  fi;
}
```

Mehrdimensionale Reihungen werden mittels eindimensionaler Reihungen interpretiert. Eine mehrdimensionale *Reihung* x der Dimension d mit den Längen $len_i(x)$ für die Dimension i, wird auf eine eindimensionale *Reihung* x1 mit der Dimension eins und der Länge $len(x1) = \prod_{i=1}^{d} len_i(x)$ abgebildet, so dass

$$[\![x[i_1][i_2]\dots[i_{d-1}][i_d]]\!] := x1[i_1 * \prod_{i=2}^{d} len_i(x) + i_2 * \prod_{i=3}^{d} len_i(x) + \dots + i_{d-1} * len_d(x) + i_d]$$

gilt.
Der ***Konfigurationenraum*** wird durch die abkürzende Schreibweise für *Reihungen* (eindimensional und mehrdimensional) wie folgt vergrößert. Beim Ersetzen von *Reihungsvariablen* durch gewöhnliche Variablen wird der *Konfigurationenraum* nicht verändert. Für den Zugriff (schreibend sowie lesend) via Index-Variable sind bei dieser *Interpretation* weitere SPDS-*Anweisungen* und Variablen nötig. Im Details handelt es sich um eine Prozedur für den lesenden *Reihungszugriff* und je eine bedingte Verzweigung für den schreibenden *Reihungszugriff*.
Beim *lesenden* Reihungszugriff ist es egal, wie häufig auf *Reihungen* zugegriffen wird. Der *Konfigurationenraum* vergrößert sich in jedem Fall stets nur um eine zusätzliche Prozedur *arracc*. Die Anzahl der *SPDS*-Übergänge (Grad der bedingten Verzweigung) und Parametervariablen $v0, v1, \dots vn$ der Prozedur *arracc* sind durch die Länge der längsten *Reihung* bestimmt. Ebenso ist der Typ der Parametervariablen i und max durch die Länge der längsten *Reihung* bestimmt und der Typ der Parametervariablen $v0, v1, \dots vn$ durch den größten Typ einer *Reihung*. Zudem vergrößern neue globale Variablen $a_1, a_2, \dots a_m$ den *Konfigurationenraum*, deren Typ b wiederum durch den größten Typ von *Reihungen* und deren Anzahl m von der maximalen Anzahl lesender *Reihungsverwendungen* innerhalb einer SPDS-*Anweisung* $s \in Stats$ abhängt. Verschiedene SPDS-*Anweisungen* verwenden natürlich die gleichen globalen Variablen $a_1, a_2, \dots a_m$.
Beim *schreibenden* Reihungszugriff vergrößert sich der *Konfigurationenraum* auch in Abhängigkeit der Häufigkeit der Zugriffe. Im Gegensatz zu lesenden *Reihungszugriffen* wird beim Interpretieren keine neue globale Variablen sondern eine bedingte Verzweigung verwendet. Dadurch wird der *Konfigurationenraum* vergrößert durch neue zusätzliche *SPDS*-Konfigurationenübergänge in Abhängigkeit des Länge der schreibend verwendeten *Reihung*. Dies gilt insbesondere für die *Interpretation* der neu erzeugten bedingten Verzweigungen.
Transformationen, welche *Reihungen* verkleinern, reduzieren in *SPDS* daher nicht nur die Anzahl verwendeter Variablen, sondern auch direkt die *Programmgröße*, da die bedingten Verzweigungen weniger Fallunterscheidungen zu treffen haben.

Quantifizierte Anweisungen Eine allquantifizierte SPDS-*Anweisung* hat die Form „$A\ i\ (a,b)\ e_1 = e_2;$“ mit $a, b \in \mathbb{N}, a < b$ und $e_1, e_2 \in Expr$. Eine existenziell quantifizierte SPDS-*Anweisung* hat die Form „$E\ i\ (a,b)\ e_1 = e_2;$“.
Eine allquantifizierte SPDS-*Anweisung* wird interpretiert als *Parallelzuweisung*, d.h.

$$\begin{aligned}\delta[\![A\ i\ (a,b)\ e_1 = e_2]\!] \quad := \quad & \delta[\![e_1[i \mapsto a] = e_2[i \mapsto a],\\ & e_1[i \mapsto a+1] = e_2[i \mapsto a+1],\\ \dots\\ & e_1[i \mapsto b-1] = e_2[i \mapsto b-1],\\ & e_1[i \mapsto b] = e_2[i \mapsto b]]\!];.\end{aligned}$$

Eine existenziell quantifizierte SPDS-*Anweisung* wird interpretiert als

$$\begin{aligned}\delta[\![E\ i\ (a,b)\ e_1 = e_2]\!] \quad := \quad & t = choose(a,b);\\ & \delta[\![e_1[i \mapsto t] = e_2[i \mapsto t];]\!]\end{aligned}$$

für eine neue globale Variable $t \notin Vars$ mit dem Typ $bits(t) = \lceil \log_2(b_{max}+1) \rceil$, wobei b_{max} das größte vorkommende b in existenziell quantifizierten SPDS-*Anweisungen* ist.

Die allquantifizierte *Anweisung* wird sehr häufig genutzt, um *Reihungen* zu initialisieren. Zum Beispiel werden durch die allquantifizierte *Anweisung* „$A\ i\ (0,5)\ x[i] = 0;$“ die *Reihungseinträge* von 0 bis 5 der Reihung x auf 0 gesetzt.
Der *Konfigurationenraum* bleibt bei der *Interpretation* allquantifizierter *Anweisungen* unverändert, jedenfalls zunächst. Erst durch weitere Interpretation der *Parallelzuweisung* vergrößert sich der *Konfigurationenraum* wie beschrieben. Sobald mindestens eine existenziell quantifizierte SPDS-*Anweisung* verwendet wird, vergrößert sich der *Konfigurationenraum* um eine globale Variable $t \notin Vars$ und um einen zusätzlichen Konfigurationenübergang.

Schlüsselwort undef Die *Anweisung* $x = undef$ setzt eine Variable auf einen undefinierten Wert, was mit einem Aufruf an die spezielle Funktion *choose* erreicht werden kann: $\delta[\![x = undef]\!] := g = choose(0, 2^{bits(x)} - 1); x = g;$, wobei $g \notin Vars$ eine neue globale Variable ist, da *choose* nur für globale Variablen verwendet werden darf (weil *choose* als *Prozeduraufruf* mit *Seiteneffekt* an g aufgefasst wird). Der *Konfigurationenraum* bleibt bei dieser *Interpretation* unverändert, so lange x eine globale Variable ist und daher die zusätzliche Variable g entfällt. In allen anderen Fällen wird der *Konfigurationenraum* um eine Variable vom Typ $bits(g) = max_{v \in Vars} bits(v)$ vergrößert (egal, wie oft $undef$ verwendet wird).

Verbunde Ein *Verbund* v (auch bezeichnet als *Record* oder *Struktur*) besteht aus Attributen $a_1, a_2, \ldots, a_n$ mit den Typen $bits(a_i) = b_i$. v hat dann den Typ $((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n))$. Auf einen Verbund v wird qualifiziert (lesend sowie schreibend) auf ein Attribut a zugegriffen: „$v.a$“. Arithmetik direkt auf einem Verbund ist nicht erlaubt. Verbunde v, w gleichen Typs können durch eine Verbund-*Zuweisungs*-*Anweisung* $v = w$ kopiert werden. Dabei werden sämtliche Attribute von w den Attributen von v zugewiesen[9].
Die Manifestation eines Verbunds v mit Attributen $a_1, a_2, \ldots, a_n$ wird interpretiert mittels gewöhnlicher Variablen $va_1, va_2, \ldots, va_n \notin Vars$ mit den Typen $bits(va_i) = bits(a_i)$. Lesende wie schreibende Attributzugriffe „$v.a_i$“ $\in Stats(P)$ werden ersetzt durch „va_i“. D.h. $\forall s \in Stats(P), \forall v.a_i \in s : \delta[\![s]\!] := \delta[\![s[v.a_i \mapsto va_i]]\!]$. Die Verbund-*Zuweisungs*-*Anweisung* $v = w$ wird interpretiert mittels einer Parallelzuweisung: $\delta[\![v = w]\!] := \delta[\![va_1 = wa_1, va_2 = wa_2, \ldots va_n = wa_n;]\!]$.
Auch *Reihungen* und Verbunde sind als Attribute möglich, werden dann aber zuvor entsprechend wie beschrieben interpretiert. Dabei wird der aktuelle Verbund erst interpretiert, wenn alle seine Attribute interpretiert sind, d.h. keine *Reihungen* mehr intern verwendet werden. Geschachtelte Verbunde sind auf gleiche Weise ebenso möglich und werden analog hierarisch von innen nach außen interpretiert.

Beispiel A.0.5 (Verbunde)
Das SPDS*' aus* Quellcode *A.6 verdeutlicht die* Interpretation *von Verbunden. Darin ist ein Verbund* $v1$ *definiert mit Typ* $S1$ *- ein Alias für* $((a0, 2), (a1, 2), (b, 3))$. *Sowie ein Verbund* w *mit Typ* $S2$*, welcher wiederum einen Verbund als Attribut enthält. Durch* Interpretation *der SPDS'*-Anweisungen *entsteht das* SPDS*' aus* Quellcode *A.7. Die Zwischenschritte der* Interpretation *finden sich in der Anlage auf der beiliegenden CD.*

Der *Konfigurationenraum* wird nur indirekt durch Einführung von *Parallelzuweisung*en beeinflusst. Es werden aber davon abgesehen bei der *Interpretation* von Verbunden keine zusätzlichen Variablen oder *Konfigurationenübergänge* eingeführt.

Zusammenfassung für SPDS' Damit wurden alle in dieser Arbeit verwendeten Abkürzungen definiert und auf SPDS-*Anweisungen* und damit letztendlich auf *PDS* zurück geführt. Bei der *Interpretation* wurden zusätzliche Hilfsvariablen und Marken eingeführt, die jedoch lediglich der Definition der Semantik dienen. Durch direktes Interpretieren von $\gamma \circ \delta$ wären diese nicht nötig, so dass abkürzende Schreibweisen den Zustandsraum dann nicht beeinträchtigen. Dann sind allerdings sämtliche *Modellanalyse*n und Transformationen auf SPDS*'-Anweisungen* zu erweitern, was teilweise für wichtige Analysen im Anhang erfolgt ist.

[9] Zwei Verbunde sind dann äquivalent, wenn ihr interpretierter Typ gleich ist und alle Attribute gleich sind.

Quellcode A.6: Beispiel für *SPDS*-Verbunde.

```
STRUCT S1 {
  int a[2](2);
  int b(3);
} v1;

STRUCT S2 {
  S1 v;
} w;

void main() {
   A i (0,1) v1.a[i]=1;
   w.v = v1;
}
```

Quellcode A.7: *Interpretation* des Beispiels aus *Quellcode* A.6.

```
int t1(2), t2(2), t3(3);
int v1a0(2), v1a1(2), v1b(3);
int wva0(2), wva1(2), wvb(3);

void main() {
   t1 = 1; t2 = 1;
   if
     :: (0==0) -> v1a0=t1;
     :: (0==1) -> v1a1=t1;
   fi;
   if
     :: (1==0) -> v1a0=t2;
     :: (1==1) -> v1a1=t2;
   fi;
   t1 = v1a0; t2 = v1a1; t3 = v1b;
   wva0 = t1; wva1 = t2; wvb = t3; }
```

B. Beweise wichtiger Aussagen

Satz 3.3.1 auf Seite 39 (*Erreichbarkeits-Invarianz*)
Die *Erreichbarkeits-Invarianz* von *Ausdrücken* kann zurückgeführt werden auf die *Erreichbarkeits-Invarianz* von Marken.

Beweis (Skizze)
Es wird der Prozedurrumpf einer jeden Prozedur p mit $Vars(b) \subseteq Vgbl \cup Vlcl_p$ der Form

```
p (...)
{
    l1: a1;
    l2: a2;
    ...
    ln: an;
}
```

wie folgt modifiziert. An allen Marken, an denen $Vars(b) \not\subseteq Vgbl \cup Vlcl_p$, wird keine Änderung vorgenommen. An solchen *Konfigurationen* s mit derartigen Marken enthält der Ausdruck b Variablen, welche nicht durch eine *Variablenbelegung* env^s belegt sind (d.h. dort ist $[\![b]\!]_{env^s} = \bot$). Nur an den Marken, an denen gilt $Vars(b) \subseteq Vars(env^s)$, werden die Transitionsübergänge einer *Konfiguration* s so ergänzt, dass nach jeder Transition an Marke li der Ausdruck b auf seinen *Wahrheitsgehalt* geprüft wird. Falls der Ausdruck b an einer *Konfiguration* s mit der Marke li erfüllt ist, d.h. $[\![b]\!]_{env^s} = 1$, so wird an eine neue ausgezeichnete globale Marke $l \notin Marken(S)$ verzweigt, welche auf *Erreichbarkeit* geprüft werden kann.

```
p (...)
{
    l1: a1;  if (b) goto l;
    l2: a2;  if (b) goto l;
    ...
    ln: an;  if (b) goto l;
    l0: return; l: goto l;
}
```

Wenn der Ausdruck b dann an einer Marke li erfüllbar ist, so wird zur Marke l verzweigt. Ist der Ausdruck b hingegen in der Prozedur p nie erfüllbar, so ist die Marke l auch nicht *erreichbar*. D.h. es ist $l \in Post^*(S) \Leftrightarrow b \in Post^*(S)$. Somit kann zur Prüfung, ob $b \in Post^*(S)$ gilt, statt dessen die Prüfung der *Erreichbarkeit* der Marke l für jede Prozedur p bestimmt werden. Dies bedeutet, dass die *Erreichbarkeits-Invarianz* von *Ausdrücken* zurückgeführt werden kann auf die Erreichbarkeits-Invarianz von Marken.

□

Lemma 4.3.9 auf Seite 61 (*Wertebereichsanalyse* ist Monotones Rahmenwerk)
$L = (2^{ENV}, \subseteq)$ ist ein Verband. Die *Ausdrücke* zur Definition der Datenflussgleichungen von Out_l und In_l aus Definition 4.3.5 auf Seite 53 sind monoton.

Beweis (Skizze)
Die Menge aller *Variablenbelegungen* ENV ist endlich, da es nur endlich viele Variablen gibt und

diese immer einen endlichen Typen haben. Auf der Teilmengenbeziehung $\subseteq$ bilden daher Mengen von *Variablenbelegung*en offensichtlich einen Verband.
Die Joins aus Definition 4.3.4 auf Seite 52 sind stets Vereinigungen von *Variablenbelegung*en. Mehr mögliche (eingehende) *Variablenbelegung*en führt auch zu mehr vereinigten (gejointen) Variablenbelegungen. D.h. die Joins sind monoton. In_l ist ebenso eine Vereinigung dieser Joins und daher auch monoton. Für Out_l wird folgende Fallunterscheidung betrachtet:

1. Fall: *Zuweisung* „$l : x = e$“.
   Dann ist $\{env|_x^{[\![e]\!]_{env}} \mid env \in In_l\}$ monoton steigend für größer werdendes In_l. Denn mehr eingehende *Variablenbelegung*en In_l führen auch zu mehr ausgehenden Variablenbelegungen Out_l (analog monoton fallend).

2. Fall: Choose-*Zuweisung* „$l : x = choose(a, b);$“
   Es ist $\{env|_x^i \mid env \in In_l,\ i \in [a, b] \cap range(x)\}$ monoton steigend in In_l, da die Variablenwerte $i \in [a, b] \cap range(x)$ stets gleich sind (unabhängig von In_l). So ergeben sich mehr eingehende *Variablenbelegung*en In_l auch zu mehr ausgehenden Variablenbelegungen Out_l (analog monoton fallend).

3. Fall: alle anderen SPDS-*Anweisungen* (*Prozeduraufruf* „$l : p(x_1, x_2, \ldots, x_n);$“, Prozedurende „$l : return;$“ und Verzweigung „$l : if\ (e)\ goto\ l';$“) In diesen Fällen werden die eingehenden *Variablenbelegung*en nicht verändert ($Out_l = In_l$). Daher führen mehr bzw. weniger eingehende *Variablenbelegung*en In_l auch zu mehr bzw. weniger ausgehenden Variablenbelegungen Out_l (monoton steigend und fallend).

Demnach stellt die *Wertebereichsanalyse* ein Monotones Rahmenwerk dar.

□

Lemma 4.3.10 auf Seite 63 (Äquivalenzanalyse ist Monotones Rahmenwerk)
$L = (2^{Expr'^2}, \subseteq)$ ist ein Verband. Die *Ausdrücke* zur Definition der Datenflussgleichungen von $EqOut_l$ und $EqIn_l$ aus Definition 4.3.11 auf Seite 58 sind monoton.

Beweis (Skizze)
Der Beweis wird analog zum Beiweis von Lemma 4.3.9 geführt. Analog gilt hier Monotonie in folgendem Sinn: je mehr eingehende Äquivalenzen es gibt, desto mehr ausgehende Äquivalenzen wird es (potentiell) geben.
Die Menge aller Äquivalenzen $Expr'^2$ ist endlich, da es nur endlich viele Variablen gibt, diese immer einen endlichen Typ haben und $\mathbb{Z}'$ endlich ist. Auf der Teilmengenbeziehung $\subseteq$ bilden daher derartige Äquivalenzrelationen über solche Ausdrücke offensichtlich einen Verband.
Die Joins aus Definition 4.3.10 auf Seite 57 und insbesondere der Operator $\cdot\big|_{\cdot}^{\cdot}$ bestehenen aus Schnitten von Äquivalenzen. Vereinigungen erfolgen nur über disjunkte Variablenmengen (lokale vs. globale Variablen bei *Prozeduraufruf* $EqOut_k^{Call}$ und Prozedurrückkehr $RetContEqJoin_l$). Daher sind die Joins monoton.
$EqIn_l$ ist ebenfalls ein Schnitt von Joins und darum ebenso monoton. Für $EqOut_l$ wird wieder folgende Fallunterscheidung betrachtet:

1. Fall: Choose-*Zuweisung* „$l : x = choose(a, b);$“
   Für monoton steigende eingehende Äquivalenzen $EqIn_l \subseteq EqIn'_l$ ist

$$\begin{aligned}
EqOut_l &= EqIn_l\Big|_{Vars}^{\{x\}} \\
&= EqIn_l \cap \{e \in Expr' \mid Vars(e) \subseteq Vars\}^2 \cap \{e \in Expr' \mid Vars(e) \cap \{x\} = \emptyset\}^2 \\
&\subseteq EqIn'_l \cap \{e \in Expr' \mid Vars(e) \subseteq Vars\}^2 \cap \{e \in Expr' \mid Vars(e) \cap \{x\} = \emptyset\}^2 \\
&= In'_l\Big|_{Vars}^{\{x\}} \\
&= EqOut'_l.
