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 6. Oktober 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 entwickelt, 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: < 10min) Lösung bekannter Testfälle. Da es sich hier um eine prototypische Lösung handelt, werden einige Sonderfälle des Pressenplanungsproblems in dieser Arbeit nicht betrachtet.

Die Ergebnisse dieser Arbeit zeigen, dass das Pressenplanungsproblem mithilfe von Constraint-Systemen optimal gelöst werden kann. Während das für Probleme kleiner bis mittlerer Größe innerhalb der gewünschten zehn Minuten möglich ist, bedarf die optimale Lösung größerer Probleme weitaus mehr Zeit (> 10h). Besonders für perfekte Pressenpläne ohne Verschnitt können schnell optimale Lösungen gefunden und zertifiziert werden.

[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 Holzlamellen zu Holzbalken verleimt, in Pressen gepresst und anschließend in Holzbauteile zersägt. Die Art und Weise der Anordnung der Bauteile in den Pressen wird als Pressenplan bezeichnet. Abbildung 1.1 visualisiert einen solchen Pressenplan.

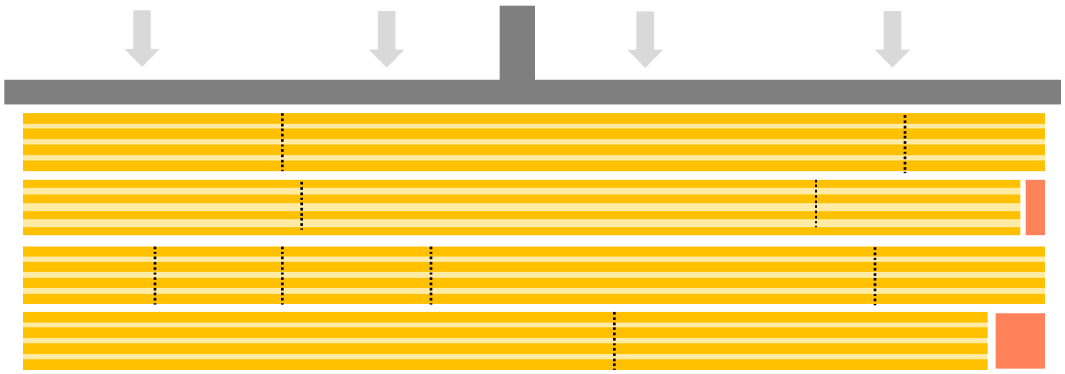

Abbildung 1.1.: Darstellung eines Pressenplans: Presse grau, Balken gelb, Rest rot und Sägelinien schwarz

