HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

IM Fakultät
Informatik und Medien

Abschlussarbeit
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Informatik
der Fakultät Informatik und Medien

Erfüllbarkeit und Optimierung von Pressenplanungsproblemen mit Constraint-Systemen

Julian Bruder
Leipzig, den 25. September 2024

Erstgutachter: Prof. Dr. Johannes Waldmann
Zweitgutachter: Dr. Sebastian Möbius

Die Pressenplanung von Holzbalken aus Bauteilen erfordert eine fein abgestimmte Konfiguration, um die Pressen optimal auszulasten und den Verschnitt der Balken zu minimieren. Dafür hat die Germanedge Solutions GmbH[1], Industriepartner dieser Arbeit, bereits eine unternehmensinterne Heuristik, welche aufgrund ihrer Natur allerdings möglicherweise (optimale) Lösungen nicht berücksichtigt. Im Rahmen dieser Arbeit wird daher eine prototypische Lösung mithilfe der Methoden der Constraint-Programmierung, speziell jener der Satisfiability Modulo Theories (SMT), erarbeitet, welche optimale Lösungen finden soll. Dazu werden die Constraints der Domäne zunächst mithilfe der Prädikatenlogik modelliert und anschließend in einer konkreten Repräsentation des Modells implementiert.

Dabei wird ein Programm entwickelt, welches aus den textuellen Eingaben der Pressen- und Bauteilspezifikationen ein Constraint-System erzeugt, das anschließend von einem externen Constraint-Solver gelöst wird. Das Programm wandelt die Lösung des Solvers in eine, zu den bereits existenten Schnittstellen passende, textuelle Ausgabe um. Weiter ist das Programm so strukturiert, dass verschiedene Logiken aus SMT zur Erzeugung des Constraint-Systems genutzt werden können, um die performanteste Kodierung des Problems in jenen Logiken einfach zu ermitteln. Dies wird durch Wahl der Programmiersprache Haskell und deren ausdrucksstarkes Typsystem ermöglicht. Als Grundlage zur Kodierung des Pressenplanungsproblems und Interaktion mit externen Solvern wird die Haskell-Bibliothek `Hasmtlib` [Bru24a] verwendet und an geeigneten Stellen erweitert.

Das Ziel für die prototypische Lösung des Problems der Pressenplanung ist die korrekte, optimale und bestenfalls schnelle (Wunsch Germanedge: unter sechs Minuten) Lösung bekannter Testfälle. Da es sich hier um eine prototypische Lösung handelt, werden einige Sonderfälle des Pressenplanungsproblems in dieser Arbeit nicht betrachtet. Am Ende wird die Performance der prototypischen Lösung mit jener der aktuell existenten heuristischen verglichen.

[1] https://www.germanedge.com/, abgerufen am 21.08.2024

Inhaltsverzeichnis

1. Einleitung

Die Germanedge Solutions GmbH spezialisiert sich in der Digitalisierung der Produktion und unterstützt Unternehmen mit individuellen Softwarelösungen für die Industrie 4.0 [VAB18]. Darunter auch ein nicht genannter Kunde, welcher Germanedge mit der Optimierung der Planung von Holzpressen beauftragt hat. Dabei werden Holzbauteile zu Holzbalken verleimt und anschließend in Pressen gepresst. Die Art und Weise der Anordnung der Bauteile in den Pressen wird als Pressenplan bezeichnet. Abbildung 1.1 visualisiert einen solchen Pressenplan. Die Presse (grau) presst in Richtung der Pfeile nach unten. Die Balken (gelb) liegen horizontal und bestehen aus Bauteilen, welche durch die vertikalen (schwarzen) Linien getrennt sind. Der Rest jedes Balkens ist rot hervorgehoben.

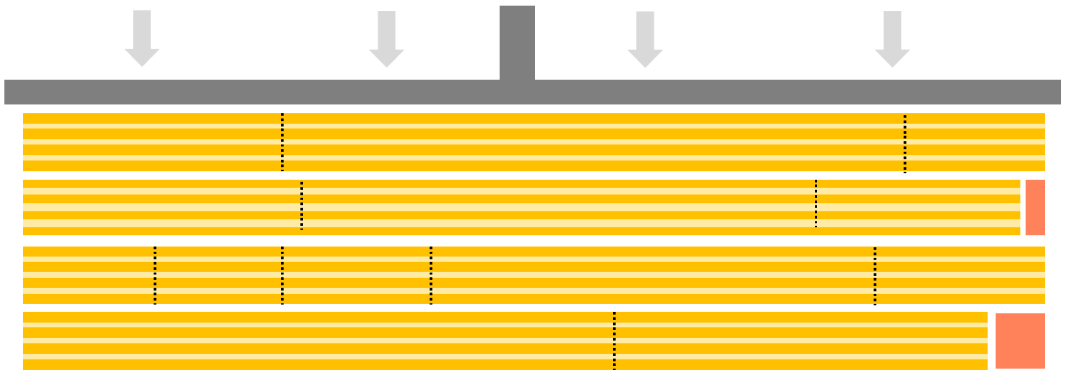

Abbildung 1.1.: Darstellung eines Pressenplans

An einen Pressenplan werden verschiedene Anforderungen wie beispielsweise die Gleichheit der Längen aller Balken in einer Presse gestellt. Neben diesen Anforderungen sind auch einige Optimierungsziele für den Pressenplan von Bedeutung. Darunter zum Beispiel die Minimierung des Rests, der Balken auf die gleiche Länge auffüllt und somit zusätzliche Materialkosten bedeutet.

Germanedge konnte für dieses Problem bereits eine Lösung finden, diese ist allerdings unter Firmenverschluss. Es ist lediglich bekannt, dass jene einen heuristischen Ansatz wählt. Das ermöglicht zwar einerseits das schnelle Ermitteln von Lösungen nahe am Optimum, andererseits gehen dabei aber auch (optimale) Lösungen verloren.

Daher wird in dieser Arbeit ein neuer, vollständiger Ansatz zur Lösung des Problems gewählt. Dieser besteht in der Kodierung der Anforderungen an den Pressenplan als Constraints in einem Constraint-System. Ein Constraint ist hier eine prädikatenlogische Formel, welche unter einer erfüllenden Belegung den Wahrheitswert `Wahr` haben muss. Ein Constraint-System besteht aus der Konjunktion mehrerer Constraints.

In dieser Arbeit wird das Constraint-System für die Satisfiability Module Theories (kurz SMT) mithilfe des Sprachstandards SMTLib Version 2.6 [BFT17] kodiert. Zu typischen Anwendungsbereichen von SMT gehören neben der Programmverifikation [BT18] auch Ressourcenzuordnungsprobleme [Ans+11; LLC21]. Das Pressenplanungsproblem kann als Ressourcenzuordnungsproblem betrachtet werden, sodass sich die Frage stellt, wie es als SMT-Problem in SMTLib kodiert werden kann. Eine Beispieleingabe von Bauteilen könnte folgende sein:

Länge in mm	**Höhe in mm**	**Anzahl**
7000	400	1
13000	400	2
3000	400	3
6000	400	1
9000	400	1
7500	400	1
5000	400	1
5500	400	1
3500	400	1

Tabelle 1.1.: Beispieleingabe von Bauteilen, jede Zeile entpricht einer Bauteilspezifikation

Da das Formulieren der Constraints für einen Pressenplan mit gegebenen Bauteilen händisch in der Constraint-Sprache SMTLib sehr schwer- und fehleranfällig ist (siehe Listing 2), erzeugt ein Programm in einer Gastsprache das Constraint-System für die Constraint-Sprache. Für diese Arbeit wird als jene Gastsprache Haskell [Hud+07] gewählt. Mit der Haskell-Bibliothek `Hasmtlib` [Bru24a] lässt sich beispielhaft das bereits erwähnte Constraint gleicher Länge aller Balken in einer Presse folgend formulieren:

```
1 barLength :: [Bar (Expr RealSort)] -> Expr BoolSort
2 barLength = all
3               (\(b1,b2) -> (b1^.press === b2^.press) ==>
4                 (b1^.length === b2^.length)
5               ) . binom
```

Listing 1: Haskell-Code für das Constraint: $\forall \{b_1, b_2\} \in \binom{\text{Balken}}{2} : \text{presse}(b_1) = \text{presse}(b_2) \rightarrow \text{länge}(b_1) = \text{länge}(b_2)$

Die Bibliothek erzeugt dann das Constraint-System in der Constraint-Sprache SMTLib. Folgend ein Ausschnitt der erzeugten SMTLib-Kodierung für das Constraint aus Listing 1:

```
...
(declare-fun var_69 () Real)
(declare-fun var_70 () Bool)
(declare-fun var_71 () Real)
...
(assert (and (or var_70 (and (>= var_69 1.44) (<= var_69 2.3))) ...))
(assert (and (or var_70 (and (>= var_68 6.0) (<= var_68 24.15))) ...))
(assert (and (and (and (and (and (and (and (= var_28 (+ (+ ...)))))))))))
...
```

Listing 2: Ausschnitt der Kodierung eines Pressenplanungsproblems

Die Kodierung wird an einen SMT-Solver gegeben, welcher die Erfüllbarkeit des Problems bestimmt und für den Fall Erfüllbar eine erfüllende Belegung für die Variablen des Problems ermittelt. Zum Beispiel:

```
sat
(
  ...
  (define-fun var_69 () Real    (/ 287 125))
  (define-fun var_70 () Bool    false)
  (define-fun var_71 () Real    (/ 483 20))
  ...
)
```

Listing 3: Ausschnitt des Solver-Outputs der Lösung eines Pressenplanungsproblems

Jene Belegung transformiert das Programm dann in das gewünschte Ausgabeformat:

Presse	**Schicht**	**Position**	**Länge in mm**	**Höhe in mm**
1	1	1	14000	400
1	1	2	7500	400
1	2	1	3000	400
1	2	2	6500	400
1	2	3	3000	400
...	...	...	...	...