\end{aligned}$$

   Damit ist $EqOut_l$ monoton steigend. Analog kann monoton fallend gezeigt werden.

2. Fall: *Zuweisung* „$l : x = e$“.
   Es ist dann $EqOut_l =< \{(x,e)\} \cup EqIn_l\big|_{Vars}^{\{x\}} >$. Da die Vereinigung von Mengen monoton ist und bereits die Monotonie von $EqIn_l\big|_{Vars}^{\{x\}}$ gezeigt wurde, bleibt zu zeigen, dass auch der Hüllenoperator $< \cdot >$ monoton ist. Dies kann leicht nachgewiesen werden. Damit ist auch in diesem Fall $EqOut_l$ monoton.

3. Fall: alle anderen SPDS-*Anweisungen* (*Prozeduraufruf* „$l : p(x_1, x_2, \ldots, x_n);$“, Prozedurende „$l : return;$“ und Verzweigung „$l : if\ (e)\ goto\ l';$“)
   In diesen Fällen werden die eingehenden Äquivalenzen nicht verändert ($EqOut_l = EqIn_l$). Daher führen mehr bzw. weniger eingehende Äquivalenzen $EqIn_l$ auch zu mehr bzw. weniger ausgehenden Variablenäquivalenzen $EqOut_l$ (monoton steigend und fallend).

Demnach stellt die Äquivalenzanalyse ein Monotones Rahmenwerk dar.

□

Satz 4.3.8 auf Seite 75
Die *SPDS*-Beschneidung T_{slice} ist CTL*-invariant für gegebenes $\phi \in$ CTL*.

Beweis (Skizze)
Seien ein *SPDS* S sowie ein Vorwärts-Slice $\Lambda_+ \in Slices(S)$ und ein Rückwärts-Slice $\Lambda_- \in Slices(S)$ gegeben. Sei weiter $\gamma[\![S]\!]_\phi = (P, \Gamma, \hookrightarrow, I, L_\phi)$ das von S beschriebene *PDS* und $\gamma[\![S']\!]_\phi = (P', \Gamma', \hookrightarrow', I', L'_\phi)$ das von $S' := T_{slice}(S)$ beschriebene *PDS*.

zu zeigen: $\gamma[\![S']\!]_\phi$ simuliert alle essenziellen Läufe von $\gamma[\![S]\!]_\phi$ bezüglich $L0$ bzw. $L0_\phi$, d.h. $T_{slice}(S)$ simuliert alle essenziellen Läufe von S bezüglich $L0$ ($L0 = L0_\phi$, falls Formel ϕ gegeben).
Wähle $R_{slice} \subseteq (P \times \Gamma^*) \times (P' \times \Gamma'^*)$ gemäß Definition 3.3.10 auf Seite 40, so dass:

$$R_{slice} = \{(s, s') \mid s = (p, \gamma_1\gamma_2 \ldots, \gamma_m) \in P \times \Gamma^*, s' = (p', \gamma'_1\gamma'_2 \ldots, \gamma'_m) \in P' \times \Gamma'^*,$$
$$env^s_{Vars(S')} = env^{s'}_{Vars(S')}, \forall i : env^{\gamma_i}_{Vars(S')} = env^{\gamma'_i}_{Vars(S')}, Marken(\gamma_i) = Marken(\gamma'_i), m \in \mathbb{N}_{\geq 0}\}.$$

Wegen $env^s_{Vars(S')} \subseteq env^s_{Vars(S)}$ und $env^{\gamma_i}_{Vars(S')} \subseteq env^{\gamma_i}_{Vars(S)}$, fallen damit *Konfigurationen* mit unterschiedlichen Variablenwerten für eliminierte Variablen $v \in Vars(S) \setminus Vars(S')$, d.h. $v \notin \Lambda_+ \cap \Lambda_-$ zusammen in eine Äquivalenzklasse.
Sei $s_1 \to s_2 \to \cdots \to s_n$ ein beliebiger essenzieller Lauf von P (bzw. S). D.h. es gilt $s_n \rightsquigarrow L0$ (damit gilt bereits $s_i \rightsquigarrow L0$). Andere Läufe brauchen nicht simuliert werden, da sie keinen Einfluss auf die *Erreichbarkeit* von Marken $L0$ und keinen Einfluss auf den *Wahrheitsgehalt* von ϕ haben (nicht essenziell).
Behauptung: Es gibt einen Lauf $s^k_1 \to s^k_2 \to \cdots \to s^k_n$ von P' (bzw. $T_{slice}(S)$), so dass gilt: $\forall i : (s_i, s^k_i) \in R_{slice}$ nach k Anwendungen der *Modelltransformationsregeln* $(S_1), (S_2), (S_3), (S_4), (S_5)$ und (S_6).
Dies wird per doppelter Induktion über die Anzahl k der Transformationsregelanwendungen und der Länge des Laufs n gezeigt.
Ind. Anfang: $k = 1, n = 1$.

- 1. Fall $(S_1), (S_2), (S_3)$: Durch die Regeln $(S_1), (S_2), (S_3)$ werden nur *Konfigurationenübergänge* geändert. Der *Konfigurationenraum* bleibt unverändert, so dass dann zunächst für die erste *Konfiguration* $s^1_1 = s_1$ gilt. Der Lauf hat die Länge 0, so dass es keine *Konfigurationenübergänge* gibt, welche durch eine Transformation beeinflusst werden könnten. Es gilt dann $(s_1, s^1_1) \in R_{slice}$ mit $s^1_1 \in I'$.

- 2. Fall $(S_4), (S_5), (S_6)$: Die Transformation

$$p(x_1, x_2, \ldots, x_m) \Rrightarrow p(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots x_m)$$

ändert nicht den *Konfigurationenraum* (mögliche *Variablenbelegung*en) des Modells, sondern lediglich Konfigurationenübergänge. Wegen $Vgbl \Rrightarrow Vgbl \setminus v$ bzw. $Param_p \Rrightarrow Param_p \setminus v$ bzw. $Vlcl_p \Rrightarrow Vlcl_p \setminus v$ ist dann $Vars(S') = Vars(S) \setminus v$. Es können somit für s_1^1 die restlichen *Variablenbelegung*en (ohne v) von s_1 verwendet werden, so dass gilt $env^{s_1}_{Vars(S')} = env^{s_1^1}_{Vars(S')}$ und $\forall i : env^{\gamma_i}_{Vars(S')} = env^{\gamma_i'}_{Vars(S')}$ (gemäß Definition R_{slice}). Dann ist $(s_1, s_1^1) \in R_{slice}$ und $s_1^1 \in I'$.

Ind. Behauptung $(k \to k+1)$: $\exists s_j^{k+1} : (s_j, s_j^{k+1}) \in R_{slice} \wedge s_1^{k+1} \to s_2^{k+1} \cdots \to s_j^{k+1}$ ist Lauf von P nach $k+1$ Transformationsregelanwendungen.
Ind. Schritt $(k \to k+1)$: Die Behauptung wird nun gezeigt per geschachtelter Induktion für die Lauflänge n.
Ind. Anfang: $n = 1$. Analoge Betrachtung wie für den Ind. Anfang $k = 1, n = 1$. Bei $(S_1), (S_2), (S_3)$ wird der *Konfigurationenraum*[1] nicht verändert und bei $(S_4), (S_5), (S_6)$ wird jeweils eine Variable v eliminiert, so dass sich ausgehend von s_1^k entsprechende *Variablenbelegung*en für die *Konfiguration* s_1^{k+1} ergeben mit $(s_1, s_1^{k+1}) \in R_{slice}$ und $s_1^{k+1} \in I'$.
Ind. Behauptung $(n \to n+1)$: $\exists s_{n+1}^{k+1} : (s_{n+1}, s_{n+1}^{k+1}) \in R_{slice} \wedge s_1^{k+1} \to s_2^{k+1} \cdots \to s_{n+1}^{k+1}$ ist Lauf von P nach $k+1$ Transformationsregelanwendungen.
Ind. Vorraussetzung: $\forall i \leq k : \forall j \leq n : \exists s_j^i : (s_j, s_j^i) \in R_{slice} \wedge s_1^i \to s_2^i \cdots \to s_j^i$ ist Lauf von P nach i Transformationsregelanwendungen.
Ind. Schritt $(n \to n+1)$:

- 1. Fall (S_1) : Wiederum bleiben die Variablen samt *Variablenbelegung*en unverändert. Sei $\{l_j\} = Marken(Kopf(s_j))$. Dann sei zur Regelanwendung (S_1) die Voraussetzung angenommen, es gelte $l_n : s \notin \Lambda_+ \cap \Lambda_-$. Wegen $s_1^k \to s_2^k \cdots \to s_n^k$ und des Laufs $s_1 \to s_2 \cdots \to s_n$ von $\gamma[\![S]\!]_\phi$ ist $l_n \in \Lambda_+$.
  - 1.1. Fall: $\exists s_{n+1} : s_n \to s_{n+1}$. Dann ist $l_{n+1} \in \Lambda_+$ und damit $l_n : s \in \Lambda_+$ (Slicingeigenschaft). D.h. es gilt $l_n : s \notin \Lambda_-$. Dies ist aber ein Widerspruch zur Essenzialität $l_{n+1} \rightsquigarrow L0$. Damit ist $l_n : s \in \Lambda_+ \cap \Lambda_-$. D.h. (S_1) wird in diesem Fall nicht auf essenziellen Läufen angewendet. Die SPDS-*Anweisung* $l_n : s$ hat dann keinen Einfluss auf den *Wahrheitsgehalt* von ϕ bzw. kann ausgehend von ihr keine Marke aus $L0$ mehr erreicht werden ($l_{n+1} \not\Rightarrow L0$). Es kommt durch die Eliminationen zu keinen neuen Läufen, welche den *Wahrheitsgehalt* von ϕ beeinflussen. Die Elimination von SPDS-*Anweisungen* (analog Prozeduren und Marken) führt daher zu keiner Änderung des *Modellprüfung*sergebnisses.
  - 1.2. Fall: $\nexists s_{n+1} : s_n \to s_{n+1}$. In diesem Fall bestehen je nach Präzision der Slicings die Möglichkeiten:
    - Möglichkeit (a): $l_n : s \in \Lambda_+ \cap \Lambda_-$. Dann wird aber wie im Fall 1.1. die Regel (S_1) nicht auf dem essenziellen Lauf angewendet.
    - Möglichkeit (b): $l_n : s \notin \Lambda_+ \cap \Lambda_-$. Dann wird die Regel zwar nicht auf dem essenziellen Lauf, aber direkt danach angewendet.
- 2. Fall $(S_2), (S_3)$: Analog zu Fall 1. werden diese Regeln nicht auf einem essenziellen Lauf bezüglich $L0$ angewendet.
- 3. Fall $(S_4), (S_6)$: Entsprechend Betrachtungen zu Fall 1. existiere o.B.d.A. ein s_{n+1} derart, dass gilt $s_n \to s_{n+1}$. Wiederum wegen Variablenmengenverkleinerung $Vgbl \Rrightarrow Vgbl \setminus v$ bzw. $Vlcl_p \Rrightarrow Vlcl_p \setminus v$ ist es möglich, ausgehend von s_{n+1}^k ein s_{n+1}^{k+1} derart zu erhalten, dass gilt $\forall x \in Vars(S') : [\![x]\!]_a = [\![x]\!]_b$ mit *Variablenbelegung*en $a = env^{s_{n+1}^k}_{Vars(S')} = env^{s_{n+1}^{k+1}}_{Vars(S')} \subseteq env^{s_{n+1}}_{Vars(S)} = b$ etc. (gemäß Definition R_{slice}). Damit ist $(s_{n+1}, s_{n+1}^{k+1}) \in R_{slice}$. Wegen $v \notin \Lambda_+ \cap \Lambda_-$ findet die Variable v bei keinem *Konfigurationenübergang* des essenziellen Laufs *Verwendung*. Wegen $(S_1), (S_2)$ und (S_3) wird v auch nicht mehr im Restmodell S^k verwendet

[1] Variablen samt *Variablenbelegung*en

und ist daher überflüssig. Zusammen mit $s_n^k \to s_{n+1}^k$ (i.V.) gilt dann auch $s_n^{k+1} \to s_{n+1}^{k+1}$. Damit ist nach Ind. Vorr. insbesondere $s_1^{k+1} \to s_2^{k+1} \cdots \to s_n^{k+1} \to s_{n+1}^{k+1}$ ein Lauf von $\gamma[\![S]\!]_\phi$ nach $k+1$ Transformationsregelanwendungen.

- 4. Fall (S_5) Analog zum Fall 3. wird die Variablenmenge verkleinert. Entsrpechende Überlegungen übertragen sich. Jedoch wird der *Konfigurationenübergang* $s_n^{k+1} \to s_{n+1}^{k+1}$ leicht verändert. Nach Ind. Vorr. ist $s_n^k \to s_{n+1}^k$. Bei Elimination der Parametervariablen v_i sind auch sämtliche *Prozeduraufrufe* anzupassen, so dass für alle möglichen Prozeduraufrufe an der Marke l_n gilt: $env_{Vars(S')}^{s_{n+1}^k} = env_{Vars(S')}^{s_{n+1}^{k+1}}$ gilt. Dies führt analog zu $(s_{n+1}, s_{n+1}^{k+1}) \in R_{slice}$ mit $s_n^{k+1} \to s_{n+1}^{k+1}$. Damit ist nach Ind. Vorr. insbesondere $s_1^{k+1} \to s_2^{k+1} \cdots \to s_n^{k+1} \to s_{n+1}^{k+1}$ ein Lauf von $\gamma[\![S]\!]_\phi$ nach $k+1$ Transformationsregelanwendungen.

Damit ist die Behauptung für alle Fälle der Transformationsregeln gezeigt. Die Umkehrung, $\gamma[\![S]\!]_\phi$ simuliert alle Läufe von $\gamma[\![S']\!]_\phi$ (d.h. S simuliert alle Läufe von $T_{slice}(S)$), ist trivial, da $T_{slice}(S)$ nach der Beschneidung ein Teilmodell von S ist. Daher ist $\gamma[\![S]\!]_\phi$ bisimular zu $\gamma[\![S']\!]_\phi$ für alle essenziellen Läufe bezüglich $L0$. Gemäß Betrachtungen aus Abschnitt 3.3.5 ab Seite 37 folgt damit die CTL*-Invarianz. Es sei bemerkt, dass der Simulationsbegriff und auch die Laufsäquivalenz die Beibehaltung sämtlicher Annotationsausdrücke verlangt. Wegoptimierte Variablen sind aber entweder nicht *erreichbar* oder werden nicht in der *temporalen Formel* ϕ verwendet, so dass diese auch nicht in der Annotationsfunktion $L_\phi(\cdot)$ auftreten. Wenn eine eliminierte Variable v nicht *erreichbar* ist, so hat sie wegen der *Konservativität* des Slicings auch keinen Beitrag am *Wahrheitsgehalt* einer *temporalen Formel* ϕ (mit $v \in \phi$), so dass für alle realisierbaren *Variablenbelegung*en $env \in ENV$ in S gilt: $[\![v]\!]_{env} = \bot$. Durch Elimination von v aus S ändert sich daran nichts, da auch für jede realisierbare *Variablenbelegung* $env \in ENV$ in $T_{slice}(S)$ gilt: $[\![v]\!]_{env} = \bot$.

□

Lemma 4.3.13 auf Seite 83 (*Konservative Seiteneffektfreiheit* τ_{eff}]
Eine Prozedur p ist für die approximierten *Variablenbelegung*en $RENV_{l_p}^*$ *seiteneffektfrei*

$$\tau_{eff}(RENV_{l_p}^*, p) = false,$$

falls p keine *Zuweisung* an globale Variablen enthält und alle von p potentiell aufrufbaren Prozeduren q *Seiteneffekt* frei sind für die approximierten *Variablenbelegung*en $RENV_{l_q}^*$, d.h.

$$\tau_{eff}(RENV_{l_q}^*, q) = false.$$

Beweis (Skizze)
Sei angenommen, die Prozedur p rufe höchstens *Seiteneffekt* freie Prozeduren auf. Weiter gelte die Voraussetzung, dass p keine *Zuweisung* an globale Variablen aufweist. Ohne solche *Zuweisung*en können die *Variablenbelegung*en für globale Variablen innerhalb von p nicht verändert werden. Die Betrachtung sämtlicher *Konfigurationenübergänge* in p ergibt:

1. Fall *Zuweisung* „$x = e$;“
Nach Voraussetzung ist $x \notin Vgbl$ und daher wird beim *Konfigurationenübergang* im zu Grunde liegendem *PDS* keine *Variablenbelegung* für eine globale Variable geändert.

2. Fall *Prozeduraufruf* „$q(x_1, x_2, \ldots, x_m)$;“
Nach Voraussetzung ist q *seiteneffektfrei* für die *Variablenbelegung*en $RENV_{l_q}^*$. Diese *Variablenbelegung*en und nicht mehr sind aber gemäß *Wertebereichsanalyse* aus den Variablenbelegungen $RENV_{l_p}^*$ an der Marke l_p höchstens realisierbar (evtl. auch nicht wegen *Konservativität*). Daher verändert der *Prozeduraufruf* „$q(x_1, x_2, \ldots, x_m)$;“ keine globalen *Variablenbelegung*en im zu Grunde liegendem *PDS*.

3. Fall Prozedurende „*return*;“ sowie Verzweigung „$if\ (e)\ goto\ l$;“
Wegen $l \in Marken(p)$, ist die Verzweigung ein intraprozeduraler bedingter Sprung. Bei *Konfigurationenübergang* verbleibt die weitere Abarbeitung in der Prozedur p. Beide *Konfigurationenübergänge* Prozedurende und Verzweigung des zu Grunde liegendem *PDS* ändern daher keine *Variablenbelegung*en. Und darum auch nicht für globale Variablen.

Somit bleiben in allen Fällen die globalen *Variablenbelegung*en erhalten und die Prozedur p ist gemäß Definition 4.3.18 auf Seite 80 *Seiteneffekt* frei bezüglich $RENV^*_{l_p}$.

□

Lemma 4.3.14 auf Seite 83 (*Konservative* Versackungsfreiheit τ_{stop})
Eine Prozedur p ist für beliebige *Variablenbelegung*en $W \subseteq ENV$ versackungsfrei $\tau_{stop}(W, p) = false$, wenn p keinen arithmetischen Überlauf und keine Division mit 0 sowie keine Schleifen[2] enthält und alle von p aufgerufenen Prozeduren ebenfalls versackungsfrei sind.