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:

```haskell
barLength :: [Bar (Expr RealSort)] -> Expr BoolSort
barLength = all
              (\(b1,b2) -> (b1^.press === b2^.press) ==>
                (b1^.length === b2^.length)
              ) . 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	**Balken**	**Bauteil**	**Länge in mm**	**Höhe in mm**
1	1	1	13000	400
1	1	2	7500	400
1	2	1	3000	400
1	2	2	3000	400
1	2	3	3500	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:

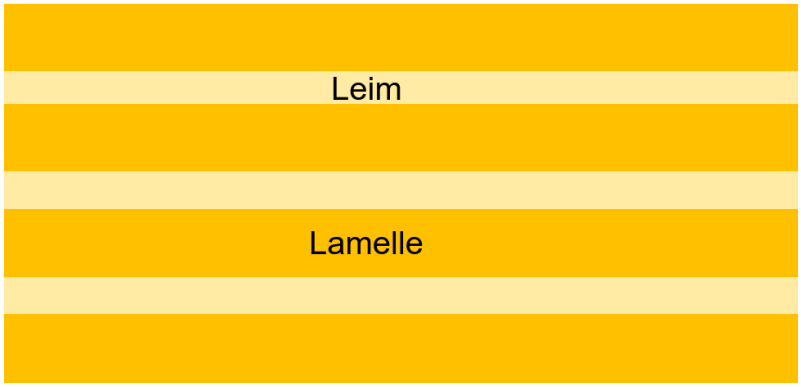

Abbildung 2.1.: Balken: Vertikale Verleimung von Lamellen

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:

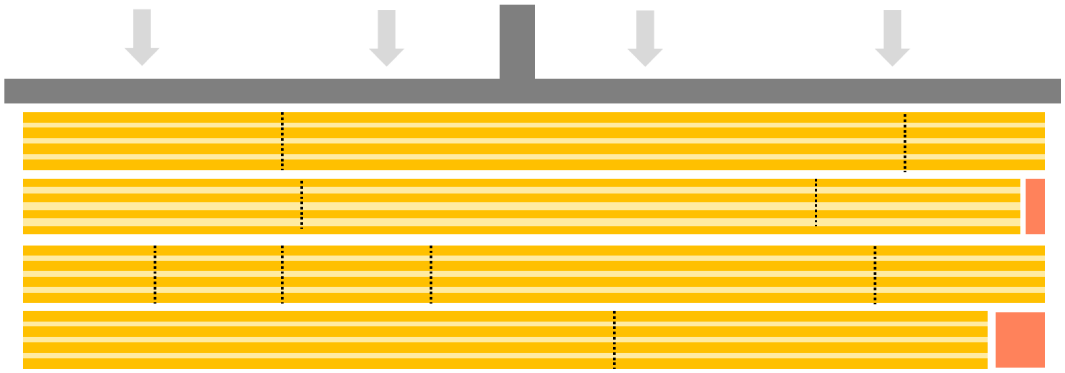

Abbildung 2.3.: Pressenplan für eine Presse mit vier Balken, 13 Bauteilen und zwei Reststücken

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	13000	400
1	1	2	7500	400
1	2	1	3000	400
1	2	2	3000	400
1	2	3	3500	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 auffüllender 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.

Damit ist das Pressenplanungsproblem ein Ressourcenzuordnungsproblem und somit NP-vollständig, also nicht in polynomieller Zeit lösbar [DKG13]. Das Problem wird allerdings noch schwieriger, da zusätzlich nach folgenden Kriterien lexikografisch optimiert wird:

1. Minimierung der Anzahl ungeplanter Bauteile
2. Minimierung der Anzahl geplanter Pressen
3. Minimierung des Gesamtrests (Summe aller Reste pro Balken)

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.

Betrachten wir für das Modell die Signatur $\Sigma = (\Sigma_F, \Sigma_R)$ und die Unbekannten $\mathbb{X} = (\mathbb{X}_{\mathbb{B}}, \mathbb{X}_{\mathbb{N}_0})$. Jede Presse hat sowohl Minimal- und Maximallänge als auch

Minimal- und Maximalhöhe. Weiter wird jeder Presse eine Menge an Schichten zugeordnet. Tatsächliche Länge und Höhe der Presse sind unbekannt:

$$\Sigma_F^P = \begin{Bmatrix} \text{minLength}: & P \to \mathbb{N}_0, \\ \text{maxLength}: & P \to \mathbb{N}_0, \\ \text{minHeight}: & P \to \mathbb{N}_0, \\ \text{maxHeight}: & P \to \mathbb{N}_0, \\ \text{layers}: & P \to 2^L \end{Bmatrix} \quad \mathbb{X}_{\mathbb{N}_0}^P = \begin{Bmatrix} \text{length}: & P \to \mathbb{N}_0, \\ \text{height}: & P \to \mathbb{N}_0 \end{Bmatrix} \qquad \mathbb{X}_{\mathbb{B}}^P = \emptyset$$

Für die Zuordnung der Schichten zu den Pressen wird eine disjunkte Aufteilung aller Schichten gefordert:

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

Schichten haben unbekannte Länge, Höhe, einen unbekannten Rest und eine boolesche Unbekannte, welche kennzeichnet, ob eine Schicht leer ist:

$$\Sigma_F^L = \emptyset \quad \mathbb{X}_{\mathbb{N}_0}^L = \begin{Bmatrix} \text{length}: & L \to \mathbb{N}_0, \\ \text{height}: & L \to \mathbb{N}_0, \\ \text{waste}: & L \to \mathbb{N}_0 \end{Bmatrix} \qquad \mathbb{X}_{\mathbb{B}}^L = \{\text{isEmpty} : L \to \mathbb{B}\}$$

Komponenten werden konkrete Längen und Höhen zugeordnet:

$$\Sigma_F^C = \begin{Bmatrix} \text{length}: & C \to \mathbb{N}_0, \\ \text{height}: & C \to \mathbb{N}_0 \end{Bmatrix} \quad \mathbb{X}_{\mathbb{N}_0}^C = \emptyset \qquad \mathbb{X}_{\mathbb{B}}^C = \emptyset$$

Zudem soll eine Relation die Zuordnung von Komponenten zu Schichten beschreiben:

$$\Sigma_R = \{\text{InLayer} \subseteq C \times L\}$$

Da diese allerdings unbekannt ist, wird die Relation durch eine unbekannte Funktion repräsentiert:

$$\mathbb{X}_{\mathbb{B}}^{\text{InLayer}} = \{\text{InLayer} : C \times L \to \mathbb{B}\}$$

Weiter wird eine Funktion benötigt, welche bestimmt, ob in einer Multimenge von Wahrheitswerten $B \in \dot{\mathbb{B}}$ höchstens $n \in \mathbb{N}_0$ wahr sind (mehr in Abschnitt 3.2):

$$\text{atMost} : \mathbb{N}_0 \times \dot{\mathbb{B}} \to \mathbb{B}$$

Insgesamt ergibt das also folgende Signatur:

$$\begin{aligned}
&\Sigma = (\Sigma_F, \Sigma_R) \text{ mit} \\
&\quad \Sigma_F = \Sigma_F^P \cup \Sigma_F^L \cup \Sigma_F^C \cup \left\{\text{atMost: } \mathbb{N} \times \dot{\mathbb{B}} \to \mathbb{B}\right\} \\
&\quad \Sigma_R = \emptyset \\
&\mathbb{X} = (\mathbb{X}_{\mathbb{B}}, \mathbb{X}_{\mathbb{N}_0}) \text{ mit} \\
&\quad \mathbb{X}_{\mathbb{B}} \ = \mathbb{X}_{\mathbb{B}}^P \cup \mathbb{X}_{\mathbb{B}}^L \cup \mathbb{X}_{\mathbb{B}}^C \cup \mathbb{X}_{\mathbb{B}}^{\text{InLayer}} \\
&\quad \mathbb{X}_{\mathbb{N}_0} = \mathbb{X}_{\mathbb{N}_0}^P \cup \mathbb{X}_{\mathbb{N}_0}^L \cup \mathbb{X}_{\mathbb{N}_0}^C
\end{aligned}$$