Tabelle 1.2.: Beispielausgabe des Pressenplans, Visualisierung in Abbildung 1.1

Die Struktur dieser Arbeit ist folgende:
in Kapitel 2 wird das Problem der Pressenplanung erklärt, modelliert und nach für das Problem relevanten Zielfunktionen optimiert.
Folgend wird in Kapitel 3 gezeigt, wie Erfüllbarkeitsprobleme der Satisfiability Modulo Theories im Sprachstandard SMTLib Version 2.6 kodiert werden können.
In Kapitel 4 wird auf einige notwendige Grundlagen von Haskell und die Haskell-Bibliothek `Hasmtlib` eingegangen. Zusätzlich werden dort einige Erweiterungen von `Hasmtlib` diskutiert.

Danach wird die Kodierung des in Abschnitt 2.2 erstellten Modells des Pressenplanungsproblems mithilfe von `Hasmtlib` in Kapitel 5 erläutert.

Weiter wird die Performance der Kodierung in Kapitel 6 mithilfe von Laufzeitmessungen ausgewertet und mit der bereits existenten Heuristik verglichen.

Abschließend wird das Ergebnis dieser Arbeit in Kapitel 7 zusammengefasst.

2. Problemdomäne Pressenplanung

In diesem Kapitel wird das Problem der Pressenplanung zunächst erläutert, bevor die prädikatenlogische Modellierung des Problems in einem Modell erfolgt. Abschließend wird das Modell um Optimierungsziele erweitert.

2.1. Anforderungen

Im Pressenplanungsproblem sind drei Entitäten und deren Eigenschaften von elementarer Bedeutung: Pressen, Balken und Bauteile. Diese haben jeweils die Eigenschaften Höhe, Breite und Länge. Zusätzlich gibt es Holzlamellen, welche durch Zersägen eines sogenannten *Endlosbretts* entstehen. Dieses kann von einer Maschine *scheinbar* endlos ausgegeben werden. Höhe und Breite der Lamelle sind fest. Lamellen werden vertikal zu Bauteilen verleimt, sodass Bauteile immer die Form eines Quaders haben. Diese Verleimung ist der eigentliche Grund für die Notwendigkeit des Pressens. Folgend eine Abbildung verleimter Lamellen:

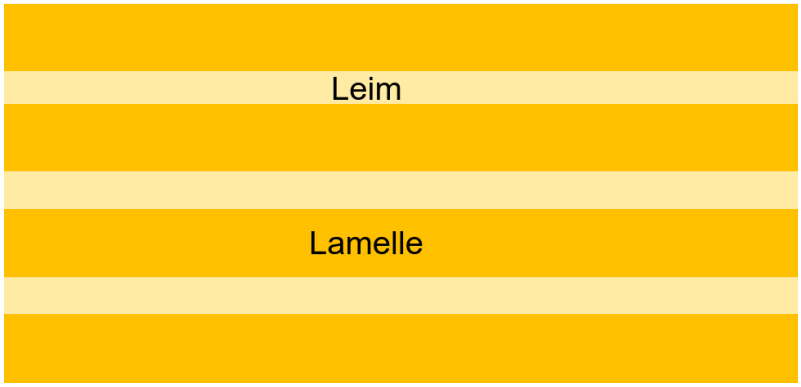

Abbildung 2.1.: Vertikale Verleimung von Lamellen

Bauteile wiederum formen Balken - ebenfalls Quader. Bauteile erhält man durch vertikales Zersägen eines Balkens. Folgend eine Abbildung eines Balkens aus drei Bauteilen mit Rest:

Abbildung 2.2.: Balken aus drei Bauteilen mit Rest, Sägelinien schwarz gepunktet

Balken werden vertikal in Pressen gestapelt und anschließend gepresst. Eine mit Balken befüllte Presse, bei der festgelegt ist, wie die Balken zu Bauteilen nach der Pressung zersägt werden müssen, nennt man auch Pressenplan. In einem Pressenplan ist jedem Bauteil höchstens eine Position in einer Schicht (von unten gezählter Balken) einer Presse zugeordnet. Folgend die Visualisierung eines Pressenplans für eine Presse:

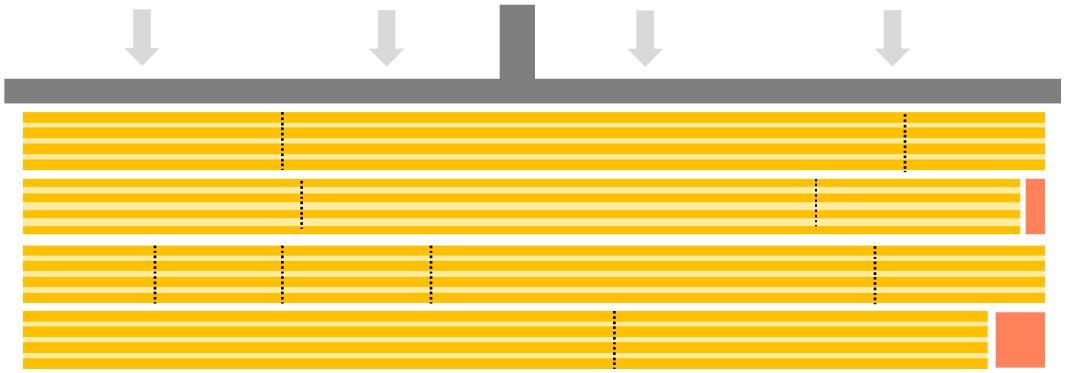

Abbildung 2.3.: Pressenplan für eine Presse mit vier Balken, 13 Bauteilen und zwei Reststücken

Da Pressen im Allgemeinfall in einer Pressung nur Bauteile gleicher Breite pressen können, werden die Bauteile nach Breite vorsortiert. Damit reduziert man die Dimension des Problems vom 3-Dimensionalen auf das 2-Dimensionale, da nur noch Länge und Höhe aller Entitäten berücksichtigt werden muss.

Die Eingabe für das Pressenplanungsproblem ist eine Liste von Bauteilen mit Länge und Höhe sowie deren Anzahl. Diese Eingabe wird auch Auftrag genannt. Folgend ein beispielhafter Auftrag:

Länge in mm	**Höhe in mm**	**Anzahl**
7000	400	1
13000	400	2
3000	400	3
6000	400	1
9000	400	1
7500	400	1
5000	400	1
5500	400	1
3500	400	1

Tabelle 2.1.: Beispielauftrag, jede Zeile entpricht einer Bauteilspezifikation

Die Lösung des Pressenplanungsproblems (Ausgabe) ist ein Pressenplan. Für den Auftrag in Tabelle 2.1 könnte beispielsweise folgende Ausgabe den Pressenplan beschreiben:

Presse	Schicht	Position	Länge in mm	Höhe in mm
1	1	1	14000	400
1	1	2	7500	400
1	2	1	3000	400
1	2	2	6500	400
1	2	3	3000	400
...	...	...	...	...

Tabelle 2.2.: Beispielausgabe des Pressenplans zu Auftrag in Tabelle 2.1, Visualisierung in Abbildung 2.3

An einen Pressenplan bestehen folgende Anforderungen:

1. Pressen haben eine Minimal- und Maximalhöhe. Die Summe der Höhe aller Balken einer Presse muss in diesem Intervall liegen.
2. Pressen haben eine Minimal- und Maximallänge. Die Länge aller Balken einer Presse muss in diesem Intervall liegen.
3. Alle Balken einer Presse haben die gleiche Länge. Dazu kann der Balken so aus Bauteilen zusammengesetzt sein, dass ein Rest übrig bleibt.
4. Alle Bauteile, die einen Balken formen, haben die gleiche Höhe.
5. Jedes Bauteil eines Auftrages ist höchstens einem Balken zugeordnet.
6. Jeder Balken ist genau einer Presse zugeordnet.

2.2. Modellierung

In diesem Abschnitt wird das Modell zur Modellierung des Pressenplanungsproblems erläutert. Alle Namen von Funktionen in jenem Modell werden englisch formuliert, um die Nähe zur später in Abschnitt 5.4 gezeigten Implementierung zu wahren. Pressen heißen also *Press*, Balken (Schichten) werden *Layer* genannt und Bauteile erhalten den Namen *Component*. Die Menge aller Pressen wird mit P, die Menge aller Schichten mit L und die Menge aller Bauteile mit C abgekürzt. Weiter werden boolesche als auch arithmetische Operatoren vorausgesetzt und in der Signatur des Modells nicht genannt.

Folgend die Signatur $\Sigma = \left(\Sigma_F^P \cup \Sigma_F^L \cup \Sigma_F^C, \Sigma_R\right)$ mit Variablen $\mathbb{X} = \mathbb{X}^P \cup \mathbb{X}^L \cup \mathbb{X}^C \cup \mathbb{X}^{InLayer}$

$$\Sigma_F^P = \left\{\begin{array}{ll}(\text{id}, & P \to \mathbb{N}_0),\\ (\text{minLength}, & P \to \mathbb{N}_0),\\ (\text{maxLength}, & P \to \mathbb{N}_0),\\ (\text{length}, & P \to \mathbb{N}_0),\\ (\text{minHeight}, & P \to \mathbb{N}_0),\\ (\text{maxHeight}, & P \to \mathbb{N}_0),\\ (\text{height}, & P \to \mathbb{N}_0),\\ (\text{layers}, & P \to 2^L)\end{array}\right\} \quad \mathbb{X}^P = \bigcup_{p \in P} \{\text{length}(p), \text{height}(p)\}$$

$$\Sigma_F^L = \left\{\begin{array}{ll}(\text{id}, & L \to \mathbb{N}_0),\\ (\text{length}, & L \to \mathbb{N}_0),\\ (\text{height}, & L \to \mathbb{N}_0),\\ (\text{waste}, & L \to \mathbb{N}_0),\\ (\text{isEmpty}, & L \to \mathbb{B})\end{array}\right\} \quad \mathbb{X}^L = \bigcup_{l \in L} \{\text{length}(l), \text{height}(l), \text{waste}(l), \text{isEmpty}(l)\}$$

$$\Sigma_F^C = \left\{\begin{array}{ll}(\text{length}, & C \to \mathbb{N}_0),\\ (\text{height}, & C \to \mathbb{N}_0)\end{array}\right\} \quad \mathbb{X}^C = \emptyset$$

$$\Sigma_R = \left\{(\text{InLayer}, C \times L \to \mathbb{B})\right\} \quad \mathbb{X}^{\text{InLayer}} = \text{InLayer}$$