Beweis (Skizze)
Seien wie beim Beweis von τ_{eff} wieder die Voraussetzungen angenommen. Dann werden wieder alle möglichen *Konfigurationenübergänge* in p an der Marke $l \in Marken(S)$ betrachtet:

1. Fall *Zuweisung* „$x = e$;“
Per Induktion über die Anzahl der Operationen wird die erfolgreiche *Auswertung* des Ausdrucks e deutlich. Konstanten $e = c \in \mathbb{Z}$ sind offensichtlich *auswertbar* und *SPDS*-Variablen $e = v \in Vars(S)$ sind entsprechend ihrer *Variablenbelegung*en auszuwerten (Induktionsanfang). Für die Basisoperationen $-, *, \&, <$ sowie !, d.h. die Fälle $e = e_1 - e_2, e = e_1 * e_2, e = e_1 \& e_2, e = e_1 < e_2$ sowie $e = !e_2$ ist die erfolgreiche *Auswertung* ebenso ersichtlich, da diese für beliebige Wertbelegungen der *Ausdrücke* e_1 und e_2 gültige Werte liefern. Für die Basisoperation /, d.h. der Fall $e = e_1/e_2$ ist eine mögliche Division mit 0 undefiniert und würde zu $\perp$ in der *Auswertung* führen. Allerdings ist dieser Fall ($[\![e_2]\!]_{env} = 0$) per Voraussetzung ausgeschlossen. Damit führt die *Auswertung* von e in jedem Fall, d.h. für jedes $env \in RENV_l$ zu gültigen Werten $[\![e]\!]_{env} \in \mathbb{Z}$. Weiter sind per Voraussetzung arithmetische Überläufe ausgeschlossen, so dass bei der *Zuweisung* „$x = e$“ der Typ von x hinreichend groß ist. D.h. es gilt $[\![e]\!]_{env} \in range(x)$. Somit ist dies eine gültige *Zuweisung*. Ein Lauf mit diesem *Konfigurationenübergang* wird entsprechend nicht versacken und eine *Folgekonfiguration* liefern.

2. Fall *Prozeduraufruf* „$q(x_1, x_2, \ldots, x_m)$;“
O.B.d.A. ist $p \neq q$, da andernfalls der interprozeduraler Kontrollflussgraph beginnend bei l_p einen Zyklus hätte. Nach Voraussetzung ist q nicht versackend für die *Variablenbelegung*en $RENV^*_{l_q}$, d.h. auch nicht der *Prozeduraufruf* „$q(x_1, x_2, \ldots, x_m)$;“ mit den *Variablenbelegung*en $RENV_l$. Jeder dort beginnende Lauf wird nach Voraussetzung auch wieder aus der Prozedur q austreten, was zu einer gültigen *Nachfolgekonfiguration* des Laufs in der Prozedur p führt.

3. Fall Prozedurende „*return*;“ sowie Verzweigung „$if\ (e)\ goto\ l$;“
Wegen $l \in Marken(p)$ ist die Verzweigung ein intraprozeduraler bedingter Sprung. Da nach Voraussetzung p keine Schleifen enthält, muss der intraprozedurale Kontrollflussgraph von p ein Baum sein. Die Wurzel ist der Prozedureintritt l_p. *Konfigurationenübergänge* innerhalb von p sind dann Pfade bzw. Läufe des zu Grunde liegenden *PDS*, in welchem keine Marke mehrfach auftritt (keine Kreise). Da die Prozedur p nur über endlich viele Marken verfügt, muss ein Lauf ebenso nach endlich vielen Schritten wieder aus der Prozedur p austreten, da ein solcher nicht terminiert (analog Fall 1 kann der Ausdruck e erfolgreich ausgewertet werden).

Somit gibt es in allen Fällen für einen Lauf eine *Nachfolgekonfiguration* innerhalb der Prozedur p oder durch Beendigung der Prozedur. Die Prozedur p ist daher gemäß Definition 4.3.20 auf Seite 80 versackungsfrei bezüglich $RENV^*_{l_p}$.

[2] zyklenfreier **inter**prozeduraler Kontrollflussgraph beginnend bei l_p (Zyklus entsteht z.B. bei rekursivem Aufruf)

□

Lemma 4.3.19 auf Seite 101 (*Konservative* Versackungsfreiheit τ_{stop})
Eine SPDS-*Anweisung* $l : z \in Stats(S)$ eines *SPDS* S ist für beliebige *Variablenbelegung*en $W \subseteq ENV$ versackungsfrei $\tau_{stop}(W, z) = false$, wenn z keinen arithmetischen Überlauf und keine Division mit 0 sowie keinen Zyklus[3] enthält.

Beweis (Skizze)
Seien analog zum Beweis von τ_{stop} für Prozeduren in Lemma 4.3.14 sämtliche Voraussetzungen angenommen. Dann werden wieder alle möglichen *Konfigurationenübergänge* an der Marke $l \in Marken(S)$ betrachtet:

1. Fall *Zuweisung* „$x = e$;“
Der Ausdruck $e \in Expr$ ist mit allen *Variablenbelegung*en $env \in W$ *auswertbar*. Die Schlussfolgerung dazu ist die gleiche wie im Beweis von Lemma 4.3.14.

2. Fall *Prozeduraufruf* „$q(x_1, x_2, \ldots, x_m)$;“
Da für sämtliche x_i gilt $bits(x_i) \leq bits(q_i)$ (wobei $Param_q = [q_1, q_2, .., q_n)$, kommt es zu einem gültigen *Prozeduraufruf*. $l : z$ enthält nach Voraussetzung keinen Zyklus. D.h. z enthält einerseits keinen Sprung an l und andererseits ist l nicht die Anfangsmarke einer Prozedur q wenn z einen *Prozeduraufruf* an q darstellt. Daher erfolgt ein Übergang in die Prozedur q an die Marke $l_q \neq l$, d.h. ein erfolgreicher *Konfigurationenübergang* eines Laufs in eine Nachfolgerkonfiguration.

3. Fall Prozedurende „*return*;“
Ein *Konfigurationenübergang* zu einer *Nachfolgekonfiguration* erfolgt bedingungslos. Auch im Fall der Beendigung der Hauptprozedur erfolgt der Übergang von einer *Konfiguration* mit einem Kellerelement zu der *Finalkonfiguration* ohne Kellerelement.

4. Fall Verzweigung „$if\ (e)\ goto\ l'$;“
Nach Fall 1 ist e *auswertbar*, weswegen es für einen Lauf mit $env \in RENV_l$ zu einem Sprung kommt, falls $[\![e]\!]_{env} = 1$. Ist $[\![e]\!]_{env} = 0$, so wird mit der nachfolgenden SPDS-*Anweisung* fortgesetzt, d.h. ein gültiger *Konfigurationenübergang* zu einer *Nachfolgekonfiguration* erfolgt. Ist andererseits $[\![e]\!]_{env} = 1$, so wird mit der SPDS-*Anweisung* an der Marke l' fortgesetzt. $l : z$ bzw. „$l : if\ (e)\ goto\ l'$;“ enthält nach Voraussetzung keinen Zyklus. D.h. z enthält keinen Sprung an l. Daher ist $l \neq l'$ und es erfolgt ein gültiger *Konfigurationenübergang* zu einer Nachfolgekonfiguration.

Somit gibt es in allen Fällen für einen Lauf an die *Konfiguration* $s \in Konf(S)$ mit der SPDS-*Anweisung* $l : z$ eine *Nachfolgekonfiguration* $s' \in Konf(S)$, so dass $s \hookrightarrow_l s'$. Die SPDS-*Anweisung* $l : z$ ist daher gemäß Definition 4.3.19 auf Seite 80 versackungsfrei bezüglich W.

□

Lemma 4.3.21 auf Seite 108 (Eliminierbarkeit ⇒ Redundanz)
Ist eine Variable $v \in Vars(S)$ eliminierbar, so ist sie auch *redundant*. Die Umkehrung gilt nicht.

Beweis (Skizze)
Sei $v \in Vars(S)$ eine eliminierbare *SPDS*-Variable ($elim_\phi(v) = true$ bzw. $elim_{L0}(v) = true$). O.b.d.A. gelte $elim_\phi(v) = true$. Dann ist wegen $\bigwedge_{l:s \in Stats(S), used_l(v)} elim_\phi(l : s) = true$ auch $\forall l : s \in Stats(S) : used_l(v) \Rightarrow elim_\phi(l : s)$. D.h. jede SPDS-*Anweisung*, in welcher v verwendet wird ist ebenfalls eliminierbar. Daher kann dort ohne Änderung des *Modellprüfung*sresultats die *Modelltransformation* T_{elim} aus Definition 4.3.29 auf Seite 93 angewendet werden. Durch sukzessive Anwendung von T_{elim} entsteht ein *SPDS* S' ohne *Verwendungen* von v. Da nun insbesondere $v \notin \phi$ (denn v ist eliminierbar), ist der Variablenwert von v unwichtig für die *Modellprüfung*. v kann daher

[3] $l : z$ enthält einen Zyklus, wenn z.B. z ein Sprung an l enthält oder l die Anfangsmarke einer Prozedur q ist und z ein *Prozeduraufruf* an q

an jeder *Konfiguration* im *Konfigurationenraum* beliebig gewählt werden ohne die *Modellprüfung* zu beeinflussen. Daher ist v *redundant*.
Für die Umkehrung wird folgender Ausschnitt eines *SPDS* mit einer entsprechend *redundant*en Variable x betrachtet:

```
l: x = 5/0;
Error: goto Error;
```

Dann führt die Entnahme der *SPDS-Zuweisung* mit der *redundant*en Variablen x mittels *Modelltransformation* T_{elim} aus Definition 4.3.29 auf Seite 93 zu dem *SPDS*

```
l: skip;
Error: goto Error;
```

Da im oberen *SPDS* stets eine arithmetische Ausnahme auftritt (Division mit Null), findet ein *Modellprüfer* im ersten Beispiel keine *Nachfolgekonfiguration* nach der Marke l und bricht die Durchsuchung des *Konfigurationenraum*s ab ohne jemals die Marke *Error* zu erreichen. Im zweiten Fall kommt die Variable x gar nicht mehr im Modell vor und ein vorzeitiges Terminieren von Modellpfaden durch Arithmetik kann nicht mehr geschehen. Daher ist die *Zuweisung* $x = 5/0$ nicht eliminierbar und damit auch x nicht (zumindest nicht auf diese Weise mittels T_{elim}), obwohl sie *redundant* ist.

□

Lemma 4.3.22 auf Seite 108 (Hinreichende Eliminierbarkeit)
Ist die *SPDS*-Variable $x \in Vars(S)$ *redundant* $\tau_{rdz}(x) = true$ und „$l : x = e;$“ $\in Stats(S)$ versackungsfrei $\tau_{stop}(RENV_l, „l : x = e;“) = false$, so ist „$l : x = e;$“ $\in Stats(S)$ eliminierbar $elim_\zeta(„l : x = e;“) = true$ im *SPDS* S, wobei $\zeta \in \{\phi, L0\}$.

Beweis (Skizze)
Seien hierzu die *SPDS*-Variable x *redundant* $\tau_{rdz}(x) = true$ und „$l : x = e;$“ versackungsfrei $\tau_{stop}(RENV_l, „l : x = e;“) = false$. Es ist zu zeigen, dass $elim_\zeta(„l : x = e;“) = true$ gilt. Die Eliminierbarkeit verlangt, dass das *Modellprüfung*sresultat durch die *Modelltransformation* T_{elim} nicht beeinflusst wird. Für den Nachweis werden die Läufe des interpretierenden *PDS* $\gamma[\![S]\!]_\phi$ vom *SPDS* S und des *PDS* $\gamma[\![S']\!]_\phi$ vom *SPDS* $T_{elim}(S)$ betrachtet. Ich werde zeigen, dass sich das *Modellprüfung*sresultat wegen Bisimulation der Läufe nicht ändert. Zwei *Konfigurationen* $(g, \gamma_1\gamma_2 \dots \gamma_m) \in P \times \Gamma^*$ des *PDS* $\gamma[\![S]\!]_\phi$ und $(g', \gamma'_1\gamma'_2 \dots \gamma'_m) \in P' \times \Gamma^{*'}$ des *PDS* $\gamma[\![S']\!]_\phi$ seien dann bisimular (dies überträgt sich automatisch auf Läufe), falls sie sich lediglich in der *Variablenbelegung* von x unterscheiden. Je nachdem, ob x eine lokale oder globale Variable ist betrifft dies g' bzw. ein γ'_i. Als Bisimulationsrelation ergibt sich damit[4]:

$$R_{elim} := \{(z, z|_x^c) \in (P \times \Gamma^*) \times (P' \times \Gamma^{*'}) \mid c \in range(v)\} \subseteq Konf(S) \times Konf(T_R(S)).$$

Da die *Erreichbarkeitsprüfung* auf temporale *Modellprüfung* reduzierbar ist (siehe Abschnitt 3.3.3 ab Seite 35), sei o.B.d.A. der allgemeinere Fall $\zeta = \phi$ betrachtet. Wegen $\tau_{rdz}(x) = true$, ist $\forall l' \in Marken(S) : \forall env \in RENV_{l'} : \forall e' \in range(v) : ((env, l') \models \phi) \Leftrightarrow ((env|_v^{e'}, l') \models \phi)$. Wegen $\tau_{stop}(RENV_l, „l : x = e;“) = false$, kommt es bei der *Auswertung* von e nicht zu einer Versackung, d.h. es terminiert der betrachtete Lauf in $\gamma[\![S]\!]_\phi$ nicht. Durch die SPDS-*Anweisung skip* nach der *Modelltransformation* T_{elim} kommt es ebenfalls nicht zur Versackung des Laufs in $\gamma[\![S']\!]_\phi$. Sei daher ein beliebiger Lauf $z_0 \to_{\gamma[\![S]\!]_\phi} z_1 \cdots \to_{\gamma[\![S]\!]_\phi} z_n \to_{\gamma[\![S]\!]_\phi} z$ des interpretierenden *PDS* $\gamma[\![S]\!]_\phi$ betrachtet mit erstem Auftreten der Marke $l \in Marken(Z_n)$ welcher nach obiger Überlegung nicht an der Marke z_n versackt (d.h. $\forall i < n : l \notin Marken(z_i)$). Durch Anwendung der *Modelltransformationsregel* T_{elim} verändert sich der letzte Konfigurationenübergang in $\gamma[\![S']\!]_\phi$ zu dem Lauf $z_0 \to_{\gamma[\![S']\!]_\phi} z_1 \cdots \to_{\gamma[\![S']\!]_\phi} z_n \to_{\gamma[\![S']\!]_\phi} z|_x^c$ mit $c = [\![x]\!]_{env^{z_n}}$, so dass der Wert von x bei *Konfigurationenübergang* sich nicht ändert. Wegen $\tau_{rdz}(x) = true$ ist aber die Belegung der Variable x unerheblich für die *Modellprüfung*, so dass $((env, l') \models \phi) \Leftrightarrow ((env|_v^{e'}, l') \models \phi)$ auch für

[4] Schreibweise siehe Gleichung 3.1 auf Seite 30

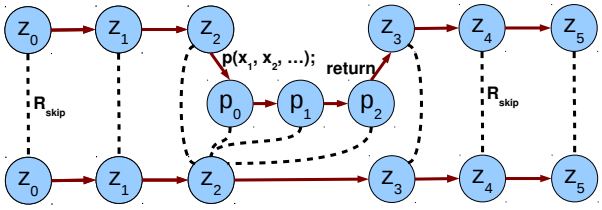

Abbildung B.1.: Veranschaulichung der Relation R_{skip} am Beispiel

unverändertes x an der Nachfolgemarke l' von l in $\gamma[\![S']\!]_\phi$ gilt und damit $(z, z|_x^c) \in R_{elim}$. D.h. der Lauf in $\gamma[\![S']\!]_\phi$ simuliert den Lauf in $\gamma[\![S]\!]_\phi$ an dieser veränderten Stelle und per Induktion auch auf den Rest de Laufs und für beliebige Läufe. Somit ist $(S \models \phi) \Leftrightarrow (T_{elim}(S) \models \phi)$. D.h. die *SPDS-Zuweisung* „$l : x = e;$“ ist eliminierbar $elim_\zeta(\text{„}l : x = e;\text{“}) = true$.

□

Satz 4.3.14 auf Seite 87
Die Prozedurüberbrückung T_{skip} ist CTL*-X-invariant für gegebenes $\phi \in$ CTL*-X und nicht LTL-invariant.

Beweis (Skizze)
Zu zeigen: T_{skip} ist nicht LTL-invariant.
Hierzu genügt die Angabe eines Gegenbeispiels. Betrachtet sei das Beispiel aus Abbildung 4.3 auf Seite 79 bzw. Abbildung B.1 sowie die LTL-Formel $\phi = XXXXXXXXl_5 = X^8 l_5$ für $l_5 \in Marken(z_5)$. Sie gilt lediglich an der *Initialkonfiguration* z_0 $(S \models \phi)$, da nach stets genau 8 *Konfigurationenübergängen* die Marke l_5 der *Konfiguration* z_5 erreicht wird. Durch Prozedurüberbrückung ist z_5 aber nach 5 statt 8 Konfigurationenübergängen *erreichbar*, was den *Wahrheitsgehalt* ändert $(T_{skip}(S) \not\models \phi)$. T_{skip} ist daher nicht LTL-invariant.

Sei nun $\gamma[\![S]\!]_\phi$ das interpretierende *PDS* des *SPDS* S und $\gamma[\![S']\!]_\phi$ das interpretierende *PDS* des *SPDS* $S' = T_{skip}(S)$.
zu zeigen: $\gamma[\![S']\!]_\phi$ stotter-simuliert $\gamma[\![S]\!]_\phi$.
Der Beweis verläuft ähnlich zur *SPDS*-Beschneidung per Induktion über die Anzahl der Anwendungen der *Modelltransformation* (4.19 auf Seite 87). Es ist eine Relation R_{skip} anzugeben, so dass die Eigenschaften aus Definition 3.3.12 auf Seite 40 gelten. Die Relation R_{skip} ist am Beispiel in Abbildung B.1 veranschaulich. Dabei garantiert Konfluenz (via τ_{eff} und τ_{stop}), dass im optimierten Modell nach der Prozedurüberbrückung die gleiche *Konfiguration* mit den gleichen *Variablenbelegung*en und dem gleichen Kellerinhalt vorliegt.