Die Anforderungen an einen Presseplan werden mit obiger Signatur sowie grundlegenden arithmetischen und booleschen Operationen folgend realisiert:

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

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

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

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

$$\forall c \in C : \text{atMost}\left(1, \biguplus_{l \in L} \dot{\{}\text{InLayer}(c, l)\dot{\}}\right) \tag{2.6}$$

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

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

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

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

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 repräsentiert Anforderung 5. Anforderung 6 ist in diesem Modell Konstruktionsvorschrift mit 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.

Das Modell beschreibt unbekannte Funktionen und nutzt damit Prädikatenlogik höherer Ordnung. Da die Mengen P, L und C endlich sind, ist auch der Definitionsbereich aller unbekannten Funktionen des Modells endlich. Daher kann das Modell auf Prädikatenlogik erster Ordnung reduziert werden, indem die unbekannten Funktionen durch unbekannte Werte realisiert werden. Folgend beispielhaft für Pressen:

$$\begin{aligned} &\mathbb{X}^{P\prime}_{\mathbb{N}_0} = \bigcup_{p \in P} \{\text{length}_p, \text{height}_p\} \\ &\text{length}(p) = \text{length}_p \\ &\text{height}(p) = \text{height}_p \\ &\Sigma^{P\prime}_F = \Sigma^P_F \cup \{\text{length}, \text{height}\} \end{aligned}$$

Wird anstelle der Quantifizierung im Modell außerdem grundinstanziiert, dann beträgt die Anzahl numerischer Unbekannter $|\mathbb{X}_{\mathbb{N}_0}| = 2|P| + 3|L|$ und die Anzahl boolescher Unbekannter $|\mathbb{X}_{\mathbb{B}}| = |L| + |C| \cdot |L|$. Die Anzahl Unbekannter steigt also linear in Abhängigkeit zur Eingabe. Das gilt offensichtlich auch für Anzahl und Größe der Constraints.

Im Folgenden der Arbeit werden Aufträge als Constraint-Systeme betrachtet. Dabei ist ein Auftrag $a \in A$ die Konjunktion aller Constraints für eine Eingabe: $a = 2.2 \wedge 2.3 \wedge \ldots \wedge 2.10$. Die Menge aller Modelle (hier erfüllende Belegung) für ein solches a wird dabei folgend notiert:

$$\mathrm{Mod}(a) = \left\{ (\mathcal{S}, (\beta_{\mathbb{N}_0}, \beta_{\mathbb{B}})) \;\middle|\; \begin{array}{l} \mathcal{S} = ((\mathbb{N}_0, \mathbb{B}), [\![\cdot]\!]_{(\mathbb{N}_0,\mathbb{B})}) \text{ ist } \Sigma\text{-Struktur}, \\ \beta_{\mathbb{N}_0} : \mathbb{X}_{\mathbb{N}_0} \to \mathbb{N}_0, \\ \beta_{\mathbb{B}} \;\; : \mathbb{X}_{\mathbb{B}} \;\; \to \mathbb{B}, \\ [\![a]\!]_{(\mathcal{S},(\beta_{\mathbb{N}_0},\beta_{\mathbb{B}}))} = \mathrm{True} \end{array} \right\}$$

Zur Vereinfachung der Notation können die Interpretationsfunktionen $\beta_{\mathbb{N}_0}$ und $\beta_{\mathbb{B}}$ zusätzlich mit einem Modell indiziert werden.

$$\forall a \in A,\ \forall s \in \mathrm{Mod}(a), s = (\mathcal{S}, (\beta_{\mathbb{N}_0}, \beta_{\mathbb{B}})) : \beta^s_{\mathbb{N}_0} = \beta_{\mathbb{N}_0} \wedge \beta^s_{\mathbb{B}} = \beta_{\mathbb{B}}$$

Weiter wird aus Notationsgründen die Menge aller erfüllbaren Aufträge definiert:

$$A_{sat} = \{a \in A \mid \mathrm{Mod}(a) \neq \emptyset\}$$

Außerdem werden folgende Funktionen zu Aufträgen definiert:

$\mathrm{presses} : A \to 2^P,\ \mathrm{presses}(a) = P_a,$ wobei P_a ist Menge aller Pressen in a
$\mathrm{layers} : A \to 2^L,\ \mathrm{layers}(a) = L_a,$ wobei L_a ist Menge aller Schichten in a
$\mathrm{components} : A \to 2^C,\ \mathrm{components}(a) = C_a,$ wobei C_a ist Menge aller Komponenten in a

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 Kunde Germanedge):

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

Ziel 1 ist die Minimierung der Anzahl unverplanter Komponenten. Dort ist das Optimum bestenfalls null, sodass alle Komponenten verplant sind. Existiert so eine Lösung immer? Also:

$$\forall a \in A_{sat},\ \exists s \in \mathrm{Mod}(a) : 0 = \left|\left\{c \in \mathrm{components}(a) \;\middle|\; \neg \bigvee_{l \in \mathrm{layers}(a)} \beta^s_{\mathbb{B}}(\mathrm{InLayer}(c, l))\right\}\right|$$

Nein, denn es lässt sich ein einfaches Gegenbeispiel konstruieren:

Seien $P = \{p_1, p_2\}$ und $C = \{c_1, c_2\}$ mit
$\text{height}(c_2) < \text{minHeight}(p_1) \leq \text{minHeight}(p_2) < \text{maxHeight}(p_2) < \text{height}(c_1) = \text{maxHeight}(p_1)$