Die Anforderungen an einen Presseplan werden damit folgend realisiert:

$$L = \dot{\bigcup_{p \in P}} \text{layers}(p) \tag{2.1}$$

$$\forall p \in P : \text{minHeight}(p) \leq \text{height}(p) \leq \text{maxHeight}(p) \tag{2.2}$$

$$\forall p \in P : \text{minLength}(p) \leq \text{length}(p) \leq \text{maxLength}(p) \tag{2.3}$$

$$\forall p \in P,\ \forall l \in \text{layers}(P) : \neg\text{isEmpty}(l) \to \text{length}(p) = \text{length}(l) \tag{2.4}$$

$$\forall c \in C,\ \forall l \in L : \text{InLayer}(c, l) \to \text{height}(c) = \text{height}(l) \tag{2.5}$$

$$\forall c \in C : \text{exactly}(1, \{\text{InLayer}(c, l) \mid l \in L\}) \tag{2.6}$$

$$\forall l \in L : \text{isEmpty}(l) \leftrightarrow \neg \bigvee_{c \in C} \text{InLayer}(c, l) \tag{2.7}$$

$$\forall l \in L : \text{isEmpty}(l) \leftrightarrow (\text{length}(l) = 0 \wedge \text{height}(l) = 0) \tag{2.8}$$

$$\forall p \in P : \text{height}(p) = \sum_{l \in \text{layers}(P)} \text{height}(l) \tag{2.9}$$

$$\forall l \in L : \text{length}(l) = \text{waste}(l) + \sum_{c \in C} \begin{cases}\text{length}(c) & , \text{InLayer}(c, l)\\ 0 & , \text{sonst}\end{cases} \tag{2.10}$$

Constraint 2.2 erfüllt Anforderung 1 aus Abschnitt 2.1, während Constraint 2.3 Anforderung 2 erfüllt. Anforderung 3 wird durch Constraint 2.4 und Anforderung 4

durch Constraint 2.5 abgedeckt. Constraint 2.6 verhärtet Anforderung 5 aus Optimierungszwecken, siehe Abschnitt 2.3. Anforderung 6 ist in diesem Modell mehr Konstruktionsvorschrift als Constraint und ergibt sich aus 2.1. Die Constraints 2.7 und 2.8 definieren, wann eine Presse leer ist. Constraints 2.9 und 2.10 liefern die Definition für Pressenhöhe und Schichtlänge.

2.3. Optimierung

Das obige Modell beschreibt beliebige Lösungen für das Problem der Pressenplanung. Um den Kostenaufwand beim Pressen zu reduzieren, sind Pressenpläne zusätzlich nach folgenden Kriterien lexikografisch zu optimieren (Vorgabe Germanedge):

1. $\min\left(\left|\left\{c \in C \mid \neg \bigvee_{l \in L} \text{InLayer}(c, l)\right\}\right|\right)$
2. $\min(|P|)$
3. $\min\left(\sum_{l \in L} \text{waste}(l)\right)$

Bei Ziel 1, der Minimierung der Anzahl unverplanter Komponenten ist das Optimum immer Null. Für jede Eingabe an Bauteilen existiert eine Lösung, bei welcher alle Bauteile verplant werden. Der Beweis dafür ist Hausaufgabe für den Leser. Diese Optimierung ist mit Constraint 2.6 bereits im Modell festgehalten.

3. Satisfiability Modulo Theories

In diesem Kapitel wird zuerst das Erfüllbarkeitsproblem der Satisfiability Module Theories - kurz SMT - vorgestellt. Dabei wird auch kurz die Kernelemente des Sprachstandards SMTLib Version 2.6 [BFT17] eingegangen. Dieser ermöglicht die Kodierung von SMT-Problemen für SMT-Solver [BT18]. Ein SMT-Solver - kurz Solver - ist ein Programm, welches für ein kodiertes Problem die Erfüllbarkeit dieses und für den Fall *erfüllbar* (englisch *satisfiable* - kurz *sat*) ein Modell für jenes bestimmen kann. Einige bekannte Solver sind zum Beispiel Z3 [DB08], CVC5 [Bar+22], Yices [Dut14] und OpenSMT [Bru+10]. Diese und viele andere nehmen jährlich an dem internationalen Wettbewerb SMT-COMP [BMS05] für SMT-Solver teil, wo sie sich in verschiedenen Disziplinen miteinander messen. Nach dem Überblick über SMT-Lib werden Anzahl-Constraints in SMT betrachtet, bevor am Ende dieses Kapitels beleuchtet wird, wie SMT-Probleme nach Zielfunktionen optimiert werden können.

3.1. Einführung in das Erfüllbarkeitsproblem SMT

Satisfiability Module Theories beschreibt das Erfüllbarkeitsproblem prädikatenlogischer Formeln unter spezifischen Theorien [BT18]. Damit ermöglicht es im Gegensatz zum Erfüllbarkeitsproblem SAT, bei welchem ausschließlich boolesche Unbekannte aussagenlogisch verknüpft werden können, die realitätsnahe Kodierung von Problemen wie beispielsweise Ressourcenzuordnungsproblemen [Ans+11]. Jene sind nachweislich stark NP-schwere kombinatorische Probleme, sodass sie nicht in polynomieller Zeit lösbar sind [DKG13]. Im Allgemeinen ist die Erfüllbarkeit prädikatenlogischer Formeln unentscheidbar [BT18]. Die Eingrenzung von Syntax und Semantik (Interpretation von Funktionssymbolen) der Prädikatenlogik in Modellen von SMT-Problemen ermöglicht die Entscheidbarkeit und erlaubt spezifische Lösungsverfahren unter bestimmten Hintergrund-Theorien (englisch *background-theory*). Diese Theorien werden zu SMT-Logiken wie beispielsweise `QF_LRA` kombiniert. Bei der Logik der quantorenfreien (QF) linearen (L) Real(R)-Arithmetik(A) sind das die Theorien der reellen Zahlen, linearen Arithmetik und Gleichheit/Ungleichheit. Der diesjährige Sieger in jener Logik der SMT-COMP 2024 `OpenSMT` [BBJ24] implementiert als Lösungsverfahren eine Kombination aus CDCL(T) und modifiziertem Simplex [Bru+10]. CDCL(T) is ein vollständiges Lösungsverfahren für prädikatenlogische Probleme unter einer Theorie T und ersetzt alle Atome einer Formel mit booleschen Unbekannten [BT18]. Dieses - nun SAT - Problem wird an einen SAT-Solver gegeben, welcher die Erfüllbarkeit prüft und für den Fall Erfüllbar eine erfüllende Belegung ermittelt. Für jene Belegung überprüft ein (möglicherweise auch mehrere) Theorie-Solver dann deren Konsistenz in der Theorie T. Wenn diese Belegung T-konsistent ist, so ist sie auch Modell für das prädikatenlogische Problem unter der Theorie T. Im Fall der T-Inkonsistenz wird der SAT-Solver erneut aufgerufen. Erwähnenswert sind hier zahlreiche integrierte Verfahren zur Effizienzsteigerung wie zum Beispiel das Lernen abgeleiteter Lemmas.

Für das Pressenplanungsproblem sind neben `QF_LRA` auch `QF_LIA` und `QF_BV` relevante Logiken. In der SMT-COMP 2024 siegte `OpenSMT` in den Logiken `QF_LRA` und `QF_LIA`, während `Bitwuzla` in der Logik `QF_BV` den ersten Platz belegte [BBJ24].

Die Kodierung der SMT-Probleme und damit Eingabe für die Solver ist im Standard SMTLib Version 2.6 festgehalten [BFT17]. Folgend die Darstellung relevanter Kernelemente des Sprachstandards anhand des Beispiels $\exists x \in \mathbb{R}, \exists y \in \mathbb{R} : 5 = x + y$:

```
1  (set-option :produce-models True)
2  (set-logic QF_LRA)
3
4  (declare-fun x () Real)
5  (declare-fun y () Real)
6
7  (assert (= 5 (+ x y)))
8
9  (check-sat)
10 (get-model)
11 (exit)
```

Listing 4: Kodierung von $\exists x \in \mathbb{R}, \exists y \in \mathbb{R} : 5 = x + y$ in SMTLib-Syntax

In Zeile 1 wird der Solver angewiesen, im Fall der Erfüllbarkeit ein erfüllendes Modell zu erstellen. Zeile zwei setzt die Logik für die darauffolgende Problemdeklaration. In den Zeilen 4 und 5 werden Unbekannte der Sorte `Real` mit den Bezeichnern `x` und `y` angelegt. Zeile 7 zeigt die Assertion eines Constraints auf den Unbekannten `x` und `y` und der Konstanten `5`. Zeile 9 weist den Solver an, das deklarierte Problem auf Erfüllbarkeit zu überprüfen, während Zeile 10 für den Fall Erfüllbar ein Modell für das Problem ausgibt. In Zeile 11 wird die Interaktion mit dem Solver beendet.