Keine Anwendung der *Modelltransformation* (4.19 auf Seite 87) ändert das Modell nicht (Induktionsanfang). Betrachtet sei nun eine (weitere) Anwendung der *Modelltransformation* (4.19 auf Seite 87) mit fest gewählten $p \in Prz(S)$ sowie $l \in Marken(S)$. Potentiell auftretende *Konfigurationen* (aufgespannter *Konfigurationenraum*) während der Abarbeitung des *Prozeduraufruf*es $l : p(x_1, x_2, \ldots x_n)$ an der *Konfiguration* $s \in Konf(S)$ bis zur Rückkehr aus der Prozedur werden mit $Post_p^*(s)$ bezeichnet. Für einen solchen *Prozeduraufruf* $l : p(x_1, x_2, \ldots x_n)$ einer Prozedur p an der *Konfiguration* $s = (g, \gamma_1\gamma_2 \ldots \gamma_m) \in P \times \Gamma^*$ gilt:

$$Post_p^*(s) := \{(g', \gamma_1'\gamma_2' \ldots \gamma_r'\gamma_1\gamma_2 \ldots \gamma_m) \mid (g', \gamma_1'\gamma_2' \ldots \gamma_r') \in Post^*((env^s_{Vgbl}, (env_{loc_p}, l_p)))\}, \quad \text{(B.1)}$$

wobei sich $env_{loc_p} \in RENV_l^p$ aus env^s durch *Auswerten* der Argumente des *Prozeduraufruf*s ergibt (siehe (4.18 auf Seite 81)). Dann kann R_{skip} so gewählt werden, dass nicht überbrückte *Konfigurationen* in Relation mit sich selbst sind und überbrückte Konfigurationen in Relation mit der *Konfiguration* s ihres ehemaligen *Prozeduraufruf*s sind. Sei

$$R_{skip}^{l,p} := \{(s, s') \in (P \times \Gamma^*) \times (P' \times \Gamma'^*) \mid s = s' \vee (Marken(s') = \{l\} \wedge s \in Post_p^*(s'))\}. \quad \text{(B.2)}$$

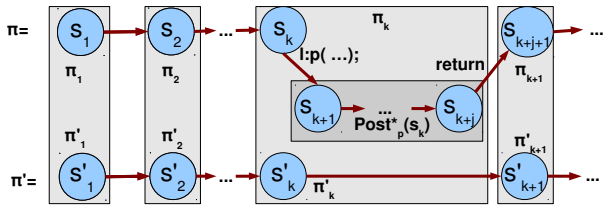

Abbildung B.2.: Zerlegung der Pfade π und π'

Für mehrere Anwendungen der *Modelltransformation* (4.19 auf Seite 87), d.h. mehreren Prozedurüberbrückungen von $l_1 : p_1(\dots), l_2 : p_2(\dots), \dots l_k : p_k(\dots)$ kann $R_{skip}^{l_1,p_1}$ sukzessive verfeinert werden zu $(s,s') \in R_{skip} := R_{skip}^{l_1,p_1 \dots l_k,p_k}$, falls $s = s'$ oder $\exists i : (Marken(s') = \{l_i\} \wedge s \in Post_{p_i}^*(s'))$. Für dieses R_{skip} sind die beiden Eigenschaften Gleichheit der Annotationsfunktion sowie Pfadzerlegbarkeit aus Definition 3.3.12 auf Seite 40 zu zeigen. Sei dazu ein $\phi \in CTL* - X$ beliebig aber fest gewählt.

zu zeigen: $\forall (s,s') \in R_{skip} : L_\phi(s) = L_\phi(s')$
Wähle $s \in P \times \Gamma^*$ und $s' \in P' \times \Gamma'^*$ beliebig aber fest.

1. Fall: $s = s'$
Dann gilt offensichtlich $L_\phi(s) = L_\phi(s')$.

2. Fall: $Marken(s') = \{l\} \wedge s \in Post_p^*(s')$
Nach (B.1) hat s die Form $(g, \gamma_1\gamma_2 \dots \gamma_r\gamma'_1\gamma'_2 \dots \gamma'_m)$ für $s' = (g', \gamma'_1\gamma'_2 \dots \gamma'_m) \in P' \times \Gamma'^*$. Wegen $\tau_{inv-X}(RENV_l^p, p) = true$ ist $\forall 1 \leq i \leq r : Marken(\gamma_i) \cap L0 = \emptyset$. D.h. weder eine die *Erreichbarkeit* beeinflussende Marke $l' \in L0$ findet statt, noch eine ϕ beeinflussende Marke (4.16 auf Seite 81) oder Prozedur (4.17 auf Seite 81) oder Variable (4.14 auf Seite 81),(4.15 auf Seite 81) bzw. globale oder lokale *Variablenzuweisung* (4.13 auf Seite 81). Nach (3.2 auf Seite 36) kommen in L_ϕ daher nur Annotationsausdrücke vor, welche unabhängig sind von $\gamma_1, \gamma_2, \dots$, sowie γ_r. Entsprechend haben sich auch nur nicht an ϕ beteiligte globale Variablen von g' nach g verändert. Daher ist auch in diesem Fall $L_\phi(s) = L_\phi(s')$.

Damit ist $\forall (s,s') \in R_{skip} : L_\phi(s) = L_\phi(s')$ gezeigt.

zu zeigen: Pfadzerlegbarkeit
Wähle einen beliebigen Pfad $\pi = s_1 \hookrightarrow s_2 \hookrightarrow \dots$ von $\gamma[\![S]\!]_\phi$. Wird die *Modelltransformation* (4.19 auf Seite 87) nicht angewendet, so ist $s'_i = s_i$ und der Pfad kann trivial zerlegt werden in $\forall i > 0 : \pi_i = s_i = \pi'_i$, so dass $\forall i \geq 0, s \in \pi_i, s' \in \pi'_i : (s,s') \in R_{skip}$, da stets $s = s'$. Sei o.B.d.A. auf π die *Modelltransformation* (4.19 auf Seite 87) angewendet an der Stelle s_k ($Marken(s_k) = \{l\}$ mit „$l : p(\dots);$“ $\in S$). Unter der Annahme $s'_{k+1} = s_{k+j+1}$ (Konfluenz) wird aus $\pi = s_1 \hookrightarrow s_2 \hookrightarrow \dots s_k \underbrace{\hookrightarrow s_{k+1} \hookrightarrow \dots \hookrightarrow s_{k+j}}_{Post_p^*(s_k)} \hookrightarrow s_{k+j+1} \hookrightarrow \dots$ dann der Pfad $\pi' = s'_1 \hookrightarrow s'_2 \hookrightarrow \dots s'_k \hookrightarrow s'_{k+1} \hookrightarrow \dots = s_1 \hookrightarrow s_2 \hookrightarrow \dots s_k \hookrightarrow s_{k+j+1} \hookrightarrow \dots$. Bis s_k wird trivial zerlegt $\pi_i = s_i$ für $(i < k)$. Die *Konfigurationen* $s_k, s_{k+1}, \dots s_{k+j}$ fallen gemäß R_{skip} (B.2) in die gleiche Äquivalenzklasse und werden in einen gemeinsamen Teilpfad zerlegt: $\pi_k = \hookrightarrow s_{k+1} \hookrightarrow \dots \hookrightarrow s_{k+j}$. Die weiteren *Konfigurationen* danach werden wiederum trivial zerlegt: $\forall i > 0 : \pi_{k+i} = s_{k+j+i} = s'_{k+i}$. Abbildung B.2 veranschaulicht diese Zerlegung. Dann gilt einerseits $\forall i < k : (s_i, s'_i) \in R_{skip}$, da $s_i = s'_i$, andererseits $\forall i \in \{k, k+1, \dots, k+j\} : (s_i, s'_k) \in R_{skip}$, da $s_i \in Post_p^*(s_k)$ und weiter ist $\forall i > k : (s_{i+j}, s'_i) \in R_{skip}$, da wegen $s'_{k+1} = s_{k+j+1}$ auch $s_{i+j} = s'_i$ mit $i > k$ ist.

noch zu zeigen: $s'_{k+1} = s_{k+j+1}$ (Konfluenz)
Dies ist eine Folgerung aus τ_{eff} und τ_{stop}. τ_{eff} garantiert per Definition 4.3.18 auf Seite 80, dass

sämtliche *Variablenbelegung*en nach Ablauf der Prozedur p erhalten bleiben, d.h. $env^{s_k} = env^{s_{k+j+1}}$. Wegen $s_k = s'_k$ und des in neuen Konfiruationenübergangs *skip*, welcher keine *Variablenbelegung*en ändert, ist dann $env^{s_k} = env^{s'_k} = env^{s'_{k+1}} = env^{s_{k+j+1}}$, d.h. insbesondere $s'_{k+1} = s_{k+j+1}$, falls die Prozedur p nicht versackend $\tau_{stop} = false$ ist. Denn sonst ist das „verlassen“ der Prozedur p nicht garantiert.

Damit stotter-simuliert das *PDS* $\gamma[\![S']\!]_\phi$ das *PDS* $\gamma[\![S]\!]_\phi$.

Die Umkehrung $\gamma[\![S]\!]_\phi$ stotter-simuliert $\gamma[\![S']\!]_\phi$ geht analog, so dass beide stotter-bisimular sind. Gemäß Betrachtungen aus Abschnitt 3.3.5 ab Seite 37 folgt damit die CTL*-X-Invarianz.

□

Satz 4.3.12 auf Seite 82
Exakte abstrakte Informationen τ_{eff}, τ_{stop} und τ_{inv-X} sind entscheidbar.

Beweis (Skizze)
Seien eine Prozedur p mit Parametern $Param_p = \{p_1, p_2, \ldots, p_n\}$ und *Variablenbelegung*en W gegeben. Im Folgenden werden Berechnungsvorschriften angeführt, um exakte abstrakte Informationen τ_{eff}, τ_{stop} und τ_{inv-X} zu berechnen.

τ_{eff}:

- $Post^*(I_W)$ berechnen für endliche Menge an *Variablenbelegung*en W (endlich, da das *SPDS* nur über endlich viele Variablen mit endlichem Typ verfügt). Die Berechnung kann *symbolisch* erfolgen, so wie es auch Aufgabe eines *Modellprüfung*swerkzeugs ist [220].
- extrahiere *Variablenbelegung*en von terminierten Köpfen (aufwändig aber geht, da endlich)[5],
- prüfe diese *Variablenbelegung*en auf Äquivalenz[6] mit W,

$\Rightarrow$ sind äquivalent gdw. p ist *Seiteneffekt* frei bzgl. W.

Denn p hat dann für jede der *Variablenbelegung*en aus W die Werte von globalen Variablen erhalten und diese wurden direkt mittels $Post^*$ korrekt berechnet.

τ_{stop}:

- für jede der **endlich** vielen *Variablenbelegung*en $env \in W$ prüfe *Wahrheitsgehalt* von $\phi = F\ l_2 \in$ LTL für „$l_1 :\ p(x_1, x_2, ..., x_n);\ l_2 :\ skip$;“ mit entsprechend gewählten Variablen x_i für die *Variablenbelegung* $env \in W$.

$\Rightarrow$ $\phi = F\ l_2$ ist falsch für min. **ein** $env \in W$ gdw. p versackend bzgl. W.

Denn $\phi = F\ l_2$ beschreibt, dass in jedem Lauf irgendwann die Marke l_2 erreicht werden muss. l_2 ist daher im Beispiel für die bestimme *Variablenbelegung* env nicht *erreichbar*, wenn der Lauf in p nicht wieder aus p austritt, d.h. versackt.

τ_{inv-X}:

- berechne analog zum ersten Schritt von τ_{eff} wieder die Menge $Post^*(I_W)$,
- bestimme die endliche Menge $L := Marken(Kopf(Post^*(I_W)))$ (aufwändig aber geht, da Menge der Köpfe endlich),

$\Rightarrow$ $L \cap L0 = \emptyset$ gdw. p invariant-X bzgl. W.

[5] kann z.B. als OBDD repräsentiert sein
[6] z.B. OBDD-Schnitt auf Leerheit testen

Denn die endliche Menge L beschreibt genau die Menge der mittels *env* $\in W$ *erreichbaren* Marken.

□

Satz 4.3.20 auf Seite 102
Die Variablen-Repräsentanten R_{var} des ILPs aus Definition 4.3.31 auf Seite 100 sind wohldefinierte *optimale* Variablen-Repräsentanten für ein gegebenes *SPDS* S.

Beweis (Skizze)
zu zeigen: Wohldefiniertheit
Sei hierzu $R = R_{fix} \cup R_{var}$. Dann ist nach Konstruktion des ILP $\forall l \in Marken(S) : \forall v \in Vars(S) : used_l^{read}(v) \Rightarrow \tau_l^{Eq}(v) \cap R \neq \emptyset$. Denn wenn eine an der Marke $l \in Marken(S)$ lesend verwendete Variable $v \in Vars(S)$ beliebig aber fest gewählt wird ($used_l^{read}(v) = true$), dann gilt:
1. Fall: $\tau_l^{Eq}(v) \cap \mathbb{N}' \neq \emptyset$
$\Rightarrow R_{const} \supseteq \tau_l^{Eq}(v) \cap \mathbb{N}' \neq \emptyset \Rightarrow \tau_l^{Eq}(v) \cap R \supseteq \tau_l^{Eq}(v) \cap R_{const} \neq \emptyset \Rightarrow \tau_l^{Eq}(v) \cap R \neq \emptyset$.
2. Fall: $\tau_l^{Eq}(v) \cap \mathbb{N}' = \emptyset \wedge \tau_l^{Eq}(v) \cap R_{fix} \neq \emptyset$
$\Rightarrow R_\phi \supseteq \tau_l^{Eq}(v) \Rightarrow \tau_l^{Eq}(v) \cap R_\phi \neq \emptyset \Rightarrow \tau_l^{Eq}(v) \cap R \supseteq \tau_l^{Eq}(v) \cap R_\phi \neq \emptyset \Rightarrow \tau_l^{Eq}(v) \cap R \neq \emptyset$.
3. Fall: $\tau_l^{Eq}(v) \cap \mathbb{N}' = \emptyset \wedge \tau_l^{Eq}(v) \cap R_{fix} = \emptyset$
$used_l^{read(v)} \Rightarrow l_v \in reads(S) \Rightarrow \sum_{w \in \tau_l^{Eq}(v) \cup Vars} \alpha_w \geq 1 \Rightarrow \exists w : \alpha_w = 1 \wedge w \in \tau_l^{Eq}(v) \cap Vars \Rightarrow w \in R_{var} \Rightarrow \tau_l^{Eq}(v) \cap R_{var} \neq \emptyset \Rightarrow \tau_l^{Eq} \cap R \neq \emptyset$ (wegen $R_{var} \subseteq R$).
Damit sind die durch R_{const} und das ILP gewählten Variablen R_{var} zusammen als $R = R_{const} \cup R_{var}$ gültige Repräsentanten.

zu zeigen: *Optimal*ität
Sei eine Lösung des ILP als Belegung der α_v und damit eine Lösung R_{var} gegeben, so dass folgende Nebenbedingungen erfüllt sind:

$$\forall L_x \in reads(S) : \sum_{v \in \tau_L^{Eq}(x) \cap Vars} \alpha_v \geq 1.$$

Angenommen, es gäbe Variablen-Repräsentanten R'_{var} mit Belegungen α'_v, so dass $|R'_{var}| < |R_{var}|$. Dann ist $\sum_{v \in Vars} \alpha_v < \sum_{v \in Vars} \alpha'_v$ und daher muss es eine lesende *Variablenverwendung* $L'_{x'} \in reads(S)$ geben, so dass die Nebenbedingung nicht erfüllt ist, denn $|R_{var}|$ ist minimal unter den Nebenbedingungen. D.h. es ist für dieses $L'_{x'}$:

$$\sum_{v \in \tau_{L'}^{Eq}(x') \cap Vars} \alpha_v < 1.$$

Da $L'_{x'}$ keinen Konstanten-Repräsentanten haben kann (andernfalls wäre $L'_{x'}$ nicht ins ILP aufgenommen), gibt es dann aber für $L'_{x'}$ keinen belegungsäquivalenten Repräsentanten. D.h. es ist $\tau_{L'}^{Eq}(x') \cap R'_{var} = \emptyset$ und damit $\tau_{L'}^{Eq}(x') \cap (R'_{var} \cup R_{fix}) = \emptyset$. Dann sind aber $R'_{var} \cup R_{fix}$ keine Repräsentanten für das *SPDS* S zum Widerspruch zur Annahme, dass R'_{var} Variablen-Repräsentanten wären.

□

Satz 4.3.25 auf Seite 108
Die Repräsentantenprägung T_R ist α-invariant für gegebenes $\phi \in \alpha$ und ein beliebiges $\alpha \in \{$ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X, ACTL, ACTL*, CTL, CTL*, LTL$\}$.

Beweis (Skizze)
Der Beweis ergibt sich durch Bisimulationsnachweis per Induktion über die Anzahl der Regelanwendungen für CTL*. Sei dazu $v \in Vars(S)$ eine *SPDS*-Variable eines *SPDS* mit einem von v verschiedenem Repräsentanten $v \neq \tau_l^R(v)$. Betrachtet sei jeweils der Induktionsschritt einer Regelanwendung der Modelltranformationsregeln 4.23 auf Seite 104 bzw. 4.24 auf Seite 104. Sei $\gamma[\![S]\!]_\phi$

jeweils das interpretierende *PDS* des *SPDS* S und $\gamma[\![S']\!]_\phi$ das interpretierende *PDS* des *SPDS* $T_R(S)$.
1. Fall: Seien die Voraussetzungen für die Modelltranformationsregel 4.23 auf Seite 104 erfüllt. D.h. es sei $l : x = e \in Z, v \in Vars, used_l^{read}(v)$ sowie $v \neq \tau_l^R(v)$. Nach Konstruktion von τ_l^R gilt somit: $v \doteq_{RENV_l^*} \tau_l^R(v)$. Wegen $RENV_l^* \supseteq RENV_l$ ist dann $\forall env \in RENV_l : [\![v]\!]_{env} = [\![\tau_l^R(v)]\!]_{env}$. Wird dann in der Modelltranformationsregel 4.23 auf Seite 104 jedes vorkommende v durch einen belegungsäquivalenten Repräsentanten $\tau_l^R(v)$ ersetzt, so ändert sich der Wert des Ausdrucks e nicht: $\forall env \in RENV_l : [\![e]\!]_{env} = [\![e[\tau_l^R(v)/v]]\!]$. Daher wird an der *SPDS*-Variable x der gleiche Wert zugewiesen und der Variablenwert von x bleibt unverändert. Das zu Grunde liegende *PDS* ist daher identisch und die Identität ist trivialerweise bisimular (es wird als Bisimulationsrelation $R_R = id$ die Identität gewählt). Eine *temporale Formel* kann damit nach Lemma 3.3.1 keinen Unterschied in einem Lauf feststellen.