Dann muss c_1 in p_1 und damit c_2 in p_2 verplant werden.
Das geht aber nicht, weil: $\text{height}(c_2) < \text{minHeight}(p_2)$

Wie Erfahrungsberichte und alle Testfälle in Kapitel 6 zeigen, existiert mit hoher Wahrscheinlichkeit aber eine Lösung, in welcher alle Bauteile verplant sind. Daher wird Constraint 2.6 durch folgendes Constraint ersetzt:

$$\forall c \in C,\ \exists! l \in L : \text{InLayer}(c, l) \tag{2.11}$$

Damit sind alle erfüllenden Belegungen des Modells bezüglich Optimierungsziel 1 optimal. In den wenigen Fällen, in welchen die minimale Anzahl unverplanter Bauteile größer als null ist, existiert in diesem Modell keine Lösung. Aufgrund des mit der sonst zusätzlich nach Ziel 1 notwendigen Optimierung einhergehenden signifikanten Performance-Verlusts bei der Lösung durch den SMT-Solver sowie der Seltenheit solcher Fälle, wird in diesem Modell das Nicht-Finden dieser Lösungen hingenommen. Aufträge $a \in A$ werden damit neu definiert:

$$a = 2.2 \wedge 2.3 \wedge 2.4 \wedge 2.5 \wedge 2.7 \wedge 2.8 \wedge 2.9 \wedge 2.10 \wedge 2.11$$

Wird im Weiteren der Arbeit auf das Modell oder Constraint-System verwiesen, dann ist diese Definition eines Auftrages gemeint.

3. Satisfiability Modulo Theories

In diesem Kapitel wird zuerst das Erfüllbarkeitsproblem der Satisfiability Module Theories - kurz SMT - vorgestellt. Dabei wird auch kurz auf 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 der Einführung in das Erfüllbarkeitsproblem SMT 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 komplexer Probleme wie beispielsweise Ressourcenzuordnungsproblemen [Ans+11].

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 SMT-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 eine erfüllende Belegung zu erstellen. Zeile zwei setzt die Logik für die darauffolgende Problemdeklaration. In den Zeilen 4 und 5 werden Unbekannte (nullstellige Funktionen) 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 Multimenge gegebener boolescher Variablen $B \in \dot{\mathbb{B}}$ soll dabei die Anzahl wahr belegter Variablen nach k restriktiert werden. Dabei existieren die folgenden drei Restriktionen:

- Mindestens k: $\text{atLeast} : \mathbb{N}_0 \times \dot{\mathbb{B}} \to \mathbb{B}$
- Genau k: $\text{exactly} : \mathbb{N}_0 \times \dot{\mathbb{B}} \to \mathbb{B}$
- Höchstens k: $\text{atMost} : \mathbb{N}_0 \times \dot{\mathbb{B}} \to \mathbb{B}$

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 $\text{count} : \dot{\mathbb{B}} \to \mathbb{N}_0$ lassen sich die Anzahl-Constraints einfach kodieren:

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

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

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

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

Je nach gewählter Logik erfolgt die Summierung in der entsprechenden Sorte. Ist für das SMT-Problem beispielsweise die Logik `QF_LRA` gewählt, dann summiert $\text{count}(B)$ Terme der Sorte `Real`. Dabei entstehen für jedes Anzahl-Constraint offensichtlich $n = |B|$ 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:

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

Die Produkt-Kodierung (englisch *product-encoding*) ist eine aus SAT bekannte Kodierung für $\text{atMost}(1, B)$ [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 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 $\exists x \in \mathbb{Z}, \exists y \in \mathbb{Z} : x < 10 \land y < 5 \land (y < 7 \rightarrow x = 1)$:

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

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

Mithilfe von Verfahren der linearen Optimierung ermittelt der Solver (hier `Z3`) korrekt die optimale Lösung $x \coloneqq 1, y \coloneqq 4$.

4. Haskell und Hasmtlib

In diesem Kapitel werden einige, für das weitere Verständnis der Arbeit 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

Haskell ist eine funktionale Programmiersprache, welche sich vor allem durch statisch polymorphe Typisierung und Bedarfsauswertung auszeichnet [Mar+10]. Besonders das ausdrucksstarke Typsystem macht Haskell zu einem mächtigen Werkzeug in der Software-Entwicklung. Haskell unterscheidet dabei zwischen Typen zur Übersetzungszeit und Werten zur Laufzeit eines Programms [EW12]. Diese strenge Trennung wird als Phasentrennung bezeichnet. Betrachten wir folgenden algebraischen Datentypen (kurz ADT) einfach gelinkter Listen:

```
1 data List a = Cons a (List a) | Nil
2
3 exampleList :: List Int
4 exampleList = Const 5 $ Cons 3 $ Cons 10 Nil
5
6 append :: List a -> List a -> List a
7 append Nil         ys = ys
8 append (Cons x xs) ys = Cons x (append xs ys)
```

Listing 6: Algebratischer Datetyp einfach gelinkter Listen

Dieser Listen-Typ ist polymorph in den Elementen der Liste, sodass `a` mit verschiedenen Typen belegt werden kann. In Zeile 3 von Listing 6 wird das `a` zum Beispiel mit `Int` belegt. Damit ist `exampleList` ein Wert vom Typ `List Int`, also in Haskell-Syntax: `exampleList :: List Int`.

Diese *ist-Wert-von-Typ-Beziehung* kann auch auf die Typ-Ebene zur Übersetzungszeit gehoben werden. Die höhere Beziehung lautet dann: *ist-Typ-von-Art-Beziehung*. Somit ist `List Int` vom *Kind* (deutsch Art) `Type`, also in Haskell-Syntax: `List Int :: Type`. Ein *Kind* für einen Typen ist also das, was ein Typ für einen Wert ist.

Das gilt auch für Funktionen. Die Funktion `append` nimmt zwei Listen und produziert eine Liste, also `append :: List a -> List a -> List a`. Der polymorphe Typ `List a` ist auch eine Funktion - aber auf Typ-Ebene. Er nimmt einen Typen und produziert einen Typen, also `List a :: Type -> Type`. Die Wahl des `a` kann durch Typ-Applikationen zur Übersetzungszeit folgend gewählt werden:

```haskell
floatAppend :: List Float -> List Float -> List Float
floatAppend xs ys = append @Float xs ys
```

Listing 7: Typ-Applikation polymorpher Funktionen

Intern werden dazu Type-Operatoren (`~`) verwendet. In Listing 7 wird das `a` mit `Float` belegt, also gilt: `a ~ Float`.

Mit diesem Wissen können nun Klassen betrachtet werden. Abstraktion über Listing 6 liefert die Spezifikation von Halbgruppen:

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

instance Semigroup (List a) where
  (<>) xs ys = append xs ys
```

Listing 8: Typ-Klasse für Halbgruppen

Die Klasse `Semigroup` spezifiziert die Funktion `(<>)` und der Typ `List a` implementiert diese Spezifikation.

Bisher wurde die Implementierung von `append` in Listing 6 noch nicht betrachtet. Die Implementierung unterscheidet sich dabei je nach Konstruktor des ADTs `List a`. Das Aufbrechen der Konstruktoren wird auch *Pattern-Matching* genannt [Mar+10]. Eine Verallgemeinerung von ADTs stellen generalisierte algebraische Datentypen (kurz GADT) dar. Dort kann der Pattern-Match auf einen Wert (Konstruktor) zur Laufzeit zusätzlich Informationen zum Typen des Werts zur Übersetzungszeit liefern. Da dadurch Typen verfeinert werden können, wird diese Möglichkeit auch *Type-Refinement* genannt [Pey+06].

Die bisher betrachteten Funktionen sind alle pur - also mathematische Funktionen. Um zusätzlich Nebenwirkungen zu modellieren, nutzt Haskell das Konzept der Monade [Mar+10]. Da die Herleitung jener für diesen Rahmen zu komplex ist, reicht es für den Leser, zu verstehen, dass monadische Funktionen Berechnungen mit Kontext sind. Dieser Kontext ist bereits am Typ zu erkennen, sodass `appendPrint` beispielsweise den Typ `List a -> List a -> IO (List a)` hätte.

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 der eingebetteten Sprache erheblich reduziert [SCK04].