3.2. SMT-Kodierung von Anzahl-Constraints

Oftmals werden zur Kodierung von SMT-Problemen auch Anzahl-Constraints benötigt [KGK19]. Auf einer Menge von n gegebenen booleschen Variablen $\{b_0, b_1, \ldots, b_{n-1}\}$ soll dabei die Anzahl wahr belegter Variablen nach k restriktiert werden. Dabei existieren die folgenden drei Restriktionen:

- Mindestens k: $\text{atLeast}(k, \{b_0, b_1, \ldots, b_{n-1}\})$
- Genau k: $\text{exactly}(k, \{b_0, b_1, \ldots, b_{n-1}\})$
- Höchstens k: $\text{atMost}(k, \{b_0, b_1, \ldots, b_{n-1}\})$

Anzahl-Constraints sind nicht Teil des SMTLib-Standards Version 2.6 [BFT17], sodass nur wenige Solver wie Z3 solche Kodierungen direkt unterstützen [Bjø+19]. Daher wird zur manuellen Kodierung oft die ITE-Kodierung verwendet [KGK19]. Dabei werden die einzelnen booleschen Unbekannten mithilfe eines If-Then-Else-Ausdrucks (kurz ITE) in numerische Ausdrücke umgewandelt, anschließend summiert und verglichen. Mit der Hilfsfunktion $\text{count}(\{b_0, b_1, \ldots, b_{n-1}\})$ lassen sich die Anzahl-Constraints einfach kodieren:

$$\text{count}(B) = \sum_{b \in B} \left(\begin{cases} 1 & , b \\ 0 & , \text{sonst} \end{cases} \right)$$

$$\text{atLeast}(k, B) = k \leq \mathit{count}(B)$$

$$\text{exactly}(k, B) = k = \mathit{count}(B)$$

$$\text{atMost}(k, B) = k \geq \mathit{count}(B)$$

Je nach gewählter Logik erfolgt die Summierung in der entsprechenden Sorte. Ist für das SMT-Problem beispielsweise die Logik QF_LRA gewählt, dann summiert count(B) Terme der Sorte `Real`. Dabei entstehen für jedes Anzahl-Constraint offensichtlich n ITE-Ausdrücke, $n-1$ binäre Additionen und ein Vergleich.

Für bestimmte k wie beispielsweise $k = 0$ oder $k = 1$ können Anzahl-Constraints allerdings von spezialisierten Kodierungen profitieren. Einige davon sind folgende:

- $\text{atLeast}(0, B) = \text{True}$
- $\text{atLeast}(1, B) = \bigvee_{b \in B} b$
- $\text{atMost}(0, B) = \neg \bigwedge_{b \in B} b$
- $\text{atMost}(1, B) = \text{productEncoding}(B)$
- $\text{exactly}(0, B) = \neg \bigwedge_{b \in B} b$
- $\text{exactly}(1, B) = \text{atLeast}(1, B) \wedge \text{atMost}(1, B)$

Die Produkt-Kodierung (englisch *product-encoding*) ist eine aus SAT bekannte Kodierung für atMost($1, B$) [Che10]. Dabei werden die $n = |B|$ Unbekannten in eine Matrix $M^{q \times p}$ mit $p = \lceil \sqrt{n} \rceil$ und $q = \lceil n/p \rceil$ gelegt. Wenn $n < pq$ gilt, dann werden die übrigen Matrixeinträge mit `False` belegt. Zusätzlich wird für jeden Zeilenindex $j \in \{0, \ldots, q-1\}$ eine Hilfsvariable v_j und jeden Spaltenindex $i \in \{0, \ldots, p-1\}$ eine Hilfsvariable u_i angelegt, welche repräsentiert, ob in der jeweiligen Zeile j oder Spalte i mindestens ein $b \in B$ wahr ist. Fasst man die Hilfsvariablen zu $U = \{u_0, \ldots, u_{p-1}\}$ und $V = \{v_0, \ldots, v_{q-1}\}$ zusammen, beschreibt folgende Gleichung, dass höchstens eine der n Unbekannten wahr ist:

$$\text{productEncoding}(B) =$$
$$\text{atMost}(1, U) \wedge \text{atMost}(1, V) \wedge \bigwedge_{\substack{0 \leq x \leq n-1, x=(i-1)(q-1)+j \\ 0 \leq i \leq p-1, 0 \leq j \leq q-1}} ((\neg b_x \vee u_i) \wedge (\neg b_x \vee v_j))$$

Daraus folgen $2\sqrt{n} + \mathcal{O}(\sqrt[4]{n})$ existentiell quantifizierte Hilfsvariablen und $2n + 4\sqrt{n} + \mathcal{O}(\sqrt[4]{n})$ Klauseln (konjugierte Disjunktionen). Das ist die nachweisbar minimal notwendige Anzahl an Klauseln für das At-Most-One-Constraint [Che10; KSV19].

Aus softwaretechnischer Sicht und wie später in Abschnitt 4.2.1 demonstriert, wäre eine Kodierung ohne Hilfsvariablen von Vorteil. Die Produkt-Kodierung kann

dazu einfach angepasst werden, indem alle Hilfsvariablen $u_i \in U$ und $v_j \in V$ folgend ersetzt werden:

$$u_i \mapsto \bigvee_{m_{j,i} \in M^{q \times p}, 0 \leq j \leq q-1} m_{j,i} \quad \text{und} \quad v_j \mapsto \bigvee_{m_{j,i} \in M^{q \times p}, 0 \leq i \leq p-1} m_{j,i}$$

Das eliminiert die Hilfsvariablen und behält die Klauselanzahl bei. Allerdings sind die entstehenden Klauseln um $\mathcal{O}(\sqrt{n})$ größer.

3.3. Optimierung von SMT-Problemen

Häufig sind bei SMT-Problemen zusätzlich auch spezifische Zielfunktionen von Interesse. Diese haben besonders in der Industrie 4.0 eine große Relevanz [TBT24], so auch beim Pressenplanungsproblem. Die Anzahl unverplanter Bauteile, eingesetzter Pressen und notwendigen Rests soll minimiert werden, siehe Abschnitt 2.3.

SMT-Probleme, bei denen zusätzlich nach bestimmten Zielfunktionen optimiert wird, fallen in eine neue Problemklasse, jene der Optimization Module Theories (kurz OMT) [TBT24]. Genau wie die lineare Optimierung unterstützt auch OMT die Optimierung nach linearen Zielfunktionen. Im Gegensatz zu ersterer, erlaubt OMT zusätzlich auch beliebige boolesche Kombinationen aus Atomen, wodurch die Modellierung komplexer (industrieller) Probleme vereinfacht wird.

Wenige SMT-Solver wie beispielsweise `Z3` unterstützen die direkte Kodierung von OMT-Zielfunktionen in SMTLib-Syntax [BPF15]. Folgend die Maximierung des Terms $x + y$ unter der Formel $\exists x \in \mathbb{Z}, \exists y \in \mathbb{Z} : x < 10 \wedge y < 5 \wedge (y < 7 \rightarrow x = 1)$:

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (and (< x 10) (and (< y 5) (=> (< y 7) (= x 1)))))
(maximize (+ x y))
```

Listing 5: Maximierung von $x + y$ unter $\exists x \in \mathbb{Z}, \exists y \in \mathbb{Z} : x < 10 \wedge y < 5 \wedge (y < 7 \rightarrow x = 1)$

Der Solver (hier `Z3`) ermittelt korrekt die optimale Lösung $x \coloneqq 1, y \coloneqq 4$.

4. Haskell und Hasmtlib

In diesem Kapitel werden einige notwendige Grundlagen der funktionalen Programmiersprache Haskell erklärt. Danach wird der Kern der Funktionsweise der Haskell-Bibliothek `Hasmtlib` [Bru24a] beschrieben. Ferner werden einige geeignete Erweiterungen dieser diskutiert.

4.1. Funktionale Programmierung mit Haskell

Was ist ein ADT? Polymorphismus. Pattern-Matching. Was ist ein GADT und was kann er besser als ein ADT? Typ- und Kind-System anreißen - Phasentrennung. Typklassen. Typeoperators. Was ist eine pure Funktion? Wie kann man Nebenwirkungen dennoch modellieren - was ist eine Monade? Was ist Lazy?

4.2. Hasmtlib - Haskell-Bibliothek für SMTLib

Die Haskell-Bibliothek `Hasmtlib` [Bru24a] ist eine eingebettete domänenspezifische Sprache (englisch *embedded domain-specific language* - kurz *eDSL*) für den SMTLib-Standard Version 2.6 [BFT17] in Haskell. Eine eDSL ist eine, in einer Gastsprache eingebettete, anwendungsspezifische Sprache, welche auf Grundkonstrukten der Gastsprache aufbaut und somit den Implementierungsaufwand einer eDSL erheblich reduziert [SCK04].

Haskell wird aufgrund seines starken Typsystems oft als Gastsprache für solche eingebetteten Sprachen verwendet [Gil14]. Listing 6 zeigt beispielsweise, wie `Hasmtlib` SMTLib-Ausdrücke als abstrakten Syntaxbaum mit einem GADT repräsentiert.

```haskell
data SMTSort =  BoolSort | IntSort | RealSort | ...

data Expr (t :: SMTSort) where
  Var       :: SMTVar t -> Expr t
  Constant  :: Value  t -> Expr t
  Plus      :: Num (HaskellType t) => Expr t -> Expr t -> Expr t
  ...
  IDiv      :: Expr IntSort  -> Expr IntSort  -> Expr IntSort
  Div       :: Expr RealSort -> Expr RealSort -> Expr RealSort
  LTH       :: (Ord (HaskellType t), KnownSMTSort t) =>
                                    Expr t -> Expr t -> Expr BoolSort
  ...
  Not       :: Boolean (HaskellType t) => Expr t -> Expr t
  And       :: Boolean (HaskellType t) => Expr t -> Expr t -> Expr t
  ...
  Sqrt      :: Expr RealSort -> Expr RealSort
```

Listing 6: Repräsentation von SMTLib-Ausdrücken mithilfe eines GADTs in `Hasmtlib` [Bru24b]

Dabei ist der GADT `Expr t` polymorph in der `SMTSort`, sodass dessen Kind `Expr t :: SMTSort -> Type` ist. Da Werte des Datentyps `SMTSort` hier als Typen verwendet werden, nennt man `SMTSort` auch *Data-Kind* oder *Promoted-Type* [EW12]. Besonders relevant dafür ist das in 4.1 erklärte *Type-Refinement*. So weist beispielsweise ein Pattern-Match auf den Konstruktor `Sqrt` bereits zur Übersetzungszeit nach, dass `t ~ RealSort` gilt. Bei anderen Konstruktoren wie `And` kann die Belegung des `t` in `Expr t` mithilfe von Singletons ermittelt werden. Ein *Singleton* ist ein Laufzeit-Zeuge (Wert) für genau einen Übersetzungszeit-Typen [EW12]. Diese Abhängigkeit ist aufgrund Haskell's strenger Phasentrennung notwendig und ermöglicht eine *dependent*-ähnliche Kodierung wie in den Dependently-Typed Programmiersprachen Idris [BRA13] und Agda [Nor09].

Jeder Typ vom Kind `SMTSort` erhält also einen Singleton (Wert). Wie in Listing 7 zu sehen, stellt `Hasmtlib` den Singleton-Typen `SSMTSort` ebenfalls als GADT dar.