2. Fall: Seien die Voraussetzungen für die Modelltranformationsregel 4.24 auf Seite 104 gegeben. D.h. „$l : s;$“ $\in Stats(S) \setminus Z, v \in Vars(S), used_l^{read}(v) = true$ sowie $v \neq \tau_l^R(v)$. Da „$l : s;$“ $\in Stats(S) \setminus Z$, ist s keine *Zuweisung*, so dass in s nur lesende *SPDS-Variablenverwendungen* auftreten können (und dies wegen $used_l^{read}(v) = true$ auch tun). Die SPDS-*Anweisung* s verfügt nun über möglicherweise mehrere *Ausdrücke* $e_1, e_2, e_k \in Expr$, welche analog zu Fall 1 behandelt werden können. Jeder dieser *Ausdrücke* e_i wird durch die Transformationsregel $l : s; \Rrightarrow l : s[\tau_l^R(v)/v];$ so verändert, dass auftretende *SPDS-Variablenverwendungen* von v ersetzt werden durch einen belegungsäquivalenten Repräsentanten $\tau_l^R(v)$. Analog zu Fall 1 kann damit gefolgert werden, dass gilt: $\forall e \in s \cap Expr : \forall env \in REANV_l : [\![e]\!] = [\![e[\tau_l^R(v)/v]]\!]$. Das zu Grunde liegende *PDS* $\gamma[\![S']\!]_\phi$ ist daher auch in diesem Fall identisch zu $\gamma[\![S]\!]_\phi$ und stellt mittels Bisimulationsrelation $R_R = id$ eine Bisimulation dar. Eine *temporale Formel* kann auch in diesem Fall damit nach Lemma 3.3.1 keinen Unterschied in einem Lauf feststellen.
Folglich ist die Repräsentantenprägung T_R demnach in beiden Fällen α-invariant für gegebenes $\phi \in \alpha$ und ein beliebiges α einer temporalen Sprache.

□

Satz 4.3.26 auf Seite 108
Das Eliminieren von *SPDS-Zuweisung*en der Form „$l : x = e;$“ mittels *Modelltransformation* T^*_{elim} ist α-invariant für gegebenes $\phi \in \alpha$ und ein beliebiges $\alpha \in$ {ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X, ACTL, ACTL*, CTL, CTL*, LTL} sowie *Erreichbarkeits-invariant* für a priori gegebene Marken $L0$.

Beweis (Skizze)
Im Fall einer versackungsfreien *SPDS-Zuweisung* $\tau_{stop}(RENV_l, „l : x = e“) = false$ (insbesondere auch $\tau_{stop}(RENV_l, „e“) = false$) folgt die Korrektheit nach Lemma 4.3.22 für die Transformationsregel 4.25 auf Seite 105. Danach ist eine solche *SPDS-Zuweisung* eliminierbar $elim_\zeta(„l : x = e;“) = true$, d.h. die Anwendung von T_{elim} unter diesen Vorbedingungen ändert das *Modellprüfung*sresultat nicht. Seien daher nun eine *SPDS-Zuweisung* „$l : x = e$“ an eine *redundante SPDS*-Variable x ($\tau_{rdz}(x) = true$) gegeben, welche nun aber möglicherweise versackt, d.h. $\tau_{stop}(RENV_l, „l : x = e“) = true$. In diesem Fall kommt die *Modelltransformationsregel* 4.26 auf Seite 105 zur Anwendung. Analog zum Beweis von Lemma 4.3.22 kann der Beweis via Bisimulation mit der Relation R_{elim} auf Läufen der zu Grunde liegenden *PDS* geführt werden. Auch hier sei wiederum o.B.d.A. der allgemeinere Fall $\zeta = \phi$ betrachtet. Es gibt nun allerdings den Unterschied, dass die mögliche Versackung des Laufs erhalten bleiben muss, da sich sonst die *Erreichbarkeit* im Modell ändert. Für den Nachweis werden daher die gleichen Läufe $z_0 \rightarrow_{\gamma[\![S]\!]_\phi} z_1 \cdots \rightarrow_{\gamma[\![S]\!]_\phi} z_n \rightarrow_{\gamma[\![S]\!]_\phi} \ldots$ der interpretierenden *PDS* $\gamma[\![S]\!]_\phi$ vom *SPDS* S und des *PDS* $\gamma[\![T_{elim}(S)]\!]_\phi$ vom *SPDS* $T_{elim}(S)$ betrachtet. Seien die *SPDS*-Variable x *redundant* $\tau_{rdz}(x) = true$ und $e \in Expr$ an der Marke $l \in Marken(S)$ ggf. versackend $\tau_{stop}(RENV_l, e) = false$, so dass auch die *SPDS-Zuweisung* „$l : x = e;$“ möglichweise versackend ist $\tau_{stop}(RENV_l, „l : x = e“) = true$. Da die Versackungs-Bedingung nur eine *konservative* Schätzung der Versackung ist, ist die Versackung von Läufen nicht garantiert. D.h. der Lauf $z_0 \rightarrow_{\gamma[\![S]\!]_\phi} z_1 \cdots \rightarrow_{\gamma[\![S]\!]_\phi} z_n$ terminiert möglicherweise an der *Konfiguration* z_n, da es bei *Auswertung* von e zu einer Division mit 0 oder bei der *Zuweisung* „$x = e$“ zu einem arithmetischen Überlauf wegen $[\![e]\!]_{env} \notin range(x)$ kommen kann.

Zunächst ist zu zeigen, dass in $\gamma[\![S]\!]_\phi$ für z_n die *Nachfolgekonfiguration* z erreicht wird (in Zeichen $z_n \rightarrow_{\gamma[\![S]\!]_\phi} z$), gdw. in $\gamma[\![T_{elim}(S)]\!]_\phi$ die *Nachfolgekonfiguration* $z|_x^c$ erreicht wird für ein $c = [\![x]\!]_{env^{z_n}}$ (in Zeichen $z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z|_x^c$, Voraussetzung für Bisimulation).

- 1. Fall: Terminiert der Lauf in $\gamma[\![S]\!]_\phi$ an der *Konfiguration* z_n, so wird dieser Lauf im Zuge der *Modellprüfung* mit *temporalen Formel*n (siehe Abschnitt 3.3.3 ab Seite 35) ergänzt zu dem unendlichen Lauf $z_0 \rightarrow_{\gamma[\![S]\!]_\phi} z_1 \ldots z_n \rightarrow_{\gamma[\![S]\!]_\phi} z_n \rightarrow_{\gamma[\![S]\!]_\phi} z_n \ldots$. Durch die Modelltransformationsregel 4.26 auf Seite 105 versackt auch der Lauf auch an der *Konfiguration* z_n in $\gamma[\![T_{elim}(S)]\!]_\phi$, wenn einerseits e nicht *auswertbar* ist, da e im Ausdruck $(e < 0)||(e > x_{max})$ vorkommt und dann ebenfalls nicht auswertbar ist. Oder wenn andererseits e zwar *auswertbar* in $\gamma[\![S]\!]_\phi$ bzw. $\gamma[\![T_{elim}(S)]\!]_\phi$ ist, aber der Ausdruck $[\![x]\!]_{env^{z_n}} \notin range(x)$ nicht geeignet für den Typ von x ist. In diesem Fall kommt es in $\gamma[\![S]\!]_\phi$ zu einem Überlauf und damit wird der Lauf wiederholt durch z_n unendlich fortgesetzt. Und in $\gamma[\![T_{elim}(S)]\!]_\phi$ ist die Bedingung $(e < 0)||(e > x_{max})$ erfüllt ($[\![(e < 0)||(e > x_{max})]\!]_{env^{z_n}} = 1$), so dass der bedingte Sprung aus Transformationsregel 4.26 auf Seite 105 zu einer Endlosschleife führt und auch damit der Lauf durch z_n unendlich fortgesetzt wird wie in $\gamma[\![S]\!]_\phi$. Der Umkehrschluss gilt ebenso, da e *auswertbar* ist gdw. $(e < 0)||(e > x_{max})$ auswertbar ist und $[\![e]\!]_{env^{z_n}} \in range(x)$ gdw. $[\![(e < 0)||(e > x_{max})]\!]_{env^{z_n}} = 0$. Zusammen gilt für eine *temporale Formel* für den erweiterten Lauf:

$$z_0 \rightarrow_{\gamma[\![S]\!]_\phi} z_1 \ldots z_n \rightarrow_{\gamma[\![S]\!]_\phi} z_n \rightarrow_{\gamma[\![S]\!]_\phi} z_n \ldots$$
$$\Leftrightarrow$$
$$z_0 \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z_1 \ldots z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z_n \ldots$$

  Wegen $(z_n, z_n) \in R_{elim}$ und nach Lemma 4.3.22 folgt nun Bisimulation in diesem Fall.

- 2. Fall: Sei nun im Folgenden angenommen, der Lauf terminiert nicht in $\gamma[\![S]\!]_\phi$, so dass gilt $z_n \rightarrow_{\gamma[\![S]\!]_\phi} z$. Dann kommt es bei der *Auswertung* von e nicht zur Versackung und es ist $[\![e]\!]_{env^{z_n}} \in range(x)$, was zu einer erfolgreichen *Zuweisung* des Werts von e an x führt. Da e erfolgreich ausgewertet werden kann, kann auch der Ausdruck $(e < 0)||(e > x_{max})$ in $\gamma[\![T_{elim}(S)]\!]_\phi$ erfolgreich ausgewertet werden. Dieser hat aber wegen $[\![e]\!]_{env^{z_n}} \in range(x)$ den Wert 0 ($false$), so dass der bedingte Sprung aus Transformationsregel 4.26 auf Seite 105 in $\gamma[\![T_{elim}(S)]\!]_\phi$ übergangen wird und mit der *Nachfolgekonfiguration* und unverändertem x statt geändertem x aus $\gamma[\![S]\!]_\phi$ fortgesetzt wird. D.h. es ist $z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z|_x^c$ für das $c = [\![e]\!]_{env^{z_n}} \in range(x)$ in $\gamma[\![T_{elim}(S)]\!]_\phi$.

  Sei nun die Umkehrung angenommen: in $\gamma[\![T_{elim}(S)]\!]_\phi$ gelte $z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z|_x^c$ für ein $c \in range(x)$. Dann ist nach vorigen Überlegungen wiederum e einerseits *auswertbar*, da auch der Ausdruck $(e < 0)||(e > x_{max})$ auswertbar ist in $\gamma[\![T_{elim}(S)]\!]_\phi$. Andererseits wurde aber auch der bedingte Sprung übergangen, so dass die Bedingung nicht erfüllt ist, d.h. es ist $[\![(e < 0)||(e > x_{max})]\!]_{env^{z_n}} = 0$. D.h. es sit $[\![e]\!]_{env^{z_n}} \in range(x)$. Entsprechend versackt auch die *SPDS-Zuweisung* „$l : x = e;$“ nicht in $\gamma[\![S]\!]_\phi$, so dass in $\gamma[\![S]\!]_\phi$ gilt $z_n \rightarrow_{\gamma[\![S]\!]_\phi} z$. Zusammen gilt:

$$z_0 \rightarrow_{\gamma[\![S]\!]_\phi} z_1 \ldots z_n \rightarrow_{\gamma[\![S]\!]_\phi} z \Leftrightarrow z_0 \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z_1 \ldots z_n \rightarrow_{\gamma[\![T_{elim}(S)]\!]_\phi} z|_x^c \tag{B.3}$$

  Wegen $(z, z|_x^c) \in R_{elim}$ und nach Lemma 4.3.22 folgt nun Bisimulation auch in diesem Fall.

Damit bisimuliert das *PDS* $\gamma[\![T_{elim}(S)]\!]_\phi$ das *PDS* $\gamma[\![S]\!]_\phi$, wonach T^*_{elim} schließlich α-invariant und *Erreichbarkeits-invariant* bezüglich $L0$ ist. (Betrachtungen aus Abschnitt 3.3.5 ab Seite 37, insb. Abbildung 3.6 auf Seite 41).

□

Satz 4.3.27 auf Seite 108
Das Eliminieren von *SPDS*-Variablen mittels Modellreduktion T_{omit} ist α-invariant für das *SPDS* $T^*_{elim}(T_R(S))$ und gegebenes $\phi \in \alpha$ sowie ein beliebiges $\alpha \in \{$ACTL-X, ACTL*-X, CTL-X, CTL*-X, LTL-X, ACTL, ACTL*, CTL, CTL*, LTL$\}$ sowie *Erreichbarkeits-invariant* für a priori gegebene

Marken $L0$.

Beweis (Skizze)
Betrachtet werden die drei *Modelltransformationen* 4.27, 4.28 sowie 4.29 auf Seite 105 für ein gegebenes $\phi \in CTL^*$. Um unnötige Fallunterscheidungen diesbezüglich zu vermeiden, sei vereinfachend o.B.d.A. im Folgenden $v_i = v$ angenommen. Wiederum wird die Behauptung per Bisimulation gezeigt. Gelte dazu $\tau_{rdz}(v)$ (insbesondere auch $\tau_{rdz}(v_i)$). In allen dieser drei Fälle kommt dann die *SPDS*-Variable v nicht mehr schreibend (in einer *SPDS-Zuweisung*) im *SPDS* $T^*_{elim}(T_R(S))$ vor gemäß Satz 4.3.26 auf Seite 108, da jede *Zuweisung* eliminiert wurde. Informal kann daher jede lesende *Verwendung* von v durch die Konstante $0 \in range(v)$ ersetzt werden ohne das *Modellprüfung*sresultat zu beeinträchtigen.
Für den Nachweis werden die Läufe des interpretierenden *PDS* $\gamma[\![S'']\!]_\phi$ vom *SPDS* $S'' = T^*_{elim}(T_R(S))$ und des *PDS* $\gamma[\![S']\!]_\phi$ vom *SPDS* $S' = T_{omit}(T^*_{elim}(T_R(S)))$ betrachtet. Ich werde zeigen, dass sich das *Modellprüfung*sresultat wegen Bisimulation der Läufe nicht ändert. Zwei *Konfigurationen* $(g, \gamma_1\gamma_2 \dots \gamma_m) \in P \times \Gamma^*$ des *PDS* $\gamma[\![S'']\!]_\phi$ und $(g', \gamma'_1\gamma'_2 \dots \gamma'_m) \in P' \times \Gamma^{*'}$ des *PDS* $\gamma[\![S']\!]_\phi$ seien dann bisimular (dies überträgt sich automatisch auf Läufe), falls sie sich lediglich darin unterscheiden, dass in $\gamma[\![S']\!]_\phi$ keine Variable v existiert und damit auch keine Belegung ($\bot$). Je nachdem, ob v eine lokale oder globale Variable ist, betrifft dies g' bzw. ein γ'_i. Als Bisimulationsrelation wird daher

$$R_{omit} := \{(z, z|^\bot_v) \mid z \in (P \times \Gamma^*)\} \subseteq (P \times \Gamma^*) \times (P' \times \Gamma^{*'}).$$

verwendet. In allen drei Fällen ist $v \notin R_{fix}$, so dass auch $v \notin \phi$ ist und damit v aus dem Modell entfernt werden kann, ohne dass die Formel ϕ ungültig werden würde (auch wenn v *redundant* ist, könnte v in ϕ vor kommen).

- 1. Fall: $v \in Vgbl \setminus R_{fix}$
  Analog zu Satz 4.3.26 sei ein beliebiger Lauf im *PDS* $\gamma[\![S'']\!]_\phi$ gegeben: $z_0 \to_{\gamma[\![S'']\!]_\phi} z_1 \dots z_n \to_{\gamma[\![S'']\!]_\phi} z_n \to_{\gamma[\![S'']\!]_\phi} z_n$..., so dass gilt $[\![v]\!]_{env^{z_{n+1}}} \neq \bot \wedge \forall i \leq n : [\![v]\!]_{env^{z_i}} = \bot$. Die Variable v komme daher an der *Konfiguration* z_n zu einer gültigen *Variablenbelegung* (z.B. Prozedureintritt oder Anfangskonfiguration). Es ist zu zeigen, dass gilt: $(z_n \to_{\gamma[\![S'']\!]_\phi} z_{n+1}) \Leftrightarrow (z_n \to_{\gamma[\![S']\!]_\phi} z_{n+1}|^\bot_v)$. Dies ist aber klar, da $z_n \to_{\gamma[\![S'']\!]_\phi} z_{n+1}$ gdw. $z_n \to_{\gamma[\![S'']\!]_\phi} z_{n+1}|^\bot_v$ gdw. $z_n \to_{\gamma[\![S']\!]_\phi} z_{n+1}|^\bot_v$ und dabei auftretende lesende *Verwendungen* wegen $S \Rrightarrow S[0/v]$ durch den Wert 0 ersetzt werden. Per Induktion für den weiteren Verlauf des betrachteten Laufs folgt dann die Bisimulation dieses Laufs in $\gamma[\![S'']\!]_\phi$ durch $\gamma[\![S']\!]_\phi$ mit ansonsten gleicher *Variablenbelegung* und den gleichen Marken. Wegen Betrachtung eines beliebigen Laufs, gilt dies damit auch für alle Läufe.

- 2. Fall: $v \in Vlcl_p \setminus R_{fix}$ für eine Prozedur $p \in Prz(S) = Prz(T^*_{elim}(T_R(S)))$
  Dieser Fall ist analog zu Fall 1.

- 3. Fall: $v_i \in Param_p \setminus R_{fix}$ für eine Prozedur $p \in Prz(S) = Prz(T^*_{elim}(T_R(S)))$
  Hier ist nun $v = v_i$. Zusätzlich zu den Betrachtungen der vorigen beiden Fälle werden sämtliche *Prozeduraufruf*e geändert. Die Behauptung ergibt sich dann analog zu Fall 1 und gemäß bisherigen Betrachtungen für S_5 beim Eliminieren von Parametervariablen bei gleichzeitiger Änderung der *Prozeduraufruf*e.

Damit stellt das *PDS* $\gamma[\![S']\!]_\phi$ eine Bisimulation vom *PDS* $\gamma[\![S'']\!]_\phi$ dar für das $\phi \in CTL^*$. Analog zu vorigen Sätzen kann damit wieder gefolgert werden, dass T_{omit} schließlich α-invariant und *Erreichbarkeits-invariant* bezüglich $L0$ ist (Betrachtungen aus Abschnitt 3.3.5 ab Seite 37, insb. Abbildung 3.6 auf Seite 41).

□

Satz 4.3.29 auf Seite 117
Die *Wertebereichsreduktion* T_{min} ist CTL*-invariant für gegebenes $\phi \in$ CTL* und somit auch α-invariant für gegebenes $\phi \in \alpha$ und ein beliebiges $\alpha \in$ {ACTL, ACTL-X, ACTL*, ACTL*-X, CTL,

CTL-X, CTL*, CTL*-X, LTL, LTL-X} sowie *Erreichbarkeits-invariant* für a priori gegebene Marken $L0$.

Beweis (Skizze)
Zum Nachweis genügt wiederum zu zeigen, dass T_{min} CTL*-invariant ist für gegebenes $\phi \in$ CTL*. Der Rest ergibt sich wie bisher. Dazu wird wiederum Bisimulation gezeigt. Sei dazu $\gamma[\![S]\!]_\phi = (P, \Gamma, \hookrightarrow, I, L_\phi)$ das von S beschriebene *PDS* und $\gamma[\![S']\!]_\phi = (P', \Gamma', \hookrightarrow', I', L'_\phi)$ das von $S' = T_{min}(S)$ beschriebene *PDS* bei einer Regelanwendung der *Modelltransformation* 4.31 auf Seite 116 (Induktionsschritt). Dazu seien die entsprechenden Vorbedingungen $v = (\alpha, b) \in V, b_{min,v} < b$ sowie $\alpha \notin \phi$ erfüllt. Die Bisimulationsrelation besteht nicht wie vielleicht erwartet aus der Identität bezüglich der *Variablenbelegung*en. Danach würde sich lediglich die Repräsentation der *Variablenbelegung*en in der Länge der Dualzahlendarstellung wegen unterschiedlicher Typen unterscheiden. Als Bisimulationsrelation wird

$$R_{min} := \{(s, s|_v^{v'} \in (P \times \Gamma^*) \times (P' \times \Gamma^{*'}) \mid v' = \min(\tau_{max}(v), [\![v]\!]_{env^s})\}$$

verwendet, welche für $\gamma[\![S']\!]_\phi$ zu große Variablenwerte auf maximal mögliche abbildet. Für die Beweisrichtung $\gamma[\![S']\!]_\phi$ simuliert $\gamma[\![S]\!]_\phi$ werde dann ein beliebiger Lauf $\pi = s_0 \hookrightarrow s_1 \ldots \hookrightarrow s_k \hookrightarrow s_{k+1} \ldots$ von $\gamma[\![S]\!]_\phi$ betrachtet. Dieser wird so zu einem Lauf von $\gamma[\![S']\!]_\phi$ abgeändert, so dass sämtliche Annotationsausdrücke bezüglich ϕ erhalten bleiben. An der *Konfiguration* s_k sei erstmalig v in die *Variablenbelegung* env^{s_k} eingeführt. Gemäß Abschnitt 3.2.1 ab Seite 24 erfolgt diese Einführung mit uninitialisierten Variablenwerten (ob lokale Variablen bei Prozedureintritt oder globale Variablen für die *Anfangskonfiguration* $s_0 = s_k$). Im *PDS* $\gamma[\![S']\!]_\phi$ kann der Lauf π dann ggf. nicht fortgesetzt werden, wenn der Typ b von v in $\gamma[\![S]\!]_\phi$ größer ist ($b > b_{min,v}$) als der Typ $b_{min,v}$ von v in $\gamma[\![S']\!]_\phi$. Dies ist der Fall, wenn $c = [\![v]\!]_{env^{s_k}} > \tau_{max}(v)$. Dies sei im Folgenden o.B.d.A. angenommen, da andernfalls $(s_k, s_k|_v^{v'}) \in R_{min}$ mit $v' = min(\tau_{max}(v), [\![v]\!]_{env^{s_k}} = c$ der Lauf ohne Änderung $\pi' = \pi$ simuliert wird. Der Lauf π wird dann für den Fall $c > \tau_{max}(v)$ zu π' mit $v' = \tau_{max}(v)$ abgeändert. D.h. zu große Variablenwerte erhalten den maximal möglichen Wert und es gilt $(s_k, s_k|_v^{v'}) \in R_{min}$ mit $v' = c$. Dabei muss sicher gestellt sein, dass dies keinen Einfluss auf das Restverhalten des Modells sowie keinen Einfluss auf eine *temporale Formel* bzw. deren genutzte Annotationsausdrücke hat. Auswirkungen auf *Variablenbelegung*en im *PDS* $\gamma[\![S]\!]_\phi$ kann es dadurch aber nicht geben, da v uninitialisiert nicht lesend verwendet wird, denn sonst wäre nicht $\tau_{max} < c$ und damit $b_{min,v} < b$ (siehe Definition 4.3.38 auf Seite 115). Entsprechend gilt dies auch für die *Folgekonfigurationen* $s_{k+1}, s_{k+2}, \ldots$ bis zur möglicherweise ersten schreibenden *Verwendung* von v. D.h. es ist $(s_{k+j}, s_{k+j}|_v^{v'}) \in R_{min}$ mit $v' = min(\tau_{max}(v), [\![v]\!]_{env^{s_k}}$. Eine schreibenden *Verwendung* von v muss aber auftreten wenn der Lauf nicht zwischenzeitlich terminiert, da v andernfalls durch die Variablenredundanzelimination eliminiert worden wäre und $v \notin \phi$ ist. Sei daher die erste schreibende *Verwendung* von v am *Konfigurationenübergang* $s_{k+j} \hookrightarrow s_{k+j+1}$ (sofern der Lauf nicht terminiert). Hier nun erhält v einen Variablenwert $c' \leq \tau_{max}(v)$ (wegen *Konservativität* von τ_{max}) und somit reicht der Typ $b_{min,v}$ für v in $\gamma[\![S']\!]_\phi$ aus, so dass $v' = \min\{\tau_{max}(v), c'\} = c'$ für die *Konfiguration* s_{k+j+1} ist. Auswirkungen auf Annotationsausdrücke (bzw. ϕ) sind damit ebenso ausgeschlossen, da diese Laufsänderung zudem auch nicht die Versackung des Laufs beeinflusst und $v \notin \phi$ ist. Per Induktion wird damit der gesamte Lauf π von $\gamma[\![S]\!]_\phi$ durch den Lauf ϕ' in $\gamma[\![S']\!]_\phi$ simuliert. Da die Umkehrung der Simulation trivialerweise erfüllt ist (jeder Lauf von $\gamma[\![S']\!]_\phi$ ist automatisch auch Lauf von $\gamma[\![S]\!]_\phi$), sind beide *PDS* bisimular. Hieraus folgt nun die Invarianz bezüglich der CTL*-Formel ϕ und damit wie bisher auch die anderen Behauptungen.

□

Literaturverzeichnis

[1] E. M. Clarke, Fujita, P. Rajan, Reps S. Shankar. *Program Slicing of Hardware Description Languages.* L. Pierre and T. Kropf (Eds.): CHARME'99, In Conference on Correct Hardware Design and Verification Methods, Lecture Notes In Computer Science (LNCS) 1703:298–313, Springer-Verlag Berlin Heidelberg, citeseer.ist.psu.edu/article/clarke99program.html, 2004.

[2] Marcelo d'Amorim, Steven Lauterburg, Darko Marinov. *Delta execution for efficient state-space exploration of object-oriented programs.* International Symposium on Software Testing and Analysis, Proceedings of the 2007 international symposium on Software testing and analysis, 2007. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4528965.

[3] A. Bouajjani, J-C. Fernandez, N. Halbwachs. Minimal model generation. In *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science (LNCS)*, pages 197–203. Springer-Verlag Berlin Heidelberg, 1991. http://www.springerlink.com/content/5774431278n151t4/.

[4] A. Jefferson Offutt, Zhenyi Jin, Jie Pan. The dynamic domain reduction procedure for test data generation. In *Software—Practice and Experience*, volume 29(2), pages 167–193. John Wiley and Sons, Inc. New York, 1999. http://portal.acm.org/citation.cfm?id=309087.309101.

[5] Akash Lal, Thomas Reps. Improving pushdown system model checking. In *Proceedings of the 18th International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science (LNCS)*, pages 343–357. Springer-Verlag Berlin Heidelberg, 2006. http://www.springerlink.com/content/dw14017l12t31621/.

[6] Aleksandr Zaks, Ilya Shlyakhter, Franjo Ivancic, Srihari Cadambi, Malay Ganai, Aarti Gupta and Pranav Ashar. Using range analysis for software verification. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006)*. Computing Research Repository (CoRR), 2006. http://www.cs.nyu.edu/~zaks/TCAD.pdf.

[7] Alex Groce, Gerard Holzmann, Rajeev Joshi, Ru-gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Sixth International Workshop on Constraints in Formal Verification Grenoble, France, June 26, 2009*, Lecture Notes in Computer Science, pages 0–15. Springer-Verlag Berlin Heidelberg, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.7934.

[8] Alexander S. Kulikov. Automated generation of simplification rules for sat and max-sat. In *Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science (LNCS)*, pages 430–436. Springer-Verlag Berlin Heidelberg, 2005. http://www.springerlink.com/content/phmdm1g0xn09l1lu/.

[9] Alfons Kemper, Guido Moerkotte, Michael Steinbrunn. Optimizing boolean expressions in object-bases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 79–90, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers:Principles, Techniques, and Tools: International Edition, 2nd Edition.* Addison-Wesley, 2007.

[11] Amir M. Ben-Amram. Size-change termination with difference constraints. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 30(3). ACM Press New York, 2008. http://portal.acm.org/citation.cfm?id=1353445.1353450.

[12] Amir M. Ben-Amram, Chin Soon Lee. Program termination analysis in polynomial time. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 29(1). ACM Press New York, 2007. http://portal.acm.org/citation.cfm?id=1180480.

[13] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language, PhD thesis.* DIKU, University of Copenhagen, Copenhagen, 1994.

[14] Anna-Lena Lamprecht, Tiziana Margaria, Berhard Steffen. Data-flow analysis as model checking within the jabc. In *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science (LNCS)*, pages 101–104. Springer-Verlag Berlin Heidelberg, 2006. http://www.springerlink.com/content/r02p1h6w17875786/.

[15] Antonin Kucera1, Jan Strejcek. *The Stuttering Principle Revisited: On the Expressiveness of Nested X and U Operators in the Logic LTL.* In Proceedings of Computer Science Logic: 16th International Workshop, CSL 2002, 11th Conference of the EACSL, Edinburgh, Scotland, UK, September 22 - 25, 2002.

[16] Arie Gurfinkel, Marsha Chechik. *Why Waste a Perfectly Good Abstraction?* Department of Computer Science, University of Toronto, ON M5S 3G4, Canada, In H.Hermanns and J.Palsberg (Eds.): TACAS 2006, Lecture Notes in Computer Science (LNCS) 3920:212–226, Springer-Verlag Berlin Heidelberg, 2006.

[17] Armin Biere. *Effiziente Modellpruefung des Mu-Kalkuels mit binaeren Entscheidungsdiagrammen.* PhD Thesis, Institut fuer Informatik, Universitaet Karlsruhe, http://fmv.jku.at/mucke, 1997.

[18] Aubrey Jaffer. *JACAL - Symbolic Mathematics System Version 1c2.* 2010. http://people.csail.mit.edu/jaffer/JACAL.html.

[19] Ayal Zwi Pinkus, Serge Winitzki, Jitse Niesen. *Using Yacas, function reference.* 2007. http://yacas.sourceforge.net/homepage.html.

[20] B. Buchberger, R. Loos. Algebraic simplification. In *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–43. Springer-Verlag, Vienna - New York, 1982. http://www.risc.jku.at/publications/download/risc2680/1982-00-00-B.pdf.

[21] Bageshri Karkare, Uday P. Khedker. An improved bound for call strings based interprocedural analysis of bit vector frameworks. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 29(6). ACM Press New York, 2007. http://portal.acm.org/citation.cfm?id=1286829.

[22] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL 79, pages 29–41, New York, NY, USA, 1979. ACM.

[23] J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core ltl model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[24] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

[25] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[26] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[27] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Probdivine-mc: Multi-core ltl model checker for probabilistic systems. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 77–78, Washington, DC, USA, 2008. IEEE Computer Society.

[28] J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *To appear in Journal of Logic and Computation*, 2009.

[29] J. Barnat, V. Forejt, M. Leucker, and M. Weber. DivSPIN – A SPIN compatible distributed model checker. In M. Leucker and J. van de Pol, editors, *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation*, pages 95–100, July 2005.

[30] J. Barnat and P. Ročkai. Shared Hash Tables in Parallel Model Checking. *ENTCS*, 198(1):79–91, 2008.

[31] Jiří Barnat, Luboš Brim, and Martin Leucker. Parallel Model Checking and the FMICS-jETI Platform. In *The Twelfth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*. IEEE Computer Society Press, 2007.

[32] Jiří Barnat and Pavel Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

[33] Jiří Barnat and Ivana Černá. Distributed Breadth-First Search LTL Model Checking. *Special Issue on Parallel and Distributed Databases of Formal Methods in System Design*, 29(2):117–134(18), September 2006.

[34] Bernhard Steffen. *Generating Data Flow Analysis Algorithms from Modal Specifications*. Science of Computer Programming, 21(2):115–139, citeseer.ist.psu.edu/steffen93generating.html, 1993.

[35] Bernhard Steffen, Anna-Lena-Lamprecht, and Tiziana Margaria. *Data-Flow Analysis as Model Checking within the jABC*. Lecture Notes in Computer Science, 3923:101–104, Springer-Verlag Berlin Heidelberg, 2006.

[36] Bernhard Steffen, David Schmidt. Program analysis as model checking of abstract interpretations. In *Proceedings of the 5th International Symposium on Static Analysis*, volume 1503 of *Lecture Notes In Computer Science (LNCS)*, pages 351–380. Springer-Verlag Berlin Heidelberg, 1998. http://portal.acm.org/citation.cfm?id=760066.

[37] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM New York, USA, 1999. http://doi.acm.org/10.1145/309847.309942.

[38] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Not.*, 24:275–284, June 1989.

[39] P. Briggs, R. Shillingsburg, and L. Simpson. *Dead code elimination*. MSCP, Rice University, operational definition [Cytron et al.] citeseer.ist.psu.edu/briggs93dead.html, 1994.

[40] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115 – 131, 1988.

[41] Bruno Dutertre, Leonardo de Moura. *The YICES SMT Solver*. Computer Science Laboratory, SRI International, USA, 2006.

[42] Bwolen Yang. *Optimizing Model Checking Based on BDD Characterization*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.9022, http://portal.acm.org/citation.cfm?id=929216.

[43] C. Tinelli. *A DPLL-based calculus for ground satisfiability modulo theories*. Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy), Giovambattista Ianni and Sergio Flesca, Lecture Notes in Artificial Intelligence 2424, Springer-Verlag Berlin Heidelberg citeseer.ist.psu.edu/tinelli02dpllbased.html, 2002.

[44] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12:10:1–10:38, December 2008.

[45] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000. 10.1023/A:1019225027893.

[46] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[47] Chao Wang, Zijiang Yang, Aarti Gupta, Franjo Ivancic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 352–365. Springer-Verlag Berlin Heidelberg, 2007. http://www.springerlink.com/content/83835rn353287342/.

[48] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In Mark Aagaard and John O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 33–51. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36126-X3.

[49] Chris Hankin. Program analysis tools. In *International Journal on Software Tools for Technology Transfer (STTT), Special section on program analysis tools*, volume 2(1), pages 6–12. Springer-Verlag Berlin Heidelberg, 1998. http://www.springerlink.com/content/ac840ym9lhtqj9dy/.

[50] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, Björn Lisper. Faster wcet flow analysis by program slicing. In *Proceedings of the ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 103–112. ACM Press New York, 2006. http://portal.acm.org/citation.cfm?id=1134650.1134666.

[51] Christian Fecht, Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science (LNCS)*, pages 90–104. Springer-Verlag Berlin Heidelberg, 1998. citeseer.ist.psu.edu/fecht98propagating.html.

[52] Christian Razafimahefa. *A study of side-effect analyses for Java*. Master's thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1999.

[53] Chritoph Scholl. *Mehrstufige Logiksynthese unter Ausnutzung funktionaler Eigenschaften*. Dissertation an der Universität des Saarlandes, 1996. http://deposit.ddb.de/cgi-bin/dokserv?idn=97233811X.

[54] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. Int'l Conf. on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[55] Corina S. Pasareanu, Matthew B. Dwyer, Willem Visser. *Finding Feasible Counter-examples when Model Checking Abstracted Java Programs*. Lecture Notes in Computer Science, Band 2031, citeseer.ist.psu.edu/areanu01finding.html, 2001.

[56] W. Zimmermann D. Richter. Variablenelimination für symbolische modelle. In *4. Workshop Modellbasiertes Testen im Rahmen der 39. Jahrestagung der Gsellschaft für Informatik*, Lecture Notes in Informatics (LNI). Köllen-Verlag, 2009.

[57] Dan Grove, Linda Torczon. A study of jump function implementation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 90–99. ACM Press New York, 1993. http://portal.acm.org/citation.cfm?id=155090.155099.

[58] Daniel Polansky. *Verifying Properties of Infinite-state Systems - Masters Thesis*. Masarykova univerzita, Fakulta informatiky, 2000.

[59] David Callahan, Keith D. Cooper, Ken Kennedy, Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 152–161. ACM Press New York, 1986. http://portal.acm.org/citation.cfm?id=12276.13327.

[60] David Eng. *Combining Static and Dynamic Data in Code Visualization.* Sable Research Group, McGill University, PASTE 2002, Charleston, South Carolina, 2002.

[61] David L. Dill, Andreas J. Drexler, Alan J. Hu, C. Han Yang. *Protocol Verification as a Hardware Design Aid.* IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, 522–525, 1992.

[62] J.W. Davidson and A.M. Hollersnm. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18:89–102, 1992.

[63] Dejvuth Suwimonteerabuth. *Reachability in Pushdown Systems: Algorithms and Applications.* Lehrstuhl für Informatik VII der Technischen Universität München, 2009. http://www.model.in.tum.de/um/bibdb/suwimont/phdthesis.pdf.

[64] Dejvuth Suwimonteerabuth, Stefan Berger, Stefan Schwoon, Javier Esparza. jmoped: A test environment for java programs. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 164–167. Springer-Verlag Berlin Heidelberg, 2007. http://www.springerlink.com/content/k504r7pg6240730m/.

[65] Dejvuth Suwimonteerabuth, Stefan Schwoon, Javier Esparza. jmoped: A java bytecode checker based on moped. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science (LNCS)*, pages 541–545. Springer-Verlag Berlin Heidelberg, 2005. http://www.springerlink.com/content/32p4x035k3rll5nh/.

[66] Dimitra Giannakopoulou, Klaus Havelund. Automata-based verification of temporal properties on running programs. *International Conference on Automated Software Engineering*, 0:412, 2001.