```haskell
data SSMTSort (t :: SMTSort) where
  SIntSort    :: SSMTSort IntSort
  SRealSort   :: SSMTSort RealSort
  SBoolSort   :: SSMTSort BoolSort
  ...
```

Listing 7: Repräsentation des Singleton-Typs SSMTSort in `Hasmtlib` [Bru24c]

Die Verbindung zwischen Typ und Wert wird mit der in Listing 8 dargestellten Typklasse `KnownSMTSort` erreicht.

```haskell
class    KnownSMTSort (t :: SMTSort) where sortSing :: SSMTSort t
instance KnownSMTSort IntSort        where sortSing = SIntSort
instance KnownSMTSort RealSort       where sortSing = SRealSort
instance KnownSMTSort BoolSort       where sortSing = SBoolSort
...
```

Listing 8: Typklasse zur Verbindung von Typ SMTSort zu Wert SSMTSort in `Hasmtlib` [Bru24c]

Mit den genannten Mechanismen garantiert `Hasmtlib` die ausschließliche Erzeugung von sauber typisierten (englisch *well-typed*) Ausdrücken [Pey+06]. Diese werden anschließend von der Bibliothek in SMTLib-Syntax übersetzt und an einen SMT-Solver gegeben. Dessen Antwort parst und übersetzt `Hasmtlib` dann zurück in die entsprechenden Haskell-Typen. Folgend ein kleines Programm zur Erläuterung:

```haskell
module Example.Foo where

import Language.Hasmtlib
import Control.Monad (replicateM)

data Bar a = Bar
  { length :: a
  , height :: a }
  deriving (Generic)
instance Variable a => Variable (Bar a)

barLength :: [Bar (Expr RealSort)] -> Expr BoolSort
barLength = all (\(b1,b2) -> ...) . binom
  where
    binom = ...

main :: IO ()
main = do
  result <- solveWith @SMT (solver cvc5) $ do
    setLogic "QF_LRA"
    -- create variables
    bars <- replicateM 10 variable
    -- constrain variables
    assert $ barLength bars
    -- find model for variables
    return foo
  ...
```

Listing 9: Beispielhafte Verwendung von Hasmtlib

Das Beispiel in Listing 9 demonstriert mit der Funktion `barLength` außerdem die pure Formelkonstruktion in `Hasmtlib`. Lediglich Variablenerzeugung (Zeile 22)

und Formel-Assertion (Zeile 24) sind monadisch. Dadurch lassen sich viele Funktionalitäten der Gastsprache wiederverwenden, zum Beispiel die Funktion `sum` aus `Data.Foldable`: `sum :: (Foldable t, Num a) => t a -> a`.

4.2.1. Ergänzung um Anzahl-Constraints

`Hasmtlib` unterstützte bis zum Zeitpunkt dieser Arbeit keine Anzahl-Constraints. Daher werden diese im Rahmen dieser Arbeit der Bibliothek hinzugefügt. Die Implementierung erfolgt analog zu den in 3.2 präsentierten Kodierungen. Folgend ein Ausschnitt für das At-Most-Constraint:

```haskell
count :: forall t f. (Functor f, Foldable f, Num (Expr t))
  => f (Expr BoolSort) -> Expr t
count = sum . fmap (\b -> ite b 1 0)

atMost  :: forall t f. (Functor f, Foldable f, KnownSMTSort t
  , Num (HaskellType t), Ord (HaskellType t))
  => Expr t -> f (Expr BoolSort) -> Expr BoolSort
atMost 0 = nand
atMost 1 = amoSqrt
atMost k = (<=? k) . count

exactly  :: forall t f. (Functor f, Foldable f, KnownSMTSort t
  , Num (HaskellType t), Ord (HaskellType t))
  => Expr t -> f (Expr BoolSort) -> Expr BoolSort
exactly 0 xs = nand xs
exactly 1 xs = atMost @t 1 xs && or xs
exactly k xs = count xs === k

-- Pure product-encoding
amoSqrt :: (Foldable f, Boolean b) => f b -> b
amoSqrt xs
  | length xs < 10 = amoQuad $ toList xs -- pair-wise encoding
  | otherwise =
      let n = toInteger $ length xs
          p = ceiling $ sqrt $ fromInteger n
          rows = splitEvery (fromInteger p) $ toList xs
          columns = transpose rows
          vs = or <$> rows
          us = or <$> columns
       in amoSqrt vs && amoSqrt us &&
          and (
            imap
            (\j r -> and $
                imap (\i x -> (x ==> us !! i) && (x ==> vs !! j)) r)
            rows
          )
  where
    splitEvery n = takeWhile (not . null) . map (take n) . iterate (drop n)
```

Listing 10: At-Most-One-Constraint in `Hasmtlib` [Bru24d]

4.2.2. Ergänzung um Incremental Refinement

`Hasmtlib` kann aufbauend auf SMT-Problemen auch OMT-Probleme kodieren. Da dies allerdings nur wenige SMT-Solver wie `Z3` direkt unterstützen, verwendet man häufig sogenannte Optimierungsschleifen [BPF15]. Dabei werden iterativ bessere Lösungen gesucht, indem obere Schranken für Maxima oder untere Schranken für Minima verfeinert werden. SMT-Solver haben einen Incremental-Stack, welcher die

Wiederverwendung von bereits definierten Problemen und auch erlernten Lemmas ermöglicht, sodass *Incremental Refinement* wesentlich effizienter als das Starten eines neuen Solver-Prozesses für jede Verfeinerung ist [Liu+14].

Im Rahmen dieser Arbeit wird `Hasmtlib` daher um eine solche Optimierungsschleife folgend erweitert:

```
1  solveMinimized = solveOptimized (<?)
2  solveMaximized = solveOptimized (>?)
3
4  solveOptimized :: (MonadIncrSMT Pipe m, MonadIO m, KnownSMTSort t)
5    => (Expr t -> Expr t -> Expr BoolSort) -- optimization comparison
6    -> Expr t                              -- term to optimize
7    -> Maybe (Expr t -> Expr t)            -- step-size-adjustment
8    -> m (Result, Solution)
9  solveOptimized op goal mStep = refine Unsat mempty goal
10   where
11     refine oldRes oldSol target = do
12       res <- checkSat
13       case res of
14         Sat   -> do
15           sol <- getModel
16           case decode sol target of
17             Nothing        -> return (Sat, mempty)
18             Just targetSol -> do
19               push
20               let step = fromMaybe id mStep
21               assert $ target `op` step (encode targetSol)
22               refine res sol target
23         _ -> do
24           pop
25           case mStep of
26             Nothing -> return (oldRes, oldSol)
27             Just _  -> solveOptimized op goal Nothing
```

Listing 11: Incremental Refinement in `Hasmtlib` [Bru24d]

Wie in Listing 11 zu erkennen ist, wurde außerdem eine Schrittweite für das Anpassen der verfeinerten Schranken hinzugefügt. Um nicht über die optimale Lösung zu schreiten, ruft Zeile 27 die Optimierungsschleife nochmals - aber ohne Schrittweite - auf. Die letzte bekannte optimale Schranke liegt dabei noch auf dem Incremental-Stack, sodass der rekursive Aufruf in Zeile 27 mit dieser als neue Schranke startet. Das garantiert das Finden der optimalen Lösung.

5. Implementierung

In diesem Kapitel wird gezeigt, wie das Problem der Pressenplanung mit Haskell und `Hasmtlib` in SMTLib kodiert werden kann. Dazu werden zunächst die notwendigen Datentypen und die Sorten-Polymorphie der Kodierung erklärt, bevor einige konkrete Repräsentationen der Constraints aus Abschnitt 2.2 mithilfe von Quellcode-Ausschnitten dargestellt werden. Danach werden Ein- und Ausgabe kurz beschrieben. Anschließend wird dargelegt, wie die Implementierung mit bestimmten Parametern konfiguriert werden kann und wie diese die Laufzeit des Solvers beeinflussen. Am Ende wird eine Möglichkeit zur Parallelisierung des Programms demonstriert.

5.1. Datentypen und Unbekannte

Analog zur Problemdomäne aus Kapitel 2 sind hier Bauteile (Components), Schichten (Layers) und Pressen (Presses) die Entitäten von Bedeutung. Deren Repräsentation erfolgt jeweils durch Record-Datenstrukturen. Bei Pressen wird zusätzlich in statische und symbolische Pressen unterschieden. Statische Pressen (`StaticPress`) beschreiben die konstanten Eigenschaften von Pressen wie beispielsweise deren maximale Höhe. Symbolische Pressen (`Press`) haben die Konstanten einer statischen Presse und beschreiben weiter symbolische Eigenschaften wie die geplante Länge in einem Pressenplan. Diese Unterscheidung ist für die Trennung von Eingabe (Einlesen der Pressenspezifikationen) und Verarbeitung der Pressen im Constraint-System sinnvoll. Folgend deren Haskell-Repräsentation:

```haskell
data Component a = Component
  { _cId        :: Integer
  , _compLength :: a
  , _compHeight :: a
  } deriving stock (Show, Eq, Ord, Functor, Foldable, Traversable, Generic)
    deriving anyclass (Codec)
$(makeLenses ''Component)

data Layer a b = Layer
  { _lId          :: Integer
  , _layerLength  :: a
  , _layerHeight  :: a
  , _layerEmpty   :: b
  , _waste        :: a
  } deriving stock (Show, Eq, Ord, Generic)
    deriving anyclass (Codec)
$(makeLenses ''Layer)

data StaticPress a = StaticPress
  { _pId            :: Integer
  , _pressName      :: String
  , _pressMinLength :: a
  , _pressMaxLength :: a
  , _pressMinHeight :: a
  , _pressMaxHeight :: a
  } deriving stock (Show, Eq, Ord, Functor, Foldable, Traversable, Generic)
    deriving anyclass (Codec)
$(makeLenses ''StaticPress)

data Press a b = Press
  { _press       :: StaticPress a
  , _pressLength :: a
  , _pressHeight :: a
  , _pressLayers :: [Layer a b]
  } deriving stock (Show, Eq, Ord, Generic)
    deriving anyclass (Codec)
$(makeLenses ''Press)
```

Listing 12: Haskell-Datentypen der Domäne Pressenplanung

Der Ausdruck `$(makeLenses ''DATATYPE)` taucht dabei mehrmals auf. Dieser erzeugt *Lenses* für die Datentypen. Da im Weiteren Lenses häufig in der Implementierung verwendet werden, wird dem Leser empfohlen, folgende Quelle bei Fragen zu Lenses zu konsultieren: [Kme24].

Ferner ist auffällig, dass alle Typen polymorph sind, sodass `Layer a b` beispielsweise zwei Typen `a` und `b` nimmt. Das `a` repräsentiert einen Zahlentypen und das `b` einen Bool-Typen. Das vereinfacht später die Kodierung und Dekodierung der symbolischen Werte. So könnte die symbolische Schicht mit `type SLayer t = Layer`

`(Expr t) (Expr BoolSort)` und die konkrete Schicht mit `type CLayer t = Layer (HaskellType t) Bool` dargestellt werden.

Mit diesen Grundtypen kann nun der Typ `Pressenplan` definiert werden:

```
1 data PressPlan t = PressPlan
2   { _components :: [Component (Expr t)]
3   , _layers     :: [Layer (Expr t) (Expr BoolSort)]
4   , _presses    :: [Press (Expr t) (Expr BoolSort)]
5   , _inLayer    :: Relation Integer Integer
6   } deriving stock (Generic)
7 $(makeLenses ''PressPlan)
```

Listing 13: Haskell-Datentyp des Pressenplans

Dieser enthält alle symbolischen Komponenten, Schichten, Pressen und auch die Relation *InLayer* aus Abschnitt 2.2. Als Index für diese wird das Tupel `(cId, lId)` aus Komponenten-ID und Schicht-ID verwendet.

Die Erzeugung eines symbolischen Pressenplans ist monadisch (`setupPressPlan :: MonadSMT s m => ... -> m (PressPlan t)`), da dort alle notwendigen Variablen angelegt werden. Identisch zur Variablenmenge $\mathbb{X}$ in Abschnitt 2.2 sind das:

- `_pressLength` und `_pressHeight` für jede Presse,
- `_layerLength`, `_layerHeight`, `_waste` und `_layerEmpty` für jede Schicht und
- ein symbolischer Wahrheitswert für jedes mögliche Element der Relation `InLayer`

Bei weiterer Betrachtung des Typs `Pressenplan` fällt auf, dass Komponenten, Schichten und Pressen im Typ Liste vorliegen. Der Listentyp ist der von Haskell, also konkret in Haskell und nicht symbolisch im SMT-Problem. Das erklärt die Nutzung der Liste für die Komponenten, da diese eine Eingabe sind, welche konkret darstellbar ist. Aber wie ist der Listentyp für Schichten und Pressen zu erklären?

Das Modell in Abschnitt 2.2 all-quantifiziert Schichten und Pressen. Das wäre analog auch in SMT möglich. Im Allgemeinen werden solche Probleme aber grundinstanziiert, um die Verwendung ineffizient (oder gar nicht) entscheidbarer Quantoren-Logiken zu vermeiden [GBT07; GM09]. Für jede Presse wird dabei im Voraus die höchstmögliche Anzahl an Schichten basierend auf der Eingabe an Komponenten ermittelt, genauer in 5.2. Da Schichten leer sein können haben sie das Attribut `_layerEmpty :: b` (Listing 12, Zeile 13). Identisch könnten Pressenpläne leere Pressen enthalten. Weiter könnte eine obere Schranke für die Anzahl an Pressen basierend auf der Eingabe an Komponenten ermittelt werden. Allerdings bedeutet jede weitere Presse weitere Unbekannte und weitere Constraints, sodass die Komplexität des Problems erhöht wird, was letztendlich zu einer ineffizienteren Lösung des Solvers führt. Daher wird jeder Auftrag mehrmals kodiert - mit jeweils unterschiedlich fester Anzahl an Pressen. Diese werden dann als separate Probleminstanzen auf verschiedene Prozessor-Kerne aufgeteilt, mehr dazu in Abschnitt **??**.

5.2. Vorverarbeitung

5.3. Sorten-Polymorphie

Wie in Kapitel 3 gezeigt bietet SMTLib verschiedene Logiken zur Kodierung von SMT-Problemen an. Nicht immer ist dabei vorher klar, in welcher Logik ein Problem am effizientesten für Solver kodiert werden kann. Das gilt auch für das Pressenplanungsproblem mit den Kandidaten `QF_LIA`, `QF_LRA` und `QF_BV`. `Hasmtlibs` Ausdrucks-Typ `Expr t` kann dafür genutzt werden, um das Problem polymorph in der Sorte zu implementieren. Betrachten wir folgende Funktion, die für einen Pressenplan und dessen Konfiguration alle Constraints für jenen erstellt:

```
problem :: (Orderable (Expr t), Num (Expr t))
    => PressPlan t
    -> Config (Expr t)
    -> [Expr BoolSort]
problem pp cfg = ...
```

Listing 14: Sorten-Polymorphie bei der Problemdeklaration

Der Aufrufer wählt die Sorte der Kodierung dann per Typ-Applikation und setzt die Logik mithilfe der Singletons:

```
runProblem = forall t. (KnownSMTSort t, Orderable (Expr t)
  , Num (Expr t), MonadSMT s m)
  => ... -> m ()
runProblem ... = do
  ...
  setLogic $ case sortSing @t of
    SIntSort    -> "QF_LIA"
    SRealSort   -> "QF_LRA"
    SBvSort _ _ -> "QF_BV"
    _           -> "ALL"
  ...
  forM_ (problem @t _ _) assert
  ...
```

Listing 15: Auswahl der Logik nach Sorte

Das ermöglicht die Wahl der Kodierung für den Aufrufer der Schnittstelle und verhindert somit Redundanz in der Implementierung. Insbesondere begünstigt das die einfache Testung verschiedener Kombinationen von Solvern und Kodierungen in verschiedenen Logiken wie später bei der Auswertung in Kapitel 6.

Für die Sorte `Real` werden alle Längenangaben in der Einheit Meter kodiert, sodass dabei offensichtlich keine Lösungen aufgrund von unzureichender Genauigkeit verloren gehen kann. Bei den Sorten `Int` und `BitVec` wird hingegen die Einheit Millimeter verwendet. Da alle Eingaben durch natürliche Zahlen in Millimetern dargestellt werden und im Constraint-System keine Division erfolgt, geht auch hier keine

Lösung aufgrund von Ungenauigkeit verloren. Für `BitVec` reichen 15 Bit bei vorzeichenloser Interpretation aus, da die größten Werte die Längen der Pressen (höchstens 25000 mm) immer kleiner als $2^{15} = 32768$ sind. Wie in [NPZ24] beschrieben, skalieren Bitvektor-Kodierungen allerdings schlecht in Bezug auf die Bitbreite. Bei deren Reduktion auf äquivalente SAT-Kodierungen entstehen dabei besonders für arithmetische Operationen große Formeln. Obwohl für den Anwendungsfall Pressenplanung hier nur Bitvektoren der Länge 15 verwendet werden, führt die Verwendung von zahlreichen arithmetischen Operationen vermutlich zu ineffizienten Kodierungen. Diese Vermutung konnte durch - in dieser Arbeit nicht dokumentierte - Testfälle bestätigt werden. Daher werden im folgenden der Arbeit Bitvektoren nicht weiter für die Kodierung in Betracht gezogen.

5.4. Kodierung der Constraints

Die Constraints aus Abschnitt 2.2 lassen sich dank `Hasmtlib` analog in Haskell formulieren. Betrachten wir die Kodierung weniger Constraints wie 2.2 und 2.3 aus Abschnitt 2.2:

```
1 pressBounds :: Orderable (Expr t) => PressPlan t -> Expr BoolSort
2 pressBounds = all
3   (\p -> p^.pressHeight >=? p^.press.pressMinHeight
4       && p^.pressHeight <=? p^.press.pressMaxHeight
5       && p^.pressLength >=? p^.press.pressMinLength
6       && p^.pressLength <=? p^.press.pressMaxLength
7   ) . toListOf (presses.folded)
```

Listing 16: Haskell-Kodierung der Constraints 2.2 und 2.3 aus Abschnitt 2.2

Hier ist darauf hinzuweisen, dass die Funktion `all` in Zeile 2 nicht all-quantifiziert, sondern über allen Pressen des Problems grundinstanziiert. Dabei entsteht also aus der Konjunktion der Anforderungen an jede einzelne Presse eine Formel, welche die Anforderung für alle Pressen kodiert. Das bestätigt auch der Rückgabetyp `Expr BoolSort` der Funktion `pressBounds`.

Betrachten wir weiter folgend das Constraint 2.6:

```
1 compInAtMostOneLayer :: forall t. (Orderable (Expr t), Num (Expr t))
2   => PressPlan t -> Expr BoolSort
3 compInAtMostOneLayer pp = all (exactly @t 1 . image rel) (domain rel)
4   where
5     rel = pp^.inLayer
```

Listing 17: Haskell-Kodierung des Constraints 2.6

Es beschreibt, dass jede Komponente genau einer Schicht zugeordnet wird. Die Funktion `exactly` in Zeile 3 ist dabei die in Abschnitt 3.2 erklärte und in Abschnitt 4.2.1 implementierte Exactly-One-Kodierung.

Zusätzlich zu den Constraints des Modells in Abschnitt 2.2 muss folgendes Constraint hinzugefügt werden.