[67] Dirk Richter. *Modellreduktionstechniken für symbolische Kellersysteme*, pages 144–153. Proc. of the 25th Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008. http://www.informatik.uni-kiel.de/uploads/tx_publication/tr_0811.pdf.

[68] Dirk Richter. Rekursionspräzise intervallanalysen. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl, 2009. http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1726.

[69] Dirk Richter. *Äquivalenzanalysen - exakt oder nicht - im Vergleich*, pages 27–36. Proc. of the 26th Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2009. http://www.informatik.uni-kiel.de/uploads/tx_publication/tr_0915.pdf.

[70] Dirk Richter, Raimund Kirner, Wolf Zimmermann. On undecidability results of real programming languages. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl, 2009. http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1726.

[71] Dirk Richter, Roberto Hoffmann. Spezifikationsgetriebene abstraktion für kellersysteme. In *16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Schloss Raesfeld, 2011. www.drgames.de/dl/kps11.pdf.

[72] Doron Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*, pages 17–28. Springer-Verlag Berlin Heidelberg, 1998. http://www.springerlink.com/content/wj352800m7534l03/.

[73] Doron Peled, T. Wilke. *Stutter-invarient temporal properties are expressible without the next-time operator*. Information Processing Letters, 63(5):243–246, 1997.

[74] Evelyn Duesterwald, Rajiv Gupta, and Mary Soffa. Reducing the cost of data flow analysis by congruence partitioning. In Peter Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 357–373. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-57877-3 24.

[75] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 177–187, Washington, DC, USA, 2001. IEEE Computer Society.

[76] E. Allen Emerson, Thomas Wahl, Nicolas Halbwachs, Lenore D. Zuck. *Dynamic Symmetry Reduction*. Department of Computer Sciences and Computer Engineering Research Center, The University of Texas, Austin/TX 78712, ETATS-UNIS, TACAS 2005 : tools and algorithms for the construction and analysis of systems, International conference on tools and algorithms for the construction and analysis of systems No11, Edinburgh, ROYAUME-UNI (04/04/2005), 3440:382–396, 2005.

[77] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-guided abstraction refinement*. In Proccedings of CAV, Lecture Notes in Computer Science (LNCS) 1855:154–169, Springer-Verlag Berlin Heidelberg, 2000.

[78] E. M. Clarke, O. Grumberg, M. Minea, D. Peled. State space reduction using partial order techniques. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 2(3), pages 279–287. Springer-Verlag Berlin Heidelberg, 1998. http://www.springerlink.com/content/kh6b21udvgutjjvh/.

[79] ECMA International. *ISO/IEC 23271:2006 - Common Language Infrastructure (CLI) Partitions I to V*. ECMA-335,www.ecma-international.org/publications/standards/Ecma-335.htm, www.microsoft.com/net, 2001.

[80] Edmund Clarke, Daniel Kroening, Flavio Lerda. *A Tool for Checking ANSI-C Programs*. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Lecture Notes in Computer Science 2988:168–176, Springer-Verlag Berlin Heidelberg, www.cs.cmu.edu/~modelcheck/cbmc, 2004.

[81] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. *Progress on the State Explosion Problem in Model Checking*. Lecture Notes in Computer Science (LNCS) 2000/2001, 176–177, citeseer.ist.psu.edu/clarke00progress.html, ISSN 1611-3349, 2001.

[82] Eric W. Allender, Michael C. Loui, Kenneth W. Regan. *Complexity Theory*. State University of New York at Buffalo, 2004. http://ftp.cs.rutgers.edu/pub/allender/ALR02.pdf.

[83] F. Ivancic, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, Chao Wang, Zijiang Yang. Model checking c programs using f-soft. In *ICCD '05: Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 297–308. IEEE Computer Society, 2005. http://portal.acm.org/citation.cfm?id=1097556.

[84] F. L. Bauer, H. Ehler, A. Horsch, B. Moeller, H. Partsch, O. Paukner, P. Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*. Lecture Notes in Computer Science (LNCS). Springer-Verlag Berlin Heidelberg, 1987.

[85] Flavio Lerda, James Kapinski, Edmund M. Clarke, Bruce H. Krogh. Verification of supervisory control software using state proximity and merging. In *Hybrid Systems: Computation and Control*, volume 4981 of *Lecture Notes in Computer Science*, pages 344–357. Springer-Verlag Berlin Heidelberg, 2008. http://www.springerlink.com/content/0156214014342k73/.

[86] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of program analysis*. Korrigierte 2. Auslage, Springer-Verlag Berlin Heidelberg New York, 2005.

[87] Florian Martin. *Generating Program Analyzers*. Dissertation der Technischen Fakultaet der Universitaet des Saarlandes, Saarbruecken, 1999.

[88] Frank Tip. A survey of program slicing techniques. In *Journal of Programming Languages*, volume 3, pages 121–189, 1995. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3782.

[89] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[90] G. Ramalinga. *The undecidability of aliasing*. ACM Press New York 16(5):1467–1471, NY, USA, IBM T. J. Watson Research Center, Yorktown Heights, 1994.

[91] G. Ramalingam. The undecidability of aliasing. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16(5), pages 1467–1471. ACM Press New York, 1994. http://portal.acm.org/citation.cfm?id=186025.186041.

[92] G. Ramalingam. *Context sensitive synchronization sensitive analysis is undecidable*. ACM Trans. on Programming Languages and Systems, 22:416-430, 2000.

[93] G. Yorsh, Thomas Reps, Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science (LNCS)*, pages 530–545. Springer-Verlag Berlin Heidelberg, 2004. http://www.springerlink.com/content/924r1k6l9p58342a/.

[94] Gerald J. Holzmann, Dragan Bosnacki. The design of a multicore extension of the spin model checker. In *IEEE Transactions on Software Engineering*, volume 33(10), pages 659–674. IEEE Press Piscataway, NJ, USA, 2007. http://portal.acm.org/citation.cfm?id=1314033.1314051.

[95] Gerard J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997. http://spinroot.com/spin/Workshops/ws97/papers.html.

[96] Gerard J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science (LNCS)*, pages 232–244. Springer-Verlag Berlin Heidelberg, 1999. http://www.springerlink.com/content/26blbaunf3j47pb5/.

[97] Amit Goel and Randal Bryant. Symbolic simulation, model checking and abstraction with partially ordered boolean functional vectors. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–325. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27813-9 20.

[98] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 442–452, New York, NY, USA, 1988. ACM.

[99] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63166-6 10.

[100] GrammaTech Inc. *CodeSurfer Path Inspector*. http://www.grammatech.com/ products/codesurfer, 2006.

[101] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16:843–871, May 1994.

[102] Guillaume Brat, Klaus Havelund, Seungjoon Park, Willem Visser. Java pathfinder - second generation of a java model checker. In *Proceedings of the Workshop on Advances in Verification*. NASA Ames Research Center, USA, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.5740.

[103] Jia Guo, María Garzarán, and David Padua. The power of belady's algorithm in register allocation for long basic blocks. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 374–389. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24644-2 24.

[104] Gustavo Quiros Araya. *Static Byte-Code Analysis for State Space Reduction*. Computer Science and Natural Sciences of RWTH University, Aachen, Germany, 2006. http://wwwhome.cs.utwente.nl/~michaelw/nips/sarn-thesis.pdf.

[105] H. Peng, Y. Mokhtari, S. Tahar. Model reduction based on value dependency. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, pages 220–224. IEEE Computer Society Washington, DC, USA, 2001. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=954701.

[106] Hans H?ttel. *Rice's Theorem*. Aalborg University, 2006.

[107] Hao Chen, David Wagner. *MOPS: an infrastructure for examining security properties of software*. In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), 235—244, Washington DC, /www.cs.ucdavis.edu/~hchen/mops, 2002.

[108] Hassen Saidi. *Model Checking Guided Abstraction and Analysis*. Lecture Notes In Computer Science, Band 1824, Proceedings of the 7th International Symposium on Static Analysis table of contents, 377–396, 2001.

[109] Helmut Seidl, Christian Fecht. *Interprocedural Analyses: A Comparison*. FB IV Informatik, Universitat Trier, Universitat des Saarlandes, Saarbrucken, Elsevier Science Publishers B.V., 1999.

[110] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams*. Lecture Notes In Computer Science (LNCS), 1999.

[111] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[112] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21:848–894, July 1999.

[113] Hiroshi Watanabe, Koki Nishizawa, Osamu Takaki. *A Coalgebraic Representation of Reduction by Cone of Influence*. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS), Electronic Notes in Theoretical Computer Science, 164(1):177–194, 2006.

[114] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[115] J. C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 9(1), pages 51–93. ACM Press New York, 2000. http://portal.acm.org/citation.cfm?doid=332740.332741.

[116] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, H. Zheng. *A Formal Study of Slicing for Multi-threaded Programms with JVM Concurrency Primitives*. In Proccedings of the 1999 International Symposium on Static Analysis, 1999.

[117] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., ISBN 0-201-31008-2, citeseer.ist.psu.edu/gosling00java.html, 2000.

[118] Jan Obdrz?lek. *Formal Verification of Sequential Systems with Infinitely Many States*. Master's Thesis, Faculty of Informatics, Masaryk University, http://homepages.inf.ed.ac.uk/s0128832/publications.html, 2001.

[119] Jan Obdrzlek. *Model Checking Java Using Pushdown Systems*. LFCS, Division of Informatics, The University of Edinburgh, 2002.

[120] Javier Esparza, Antonin Kucera, Stefan Schwoon. Model-checking ltl with regular valuations for pushdown systems. In *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science (LNCS)*, pages 316–339. Springer-Verlag Berlin Heidelberg, 2001. http://www.springerlink.com/content/l4lm9b4jvc35r80x/.

[121] Javier Esparza, Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Foundations of Software Science and Computation Structures*, volume 1578 of *Lecture Notes in Computer Science (LNCS)*, pages 14–30. Springer-Verlag Berlin Heidelberg, 1999. http://www.springerlink.com/content/w7jxa283dv4pd9bx/.

[122] Javier Esparza, Stefan Schwoon. *A BDD-based model checker for recursive programs*. Lecture Notes in Computer Science, 2102:324–336, Springer-Verlag Berlin Heidelberg, 2001.

[123] Javier Esparza, Stefan Schwoon. *An Automata-theoretic Approach to Software Model-Checking*. Final Report on EPSRC Grant R93346/02, Institute of Formal Methods in Computer Science, University of Stuttgart, 2005.

[124] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasarenu, Robby, H. Zheng. Bandera: Extracting finite–state models from java source code. In *Proceedings of the International Conference on Software Engineering*, pages 439–448. ACM Press New York, 2000. http://portal.acm.org/citation.cfm?id=337234.

[125] Jens Knoop, Oliver Rüthing. Constant propagation on predicated code. *Journal of Universal Computer Science*, 9(8):829–850, 2003. http://www.jucs.org/jucs_9_8/constant_propagation_on_predicated.

[126] Jens Krinke. Effects of context on program slicing. In *Journal of Systems and Software, 4th Worshop on source code analysis and manipulation (SCAM)*, volume 79(9), pages 1249–1260. Elsevier Science Publishers New York, USA, 2006. http://portal.acm.org/citation.cfm?id=1167771.1167775.

[127] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science (LICS)*, volume 98(2) of *Information and Computation*, pages 142–170. ACM Press New York, 1992. http://portal.acm.org/citation.cfm?id=162046.

[128] K. McMillan. *Symbolic Moldel Checking*. Kluwer Academic Publishers, 1993.

[129] K. Yorav, O. Grumberg. Static analysis for state-space reductions preserving temporal logics. In *Formal Methods in System Design*, volume 25(1), pages 67–96. Springer-Verlag Netherlands, 2004. http://www.springerlink.com/content/u146647t72v74545/te.

[130] Yonit Kesten and Amir Pnueli. Control and data abstraction: the cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:328–342, 2000. 10.1007/s100090050040.

[131] Klaus Havelund and Thomas Pressburger. *Model Checking Java Programs Using Java PathFinder*. International Journal of Software Tools for Technology Transfer (STTT), 2(4):366–381, citeseer.ist.psu.edu/havelund98model.html, 2000.

[132] Emanuel Kolb, Ondřej Šerý, and Roland Weiss. Applicability of the blast model checker: An industrial case study. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 218–229. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11486-1 19.

[133] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 389–392, New York, NY, USA, 2007. ACM.

[134] F. J. Kurdahi and A. C. Parker. Real: a program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pages 210–215, New York, NY, USA, 1987. ACM.

[135] Kwang Ting Cheng, A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th international conference on Design automation*, pages 86–91. ACM Press New York, 1993. http://portal.acm.org/citation.cfm?id=157485.164585.

[136] L. I. Millett, T. Teitelbaum. Slicing promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, 1998. http://netlib3.cs.utk.edu/spin/ws98/program.html.

[137] L. I. Millett, T. Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 2(4), pages 343–349. Springer-Verlag Berlin Heidelberg, 2000. http://www.springerlink.com/content/weq55e5taf082pek/.

[138] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[139] Anatole Le, Ondrej Lhotak, and Laurie Hendren. Using inter-procedural side-effect information in jit optimizations. In Rastislav Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 287–304. Springer Berlin, Heidelberg, 2005. 10.1007/11406921 22.

[140] Li Xin, Ogawa Mizuhito. Interprocedural program analysis for java based on weighted pushdown model checking. In *5th International Workshop on Automated Verification of Infinite-State Systems*, Electronic Notes on Theoretical Computer Science (ENTCS). Elsevier Science Publishers Ltd. Essex, UK, 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.1284.

[141] D. Liang, M. Pennings, and M.J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. 2008.

[142] Donglin Liang and Mary Jean Harrold. Equivalence analysis: a general technique to improve the efficiency of data-flow analyses in the presence of pointers. *SIGSOFT Softw. Eng. Notes*, 24:39–46, September 1999.

[143] M. Christodorescu, S. Jha. *Static Analysis of Executables to Detect Malicious Patterns*. In Proceedings of the 12th USENIX Security Symposium, 169–186, citeseer.ist.psu.edu/christodorescu03static.html, 2003.

[144] M. Weiser, Peter Henderson, Jim Lyle, Glenn Pearson, Joan Shertz, Randall H. Trigg. *Program slicing*. IEEE Transactions on Software Engineering, SE-10(4):352-357, San Diego, California, United States, 1984.

[145] Andrew Makhorin. *GNU Linear Programming Kit, Reference Manual for GLPK Version 4.45*. Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia, 51 Franklin St, Fifth Floor, Boston, MA, 02110-1301, USA, 2010.

[146] Malay Ganai, Aarti Gupta. *SAT-based scalable formal verification solutions*. Springer-Verlag Berlin Heidelberg, 2007. http://books.google.de/books?id=y0CiQNzG2yYC.

[147] Malay K. Ganai, Aarti Gupta, Franjo Ivancic, Vineet Kahlon,Weihong Li, Nadia Papakonstantinou, Sriram Sankaranarayanan, Chao Wang. Towards precise and scalable verification of embedded software. In *Design and Verification Conference (DVCon)*. San Jose, CA, 2008. http://www.nec-labs.com/~malay/pubs.htm.

[148] Federica Mandreoli, Riccardo Martoglia, and Enrico Ronchetti. Supporting temporal slicing in xml databases. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 295–312. Springer Berlin / Heidelberg, 2006. 10.1007/11687238 20.

[149] Maple Team. *Maple User Manual*. Copyright (C) Maplesoft, a division of Waterloo Maple Inc., 2010. http://www.maplesoft.com/.

[150] Margus Veanes, Nikolaj Bjorner, Yuri Gurevich, Wolfram Schulte. Symbolic bounded model checking of abstract state machines. In *International Journal of Software Informatics*, volume 3, pages 149–170. Institute of Software, Chinese Academy of Sciences, 2009. http://research.microsoft.com/apps/pubs/?id=79531.

[151] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu. State space reduction based on live variables analysis. In *6th International Symposium on Static Analysis*, volume 1694 of *Lecture Notes in Computer Science (LNCS)*, page 848. Springer-Verlag Berlin Heidelberg, 1999. http://books.google.de/books?id=nZtNAvediHsC&printsec=frontcover.

[152] Matthew B. Dwyer, John Hatcliff. *Slicing Software for Model Construction*. In Partial Evaluation and Semantic-Based Program Manipulation, 105–118, Kansas State University, 1999.

[153] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, Todd Wallentine. *Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs*. In the Proceedings of the Twefith International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS, 2005.

[154] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187. IEEE Computer Society Washington, DC, USA, 2001. http://portal.acm.org/citation.cfm?id=381493.

[155] Matthew B. Dwyer, Robby, John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267 – 276. ACM New York, 2003. http://portal.acm.org/citation.cfm?id=940107&dl=.

[156] Matthew B. Dwyer, Robby, John Hatcliff. Bogor: A flexible framework for creating software model checkers. In *Proceedings of the Testing: Academic And Industrial Conference on Practice And Research Techniques (TAIC-PART)*, pages 3–22. IEEE Computer Society Washington, 2006. http://portal.acm.org/citation.cfm?id=1158759.

[157] Maxim Mozgovoy. *Algorithms, Languages, Automata, and Compilers: A Practical Approach.* Infinity Science Press, 2009.

[158] Maxima Team. *Maxima Manual, Ver. 5.22.* 2010. http://maxima.sourceforge.net.

[159] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8:443–444, July 1965.

[160] Kenneth McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36577-X2.

[161] Melvin Fitting. Kleene's three valued logics and their children. *Journal of Fundamenta Informaticae*, 20(1-3):113–131, 1994. http://portal.acm.org/citation.cfm?id=183533.

[162] M.H. Schulz. Advanced automatic test pattern generation and redundancyidentification techniques. In *Eighteenth International Symposium on Fault-Tolerant Computing (FTCS)*, pages 30–35. IEEE Computer Society Washington, 1988. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5293.

[163] Miael I. Schwatzbach. *Lecture Notes on Static Analysis.* BRICS, Department of Computer Science, University of Aarhus, Denmark, 2008. http://www.itu.dk/people/brabrand/static.pdf.

[164] Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* In Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis: Theory and Applications, 189–234, Prentice-Hall Inc., Englewood Cliffs, New Jersy, 1981.

[165] Michael Codish, Vitaly Lagoon, Peter Schachte, Peter J. Stuckey. Size-change termination analysis in k-bits. In *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science (LNCS)*, pages 230–245. Springer-Verlag Berlin Heidelberg, 2006. http://www.springerlink.com/content/q68h84351n349456/.

[166] Michael Kaplan. *Computeralgebra.* Springer, Berlin, 2008. http://www.amazon.de/Computeralgebra-Michael-Kaplan/dp/3540213791.

[167] Microsoft Corporation. *Zing – User Manual.* Microsoft Research, 2004.

[168] Microsoft Corporation. *Zing – Language Specification.* Microsoft Research, 2005.

[169] Mooly Sagiv, Thomas Reps, Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Theoretical Computer Science*, volume 167(1-2), pages 131–170. Elsevier Science Publishers Ltd. Essex, UK, 1996. http://portal.acm.org/citation.cfm?id=243754.243762.

[170] Mukul R. Prasad, Armin Biere,Aarti Gupta. A survey of recent advances in sat-based formal verification. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 7(2), pages 156–173. Springer-Verlag Berlin Heidelberg, 2005. http://www.springerlink.com/content/ucyjdvmqaquetapx/.

[171] Jan Mühlberg and Gerald Lüttgen. Blasting linux code. In Luboš Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 211–226. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-70952-7 14.

[172] Kedar Namjoshi and Robert Kurshan. Syntactic program transformations for automatic abstraction. In E. Emerson and Aravinda Sistla, editors, *Computer Aided Verification*, volume 1855 of *LNCS*, pages 435–449. Springer Berlin / Heidelberg, 2000. 10.1007/1072216733.

[173] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, Kenneth L. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science (LNCS)*, pages 254–268. Springer-Verlag Berlin Heidelberg, 2005. http://www.springerlink.com/content/7rvpxyvunnne23a9/.

[174] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. http://www.amazon.de/Computational-Complexity-Perspective-Oded-Goldreich/dp/052188473X.

[175] P. Cousot, R. Cousot. *Temporal abstract interpretation*. Conference Record of the Twentyseventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 12–25, ACM Press, New York, citeseer.ist.psu.edu/context/1309676/0, 2000.

[176] P. Prinetto, M. Rebaudengo, M. Sonza Reorda,. An automatic test pattern generator for large sequential circuitsbased on genetic algorithms. In *Proceedings of the International Test Conference*, pages 240–249. IEEE Computer Society Washington, 1994. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=527955.

[177] Pankaj Jalote, Vipindeep Vangala, Taranbir Singh, Prateek Jain. Program partitioning: a framework for combining static and dynamic analysis. In *Proceedings of the 2006 international workshop on Dynamic systems analysis, SESSION: Testing and debugging*, pages 11–16. ACM Press New York, 2006. http://portal.acm.org/citation.cfm?id=1138912.1138916.

[178] Patrick Cousot. *Program analysis: the abstract interpretation perspective*. In ACM Workshop on Strategic Directions in Computing Research, MIT Lab. for Comput. Sci., Boston, Electronically available abstract in ACM Comput. Surv. 28A, 4, http://citeseer.ist.psu.edu/cousot96program.html, 1996.

[179] Patrick Cousot, Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM Press New York, 1978. http://portal.acm.org/citation.cfm?id=512770.

[180] Patrick Cousot, Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science (LNCS)*, pages 269–295. Springer-Verlag Berlin Heidelberg, 1992. http://www.springerlink.com/content/p869x76457527706/.

[181] Paul Molitor, Christoph Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. B.G. Teubner Verlag Stuttgart, Leipzig, 1999. http://books.google.de/books?id=aRXI_tfbFCsC&lpg=PP1&dq=inauthor%3A%22Paul%20Molitor%22&pg=PP1#v=onepage&q&f=false.

[182] Pedro de la C?mara, Mar?a del Mar Gallardo, Pedro Merino. *Abstract Matching for Software Model Checking*. in A. Valmari (Ed.): SPIN 2006, Lecture Notes In Computer Science (LNCS) 3925:182–200, Springer-Verlag Berlin Heidelberg, 2006.

[183] Radek Pelánek. Fighting state space explosion: Review and evaluation. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03240-0 7.

[184] Wojciech Penczek, Maciej Szreter, Rob Gerth, and Ruurd Kuiper. Improving partial order reductions for universal branching time properties. *Fundam. Inf.*, 43:245–267, August 2000.

[185] Phil McMinn. Search-based software test data generation: a survey: Research articles. In *Software Testing, Verification and Reliability*, volume 14(2), pages 105–156. John Wiley and Sons Ltd. Chichester, UK, 2004. http://portal.acm.org/citation.cfm?id=1077276.1077279.

[186] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, September 1999.

[187] H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 188–197, Washington, DC, USA, 2008. IEEE Computer Society.

[188] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In *Proceedings of the 6th international conference on Integrated formal methods*, IFM'07, pages 518–537, Berlin, Heidelberg, 2007. Springer-Verlag.

[189] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Formal Methods in System Design*, volume 18(2), pages 97–116. Springer-Verlag Netherlands, 2001. http://www.springerlink.com/content/xuju1h34k4632106/.

[190] Radek Pelanek. *On-the-fly State Space Reductions*. Technical Report, Faculty of Informatics Masaryk University Brno, 2005. www.fi.muni.cz/veda/reports.

[191] Radu Iosif. *Symmetry Reductions for Model Checking of Concurrent Dynamic Software*. Software Tools for Technology Transfer (STTT), 6(4):302–319, Springer-Verlag Berlin Heidelberg, 2004.

[192] Radu Iosif, Marius Bozga, Claudio DeMartini, Claudio Demartini, Matthew B. Dwyer, John Hatcliff, Yassine Laknech, Riccardo Sisto. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 254–255. IEEE Computer Society Washington, 2001. http://portal.acm.org/citation.cfm?id=872023.872566.

[193] Radu Rugina. Quantitative shape analysis. In *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science (LNCS)*, pages 228–245. Springer-Verlag Berlin Heidelberg, 2004. http://www.springerlink.com/content/kwgqwarqylye9528/.

[194] Radu Rugina, Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 27(2), pages 185–235. ACM Press New York, 2005. http://portal.acm.org/citation.cfm?id=1057388.

[195] Ralf Hartmut Güting, Martin Erwig. *Übersetzerbau*. Spinger Verlag Berlin Heidelberg, 1999.

[196] Rance Cleaveland, Tan Li, Steve Sims. *The Concurrency Workbench of the New Century, Version 1.2*. Department of Computer Science, North Carolina State University, SUNY at Stony Brook, NY 11794–4400, 2000.

[197] Ranjit Jhala, Rupak Majumdar. *Path Slicing*. PLDI, ACM 1-59593-080-9/05/0006, 2005.

[198] Rastislav Bodik, Rajiv Gupta, Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 321–333. ACM Press New York, 2000. http://portal.acm.org/citation.cfm?id=349342.

[199] Reinhard Wilhelm, Shmuel Sagiv, Thomas W. Reps. *Shape Analysis*, volume 1781 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer-Verlag Berlin Heidelberg, 2000. http://www.springerlink.com/content/bb4uca5e8xcuhcb3/.

[200] Richard M. Karp. Reducibility among combinatorial problems. In *Symposium on the Complexity of Computer Computations (Raymond E. Miller, James W. Thatcher)*, pages 85–103. Plenum Press, 1972. http://books.google.de/books?hl=de&lr=&id=bUJcweiYfkC&oi=fnd&pg=PA219&ots=0wAVOgvoJY&sig=jwjhEOlqu26sRV5MizirBpes9v8.

[201] Richard Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. Dissertation am Lehrstuhl fuer Theoretische Informatik und Grundlagen der KI, Institut fuer Informatik der Technischen Universitaet Muenchen, 1998.

[202] Robby, Edwin Rodriguez, Matthew B. Dwyer, John Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes In Computer Science (LNCS)*, pages 404–420. Springer-Verlag Berlin Heidelberg, 2004. http://www.springerlink.com/content/unkx5bhgyb0ybr1e/.

[203] Robby, Edwin Rodriguez, Matthew B. Dwyer, John Hatcliff. *Checking JML specifications using an extensible software model checking framework*. International Journal on Software Tools for Technology 8(6):280–299, Springer-Verlag Berlin Heidelberg, 2006.

[204] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, Tiziano Villa. *VIS: A System for Verification and Synthesis*. Proceedings of the 8th International Conference on Computer Aided Verification, Lecture Notes In Computer Science 1102:428–432, http://embedded.eecs.berkeley.edu/research/vis//whatis.html, 1996.

[205] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, Helmut Veith. *Modular Verification of Software Components in C*. Transactions on Software Engineering (TSE), 30(6):388–402, IEEE, http://www.cs.cmu.edu/~chaki/magic/, 2004.

[206] Sanjai Rayadurgam, Mats P. E. Heimdahl. Coverage based test-case generation using model checkers. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:0083, 2001.

[207] Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Program analysis using symbolic ranges. In Hanne Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 366–383. Springer Berlin / Heidelberg, 2007.

[208] Sarfraz Khurshid, Corina S. Pasareanu, Willem Visser. Generalized symbolic execution for model checking and testing. In H. Garavel, J. Hatcliff, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science (LNCS)*, pages 553–568. Springer-Verlag Berlin Heidelberg, 2003. http://www.springerlink.com/content/pllk4effd3vrec7l/.

[209] Bastian Schlich and Stefan Kowalewski. Model checking c source code for embedded systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:187–202, 2009. 10.1007/s10009-009-0106-5.

[210] Ethan L. Schreiber. *A CUDD Tutorial*. The University of California, Los Angeles, 2008.

[211] Sebastian Weber. *Defense in Depth – Vista Security: Abwehr von unerwuenschtem Code.* iX 12/2006 Magazin fuer professionelle Informationstechnik, Heise Zeitschriften Verlag GmbH, 2006.

[212] Seth C Goldstein, Seth Copen Goldstein, Klaus E schauser, David Culler. Enabling primitives for compiling parallel languages, 1995.

[213] Shaz Qadeer, Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science (LNCS)*, page 93.107. Springer-Verlag Berlin Heidelberg, 2005.

[214] Shuo Sheng,Michael Hsiao. Efficient preimage computation using a novel success-driven atpg. In *Proceedings of the conference on Design, Automation and Test in Europe*, volume 1, pages 822–827. IEEE Computer Society Washington, 2003. http://portal.acm.org/citation.cfm?id=1022826.

[215] Henny Sipma, Tomas Uribe, and Zohar Manna. Deductive model checking. In Rajeev Alur and Thomas Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61474-5 70.

[216] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[217] Stefan Edelkamp, StefanLeue, Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 6(4), pages 277–301. Springer-Verlag Berlin Heidelberg, 2004. http://www.springerlink.com/content/vk3bneaubjt2g8cg/.

[218] Stefan Kiefer. *Abstraction Refinement for Pushdown Systems.* Universitaet Stuttgart, Institut fuer Formale Methoden der Informatik, Abteilung Sichere und Zuverlaessige Softwaresysteme, Fakultaet Informatik, Elektrotechnik und Informationstechnik, Universitaetsstr. 38, 70569 Stuttgart, 2005.

[219] Dejvuth Suwimonteerabuth Stefan Kiefer, Stefan Schwoon. *Introduction to Remopla.* Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.

[220] Stefan Schwoon. *Model-Checking Pushdown Systems.* Dissertation am Lehrstuhl fuer Informatik VII der Technischen Universitaet Muenchen, 2002.

[221] Steven S. Muchnick. *Advanced compiler design and implementation.* San Francisco, Calif.: Morgan Kaufmann Publishers, 1997.

[222] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122 – 138, 2005. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004).

[223] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In Klaus Havelund and Rupak Majumdar, editors, *Proceedings of SPIN 2008*, volume 5156 of *Lecture Notes in Computer Science*, pages 270–287, Los Angeles, USA, August 2008. Springer.

[224] T. Lev-Ami, M. Sagiv. *TVLA: A framework for Kleene based static analysis.* In Static Analysis Symposium. Springer-Verlag Berlin Heidelberg, 2000.

[225] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, 4:21–36, January 1982.

[226] Tao Xie, Darko Marinov, Wolfram Schulte, David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science (LNCS)*, pages 365–381. Springer-Verlag Berlin Heidelberg, 2005. http://www.springerlink.com/content/4d3vv767nvtmh9hk/.

[227] Tayssir Touili. Dealing with communication for dynamic multithreaded recursive programs. In *Proceedings of the NATOR Advanced Research Workshop on Verification of infinite-state systems with applications to security (VISSAS)*, volume 1 of *Information and Communication Security*, pages 213–228. IOS Press, 2006. http://books.google.de/books?id=CQhT3X6TiCcC&source=gbs_navlinks_s.

[228] Thi Viet Nga Nguyen , Francois Irigoin. Interprocedural program analyses for efficient array bound checking. In *2nd International Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*. ACM Press New York, 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.1279.

[229] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Gr?goire Sutre. *Software Verifcation with BLAST*. EECS Department, University of California, Berkeley, LaBRI, Universitae de Bordeaux, France, 2003.

[230] Thomas Ball, Orna Kupferman,Mooly Sagiv. Leaping loops in the presence of abstraction. In *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 491–503. Springer-Verlag Berlin Heidelberg, 2007. http://www.springerlink.com/content/a38103r167434358/.

[231] Thomas Ball, Rupak Majumdar, Todd Millstein, Sriram K. Rajamani . Automatic predicate abstraction of c programs. In *ACM SIGPLAN Notices*, volume 36(5), pages 203–213. ACM Press New York, 2001. http://portal.acm.org/citation.cfm?id=378846.

[232] Thomas Ball, Sriram K. Rajamani. *Checking Temporal Properties of Software with Boolean Programs*. Proceedings of the Workshop on Advances in Verification, citeseer.ist.psu.edu/article/ball00checking.html, 2000.

[233] Thomas Firley, Ursula Goltz. *Property Preserving Abstraction for Software Verification – A Case Study*. Technische Universitaet Braunschweig, 2004.

[234] Thomas Gawlitza, Jan Reineke, Helmut Seidl, Reinhard Wilhelm. *Polynomial precise interval analysis revisited*. Technical Report, TU München, Germany, 2006. http://rw4.cs.uni-sb.de/publications.shtml.

[235] Thomas Reps. Undecidability of context-sensitive data-independence analysis. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 22(1), pages 162–186. ACM Press New York, 2000. http://portal.acm.org/citation.cfm?doid=345099.345137.

[236] Thomas Reps, Stefan Schwoon, Somesh Jha, David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Science of Computer Programming*, volume 58(1-2) of *Lecture Notes in Computer Science (LNCS)*, pages 206–263. Elsevier North-Holland, Inc. Amsterdam, The Netherlands, 2005. http://portal.acm.org/citation.cfm?id=1115650.

[237] Tom Ball, Sriram Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science (LNCS)*, pages 113–130. Springer-Verlag Berlin Heidelberg, 2000. http://www.springerlink.com/content/h437381h7n31308u/.

[238] Tom Ball, Sriram Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.2739.

[239] Tomas Matousek, Filip Zavoral. *Extracting Zing Models from C Source Code.* Lecture Notes in Computer Science (LNCS) 4362:900–910, Springer-Verlag Berlin Heidelberg, 2007.

[240] Tony Andrews, Shaz Qadeer, Siriam K. Rajamani, Jakob Rehof, Yichen Xie. *Zing: A Model Checker for Concurrent Software.* Technical Report MSR-TR-2004-10, Microsoft Research Redmond, 2004.

[241] Ulrich Stern, David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science (LNCS)*, pages 206–224. Springer-Verlag Berlin Heidelberg, 1995. http://www.springerlink.com/content/y0g21g4373602135/.

[242] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

[243] Viktor Kuncak, Patrick Lam, Martin Rinard . Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32. ACM Press New York, 2002. http://portal.acm.org/citation.cfm?id=503272.503276.

[244] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[245] Igor Walukiewicz. Model checking ctl properties of pushdown systems. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138. Springer Berlin Heidelberg, 2000. 10.1007/3-540-44450-5 10.

[246] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking uml statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, pages 435–446, London, UK, UK, 2002. Springer-Verlag.

[247] R. Wilhelm and D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung.* Spinger Verlag Berlin Heidelberg. Springer, 1997.

[248] Willem Visser, Corina S. Pasareanu, Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM Press New York, 2004. http://portal.acm.org/citation.cfm?id=1007526.

[249] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, Flavio Lerda. Model checking programs. In *Automated Software Engineering*, volume 10, pages 203–232. Springer-Verlag Netherlands, 2003. http://www.springerlink.com/content/v36q54n726773812/.

[250] William Visser, SeungJoon Park, John Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the third workshop on Formal methods in software practice (FMSP)*, pages 3–182. ACM Press New York, 2000. http://doi.acm.org/10.1145/349360.351125.

[251] Wojciech Penczek, Maciej Szreter, Rob Gerth, Ruurd Kuiper. Improving partial order reductions for universal branching time properties. *Journal of Fundamenta Informaticae - Special issue on Concurrency specification and programming (CSP), Issue 1-4*, 43:1–23, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.7132.

[252] Wolf Zimmermann, Michael Schaarschmidt. *Automatic Checking of Component Protocols in Component-Based Systems.* 5th International Symposium, SC 2006 Vienna, Austria, Lecture Notes In Computer Science (LNCS) 4089, 2006.

[253] Wolfram Team. *Wolfram Mathematica 8 Documentation Center*. 2010. http://www.wolfram.com/mathematica/.

[254] Bwolen Yang, Reid Simmons, Randal Bryant, and David O'Hallaron. Optimizing symbolic model checking for constraint-rich models. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 681–681. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48683-6 29.

[255] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24:393–423, November 2006.

[256] Y.N. Srikant, Pitri Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, 2nd Edition*. CRC Press Florida, 2008. http://books.google.de/books?id=1kqAv-uDEPEC.

[257] Z. Yang, C. Wang, A. Gupta. Model checking sequential software programs via mixed symbolic analysis. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 14(1). ACM Press New York, 2009. http://portal.acm.org/citation.cfm?id=1455239.

[258] Zaher S. Andraus, Mark H. Liffiton, Karem A. Sakallah. *CEGAR-Based Formal Hardware Verification: A Case Study*. Technical Report, Department of Electrical Engineering and Computer Science, University of Michigan, 2007. http://en.scientificcommons.org/43406181.

[259] Zijiang Yang, Chao Wang, Aarti Gupta, Franjo Ivancic. *Model Checking Sequential Software Programs via Mixed Symbolic Analysis*. ACM Transactions on Design Automation of Electronic Systems, 2009.

[260] Zijiang Yang, Karem Sakallah. Smt-based symbolic model checking for multi-threaded programs. In *Exploiting Concurrency Efficiently and Correctly Workshop*, 2008. http://www.cs.wmich.edu/~zijiang/publication.html.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Schlagwortverzeichnis