```
compInAtMostOneLayer :: forall t. (Orderable (Expr t), Num (Expr t))
  => PressPlan t -> Expr BoolSort
compInAtMostOneLayer pp = all (exactly @t 1 . image rel) (domain rel)
  where
    rel = pp^.inLayer
```

Listing 18: Haskell-Kodierung des Constraints 2.6

Im Modell sind die Unbekannten Elemente der Menge $\mathbb{N}_0$. In SMTLib repräsentieren die Sorten `Int` und `Real` jedoch die Mengen $\mathbb{Z}$ und $\mathbb{R}$. Daher muss sichergestellt werden, dass alle Unbekannten nicht-negativ sind. Sowohl für Pressenlänge und Pressenhöhe als auch Schichtlänge und Schichthöhe ergibt sich das bereits aus den Constraints in Abschnitt 2.2. Wie aus Listing 18 zu schließen allerdings nicht für den Schichtrest, sodass obiges Constraint hinzugefügt werden muss.

5.5. Parallelisierung und Optimierung

In Abschnitt 2.3 wurden drei Optimierungsziele genannt. Optimierungsziel 1, die Minimierung unverplanter Komponenten ist wie dort bereits beschrieben immer durch einen Pressenplan mit genau null übrigen Komponenten erfüllbar. Optimierungsziel 2 ist die Minimierung der Anzahl geplanter Pressen. Da erfahrungsgemäß selbst bei den größten Aufträgen höchstens zehn Pressen pro Auftrag geplant werden, gilt also $1 \leq |P| \leq 10$. Wird ein Auftrag also zehnmal mit jeweils unterschiedlicher Pressen-Anzahl kodiert, dann enthält eine jener Kodierungen die optimale Belegung. Optimal ist jene Belegung in entsprechender Kodierung genau dann, wenn der Gesamtrest unter der Belegung minimal ist (Optimierungsziel 3). Das kleinste Optimum des Gesamtrests liegt bestenfalls bei null. Durch das Hinzufügen des Constraints $\forall p \in P : 0 = \sum_{l \in \text{layers}(P)} \text{waste}(l)$ zur Kodierung mit der minimalen Pressen-Anzahl kann dieses Optimum forciert werden. Ist das Problem nach Hinzufügen des Constraints noch immer erfüllbar, dann ist jede erfüllende Belegung des Problems optimal. Ein solches zusätzliches Constraint schränkt den Lösungsraum des Problems erheblich ein, sodass die Lösung des Problems durch den Solver wesentlich effizienter wird. Hier nicht dokumentierte Testfälle bestätigen diese Annahme.

Allerdings kann nicht garantiert werden, dass es eine Lösung ohne Rest gibt. Daher wird ein Auftrag 20 Mal kodiert: Auf zehn verschiedene Pressen-Anzahlen in jeweils zwei Variationen: Mit dem extra Constraint, welches den Gesamtrest auf null fixiert und ohne dieses. Diese 20 verschiedenen Kodierungen eines Auftrages werden bestenfalls auf 20 verschiedene Kerne einer CPU (oder mehrere) verteilt. Die Maschine des Kunden, welcher Germanedge mit dem Pressenplanungsproblem beauftragt hat, verfügt allerdings nur über eine CPU mit acht Kernen. Daher wird ein Auftrag mit vier verschiedenen Pressen-Anzahlen kodiert, sodass insgesamt acht Kodierungen des Auftrags entstehen. Ein Eingabeparameter $p \in \mathbb{N}^+$ des Programms bestimmt dabei die Mindestanzahl an Pressen. Für ein bestimmtes p wird ein Auftrag dann für p, $p+1$, $p+2$ und $p+3$ kodiert. Dabei könnte offensichtlich die optimale Lösung mit

Pressen-Anzahl $p\prime$ verloren gehen, da $p\prime < p \oplus p\prime > p+3$ gelten könnte. Ähnlich wie bei der Ermittlung der maximalen Schichtanzahl pro Presse in Abschnitt 5.2 könnte auch hier ein (vermutlich linearer) Algorithmus entwickelt werden, der die Mindestanzahl an Pressen ermittelt. Da in dieser Arbeit allerdings lediglich ein Prototyp und kein vollständiges Produkt zur optimalen Lösung des Pressenplanungsproblems erarbeitet werden soll, wird als Eingabeparameter ein niedriges p basierend auf Germanedge's interner Heuristik gewählt. Dennoch kann auch damit teilweise die Optimalität einer Lösung nachgewiesen werden. Zum Beispiel, wenn die Kodierung eines Auftrages mit p unerfüllbar und mit $p+1$ erfüllbar ist. Dann ist $p+1$ die minimale Pressen-Anzahl.

Symbolisch wird also nur der Gesamtrest optimiert (Optimierungsziel 3). Das geschieht entweder durch inkrementelles Verfeinern (SMT) mit der Implementierung aus Abschnitt 4.2.2 oder durch die linearen Optimierungsverfahren von Z3 (OMT). Um bei ersterer realistische Verbesserungen der Lösung zu gewährleisten, wird als Schrittweite ein Millimeter der Kodierung verwendet. Also 0.001 Einheiten (Meter) bei Kodierung in der Sorte `Real` und 1 Einheit (Millimeter) bei der Kodierung in der Sorte `Int`. Der Aufruf der Optimierungsfunktion aus Abschnitt 4.2.2 sieht somit folgenderweise aus:

```
1  totalWaste :: Num (Expr t) => PressPlan t -> Expr t
2  totalWaste = sumOf (layers.folded.waste)
3
4  step :: (KnownSMTSort t, Num (Expr t)) => Expr t -> Expr t
5  step x = case sortSing @t of
6    SIntSort  -> x - 1
7    SRealSort -> x - 0.001
8    _         -> x
9
10 main :: IO ()
11 main = do
12   disperseOnPresses $ do
13     ...
14     (res, sol) <- solveMinimized (totalWaste pressPlan) (Just step) Nothing
15     ...
```

Listing 19: Aufruf der Optimierungsfunktion `solveMinimized` für den Gesamtrest

5.6. Benutzerschnittstelle

Germanedge's bisherige heuristische Lösung spezifiziert Ein- und Ausgabeformat als CSV. Diese Spezifikation wird hier ebenfalls implementiert. Dazu wird die Haskell-Bibliothek `cassava` [Tib24] verwendet.

Der Aufruf des Programms erfolgt über die Kommandozeile mit wenigen Konfigurationsparametern. Konfiguriert werden können:

- Timeout: Maximale Laufzeit des Programms
- Art der Kodierung: `Int` mit `QF_LIA` (`OpenSMT`), `Real` mit `QF_LRA` (`OpenSMT`) oder `BitVec 15 (Unsigned)` mit `QF_BV` (`Bitwuzla`)

- Pressen-Spezifikation: Dimensionen der Presse
- Pressen-Anzahl p: Kodiert wird dann mit p, $p+1$, $p+2$ und $p+3$
- Constraints: Maximaler Gesamtrest, maximaler Rest pro Schicht. Hier könnten optimale Lösungen verloren gehen. Der Kunde wünscht sich diese Parameter dennoch.
- Ausgabe-Verzeichnis: Verzeichnis, in welches die Lösungen der verschiedenen Pressen ausgegeben werden

6. Auswertung und Laufzeitmessung

Was haben wir für Testfälle? Woher kommen die Testfälle? Wie wurden die Testfälle ausgewählt? Sind sie repräsentativ für reale Probleme? Vorgehen Laufzeitmessung erwähnen. Maschinen-Specs nennen. Vergleich von:

- OMT: Z3 QF_LIA
- OMT: Z3 QF_LRA
- SMT: OpenSMT QF_LIA
- SMT: OpenSMT QF_LRA
- SMT: Bitwuzla QF_BV mit 15 Bit

6.1. Vergleich verschiedener Kodierungen

6.2. Vergleich mit liberalem Timeout

Wer findet bessere Lösung, wenn Timeout sehr liberal (24h?)?

7. Zusammenfassung

Literatur

[Ans+11] Carlos Ansótegui u. a. „Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem“. In: Jan. 2011, S. 2–9. ISBN: 9781577355434.

[Bar+22] Haniel Barbosa u. a. „cvc5: A Versatile and Industrial-Strength SMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Dana Fisman und Grigore Rosu. Cham: Springer International Publishing, 2022, S. 415–442. ISBN: 978-3-030-99524-9.

[BBJ24] Martin Bromberger, François Bobot und Martin Jonáš. *SMT-COMP 2024 Results*. Abgerufen am: 18.09.2024. 2024. URL: https://smt-comp.github.io/2024/results/.

[BFT17] Clark Barrett, Parcal Fontaine und Cesare Tinelli. *The SMT-LIB Standard Version 2.6*. Techn. Ber. Abgerufen am: 21.08.2024. Department of Computer Science, The University of Iowa, 2017. URL: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

[Bjø+19] Nikolaj Bjørner u. a. „Programming Z3“. In: Springer Nature, Apr. 2019, S. 148–201.

[BMS05] Clark Barrett, Leonardo de Moura und Aaron Stump. „SMT-COMP: Satisfiability Modulo Theories Competition“. In: *Computer Aided Verification*. Hrsg. von Kousha Etessami und Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 20–23. ISBN: 978-3-540-31686-2.

[BPF15] Nikolaj S. Bjørner, Anh-Dung Phan und Lars Fleckenstein. „νZ - An Optimizing SMT Solver“. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 2015. URL: https://api.semanticscholar.org/CorpusID:17542710.

[BRA13] EDWIN BRADY. „Idris, a general-purpose dependently typed programming language: Design and implementation“. In: *Journal of Functional Programming* 23 (Sep. 2013). DOI: 10.1017/S095679681300018X.

[Bru+10] Roberto Bruttomesso u. a. „The OpenSMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Javier Esparza und Rupak Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 150–153. ISBN: 978-3-642-12002-2.

[Bru24a] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.3.2. Abgerufen am: 20.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib.

[Bru24b] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.4.0. Abgerufen am: 25.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.4.0/docs/src/Language.Hasmtlib.Type.Expr.html#Expr.

[Bru24c] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.4.0. Abgerufen am: 25.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.4.0/docs/src/Language.Hasmtlib.Type.SMTSort.html#SSMTSort.

[Bru24d] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.5.0. Abgerufen am: 26.08.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.5.0/docs/src/Language.Hasmtlib.Counting.html#count%27.

[BT18] Clark Barrett und Cesare Tinelli. „Satisfiability Modulo Theories“. In: *Handbook of Model Checking*. Hrsg. von Edmund M. Clarke u. a. Cham: Springer International Publishing, 2018, S. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[Che10] Jingchao Chen. „A New SAT Encoding of the At-Most-One Constraint“. In: 2010. URL: https://api.semanticscholar.org/CorpusID:15322159.

[DB08] Leonardo De Moura und Nikolaj Bjørner. „Z3: an efficient SMT solver“. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, S. 337–340. ISBN: 3540787992.

[DKG13] Olfa Dridi, Saoussen Krichen und Adel Guitouni. „Solving resource-constrained project scheduling problem by a genetic local search approach“. In: *2013 5th International Conference on Modeling, Simulation and Applied Optimization (ICMSAO)*. 2013, S. 1–5. DOI: 10.1109/ICMSAO.2013.6552544.

[Dut14] Bruno Dutertre. „Yices 2.2“. In: *Computer Aided Verification*. Hrsg. von Armin Biere und Roderick Bloem. Cham: Springer International Publishing, 2014, S. 737–744. ISBN: 978-3-319-08867-9.

[EW12] Richard A. Eisenberg und Stephanie Weirich. „Dependently typed programming with singletons“. In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: Association for Computing Machinery, 2012, S. 117–130. ISBN: 9781450315746. DOI: 10.1145/2364506.2364522. URL: https://doi.org/10.1145/2364506.2364522.

[GBT07] Yeting Ge, Clark Barrett und Cesare Tinelli. „Solving Quantified Verification Conditions Using Satisfiability Modulo Theories“. In: Apr. 2007, S. 167–182. ISBN: 978-3-540-73594-6. DOI: 10.1007/978-3-540-73595-3_12.

[Gil14] Andy Gill. „Domain-specific Languages and Code Synthesis Using Haskell: Looking at embedded DSLs“. In: *Queue* 12.4 (Apr. 2014), S. 30–43. ISSN: 1542-7730. DOI: 10.1145/2611429.2617811. URL: https://doi.org/10.1145/2611429.2617811.

[GM09] Yeting Ge und Leonardo de Moura. „Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories“. In: *Computer Aided Verification*. Hrsg. von Ahmed Bouajjani und Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 306–320. ISBN: 978-3-642-02658-4.

[Hud+07] Paul Hudak u. a. „A history of Haskell: Being lazy with class“. In: Juni 2007, S. 1–55. DOI: 10.1145/1238844.1238856.

[KGK19] Gergely Kovásznai, Krisztián Gajdár und Laura Kovács. „Portfolio SAT and SMT Solving of Cardinality Constraints in Sensor Network Optimization“. In: *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2019, S. 85–91. DOI: 10.1109/SYNASC49474.2019.00021.

[Kme24] Edward A. Kmett. *lens: Lenses, Folds and Traversals*. Version 5.3.2. Abgerufen am: 16.09.2024. 2024. URL: https://hackage.haskell.org/package/lens.

[KSV19] Petr Kučera, Petr Savický und Vojtěch Vorel. „A lower bound on CNF encodings of the at-most-one constraint“. In: *Theoretical Computer Science* 762 (2019), S. 51–73. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2018.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0304397518305681.

[Liu+14] Tianhai Liu u. a. „A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution“. In: *Hardware and Software: Verification and Testing*. Hrsg. von Eran Yahav. Cham: Springer International Publishing, 2014, S. 284–299. ISBN: 978-3-319-13338-6.

[LLC21] Daeyeal Lee, Bill Lin und Chung-Kuan Cheng. „SMT-Based Contention-Free Task Mapping and Scheduling on 2D/3D SMART NoC with Mixed Dimension-Order Routing“. In: *ACM Trans. Archit. Code Optim.* 19.1 (Dez. 2021). ISSN: 1544-3566. DOI: 10.1145/3487018. URL: https://doi.org/10.1145/3487018.

[Nor09] Ulf Norell. „Dependently Typed Programming in Agda“. In: Jan. 2009, S. 1–2. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5.

[NPZ24] Aina Niemetz, Mathias Preiner und Yoni Zohar. „Scalable Bit-Blasting with Abstractions“. In: *Computer Aided Verification*. Hrsg. von Arie Gurfinkel und Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, S. 178–200. ISBN: 978-3-031-65627-9.

[Pey+06] Simon Peyton Jones u. a. „Simple Unification-based Type Inference for GADTs“. In: *International Conference on Functional Programming (ICFP)*. ACM, 2006. DOI: 10.1145/1159803.1159811.

[SCK04] Sean Seefried, Manuel Chakravarty und Gabriele Keller. „Optimising Embedded DSLs Using Template Haskell“. In: *Generative Programming and Component Engineering*. Hrsg. von Gabor Karsai und Eelco Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 186–205. ISBN: 978-3-540-30175-2.

[TBT24] Nestan Tsiskaridze, Clark Barrett und Cesare Tinelli. „Generalized Optimization Modulo Theories“. In: *Automated Reasoning*. Hrsg. von Christoph Benzmüller, Marijn J.H. Heule und Renate A. Schmidt. Cham: Springer Nature Switzerland, 2024, S. 458–479. ISBN: 978-3-031-63498-7.

[Tib24] Johan Tibell. *cassava: A CSV parsing and encoding library*. Version 0.5.3.2. Abgerufen am: 16.09.2024. 2024. URL: https://hackage.haskell.org/package/cassava.

[VAB18] Saurabh Vaidya, Prashant Ambad und Santosh Bhosle. „Industry 4.0 – A Glimpse“. In: *Procedia Manufacturing* 20 (2018). 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA, S. 233–238. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.02.034. URL: https://www.sciencedirect.com/science/article/pii/S2351978918300672.

Appendix

A. Vergleich der Kodierung mit realistischem Timeout

Eingabe				Problem							Lösung				
Nr.	**Name**	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Kod.**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
1	Trivial	15	1	SMT	QF_LRA	OpenSMT					1		1	0	✓
				SMT	QF_LIA	OpenSMT	1	??	??	??	1	0	1	0	✓
				OMT	QF_LIA	Z3					??		??	??	??
2	Simple	20	2	SMT	QF_LRA	OpenSMT					18		1	0	✓
				SMT	QF_LIA	OpenSMT	1	??	??	??	10	0	1	0	✓
				OMT	QF_LIA	Z3					??		??	??	??
3	Simple2	40	2	SMT	QF_LRA	OpenSMT					49		1	0	✓
				SMT	QF_LIA	OpenSMT	1	??	??	??	50	0	1	0	✓
				OMT	QF_LIA	Z3					??		??	??	??
4	Small	27	6	SMT	QF_LRA	OpenSMT					TO		3	298	✗
				SMT	QF_LIA	OpenSMT	1	??	??	??	TO	0	3	298	✗
				OMT	QF_LIA	Z3					??		??	??	??
5	Optimal	41	6	SMT	QF_LRA	OpenSMT					2		1	0	✓
				SMT	QF_LIA	OpenSMT	1	??	??	??	1	0	1	0	✓
				OMT	QF_LIA	Z3					??		??	??	??
6	Optimal2	82	6	SMT	QF_LRA	OpenSMT					TO		2	58034	✗
				SMT	QF_LIA	OpenSMT	1	??	??	??	TO	0	2	6000	✗
				OMT	QF_LIA	Z3					??		??	??	??
7	2082510	39	8	SMT	QF_LRA	OpenSMT					TO		4	16114	✗
				SMT	QF_LIA	OpenSMT	2	??	??	??	TO	0	4	5359	✗
				OMT	QF_LIA	Z3					??		??	??	??
8	2082539	82	6	SMT	QF_LRA	OpenSMT					TO		2	13600	✗
				SMT	QF_LIA	OpenSMT	2	??	??	??	TO	0	2	14996	✗
				OMT	QF_LIA	Z3					??		??	??	??
9	2082518	103	10	SMT	QF_LRA	OpenSMT					TO		8	83362	✗
				SMT	QF_LIA	OpenSMT	7	??	??	??	TO	0	8	83363	✗
				OMT	QF_LIA	Z3					??		??	??	??
10	??	??	??	SMT	QF_LRA	OpenSMT					??		??	??	??
				SMT	QF_LIA	OpenSMT	??	??	??	??	??	??	??	??	??
				OMT	QF_LIA	Z3					??		??	??	??
11	??	??	??	SMT	QF_LRA	OpenSMT					??		??	??	??
				SMT	QF_LIA	OpenSMT	??	??	??	??	??	??	??	??	??
				OMT	QF_LIA	Z3					??		??	??	??

Danksagung

Zuallererst möchte ich mich bei Herrn Prof. Dr. Waldmann für das im Wintersemester 2023/2024, in der Vorlesung Constraint-Programmierung geweckte Interesse an jener bedanken. Vielen Dank auch für Ihre zahlreichen Hinweise und Vorschläge während dieser Bachelorarbeit. Weiter möchte ich meinen Dank der Germanedge Solutions GmbH und dort besonders Dr. Christoph Schumacher aussprechen, die diese Bachelorarbeit ermöglicht haben. Abschließend auch ein großes Dankeschön an Dr. Sebastian Möbius, welcher mir als Firmenbetreuer jederzeit alle Fragen bestens beantwortet hat.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Die den verwendeten Quellen und Hilfsmitteln wörtlich oder sinngemäß entnommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mir sind die strafrechtlichen Konsequenzen einer falschen eidesstattlichen Erklärung bekannt.

Leipzig, den 25. September 2024