Haskell wird aufgrund seines starken Typsystems oft als Gastsprache für solche eingebetteten Sprachen verwendet [Gil14]. Listing 9 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 9: Repräsentation von SMTLib-Ausdrücken mithilfe eines GADTs in `Hasmtlib` [Bru24b]

Dabei ist der GADT `Expr t` polymorph in der `SMTSort`, sodass dessen Kind `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 10 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 10: Repräsentation des Singleton-Typs `SSMTSort` in `Hasmtlib` [Bru24c]

Die Verbindung zwischen Typ und Wert wird mit der in Listing 11 dargestellten Typ-Klasse `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 11: Typ-Klasse zur Verbindung von Typ `SMTSort` zu Wert `SSMTSort` in `Hasmtlib` [Bru24d]

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 12: Beispielhafte Verwendung von Hasmtlib

Das Beispiel in Listing 12 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 und ist in Anhang A unter Listing 22 zu finden.

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 bereits definierter Probleme und auch erlernter 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 erweitert. Deren Implementierung ist im Anhang A in Listing 23 zu finden. Wie dort zu erkennen ist, wurde außerdem eine Schrittweite für das Anpassen der verfeinerten Schranken hinzugefügt. Um nicht über die optimale Lösung zu schreiten, ruft Zeile 37 die Optimierungsschleife nochmals - aber ohne Schrittweite - auf. Die letzte bekannte optimale Schranke liegt dabei noch auf dem Incremental-Stack, sodass der rekursive Aufruf mit dieser als neue Schranke startet. Das garantiert das Finden der optimalen Lösung.

5. Implementierung

In diesem Kapitel wird ein Haskell-Programm zur Lösung des Pressenplanungsproblems entwickelt. Das Programm erzeugt aus den textuellen Eingaben der Pressen- und Bauteilspezifikationen ein Constraint-System, welches anschließend von einem externen Constraint-Solver gelöst wird. Danach wandelt das Programm die Lösung des Solvers in das von Germanedge spezifizierte Ausgabeformat CSV um.

Dazu werden zunächst die notwendigen Datentypen und die Sorten-Polymorphie der Kodierung erklärt, bevor einige konkrete Repräsentationen der Constraints aus Abschnitt 2.2 implementiert werden. Anschließend werden Optimierung und Parallelisierung des Problems diskutiert. Am Ende des Kapitels wird die Benutzerschnittstelle des Programms beschrieben.

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 13: 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 14: 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 `PressPlan` 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 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 unter diese Schichten auch einige leer sein können haben sie das Attribut `_layerEmpty :: b` (Listing 13, 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 durch den Solver 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 5.5.

5.2. Vorverarbeitung

Bei der Kodierung des Pressenplanungsproblems wird jeder Presse eine feste Anzahl an Schichten zugewiesen. Da die Anzahl tatsächlich geplanter Schichten bei der Zuweisung unbekannt ist, wird jeder Presse die höchstmögliche Anzahl an Schichten zugewiesen. Unter diesen können einige leer sein, in jenen sind also keine Komponenten (Constraints 2.7 und 2.8). Betrachten wir die Funktion $\text{maxLayers} : P \times A \to \mathbb{N}_0$ zur Berechnung der maximalen Schichtanzahl. Das triviale Maximum der Schichtanzahl pro Presse berechnet sich aus der kleinsten Komponente eines Auftrages und der Maximalhöhe der Presse:

$$\text{maxLayers}(p, a) = \left\lfloor \frac{\text{maxHeight}(p)}{\min \{\text{height}(c) \mid c \in \text{components}(a)\}} \right\rfloor$$

Diese Schranke ist korrekt, bezieht aber mögliche weitere Informationen des Auftrages nicht mit ein. Daher werden vermutlich meist mehr Schichten und damit auch Unbekannte und Formeln erzeugt, als für die Lösung des Problems überhaupt notwendig. Wenn die Schranke also niedriger gewählt werden kann, kann die Komplexität des Problems verringert und somit auch die Laufzeit des Solvers reduziert werden. Das ist möglich, indem zusätzlich die Anzahl der kleinsten Komponenten betrachtet wird:

Seien $h_1 \leq h_2 \leq \ldots \leq h_{|\text{components}(a)|}$ die sortierten Höhen aller Komponenten und $S_k = \sum_{i=1}^{k} h_i$ die kumulierte Summe dieser sortierten Höhen.

Dann gilt: $\text{maxLayers}(p, a) = \max \{k \mid S_k \leq \text{maxHeight}(p)\}$

Mithilfe von *Discrimination* [Hen08; Kme22] können die Höhen in $\mathcal{O}(|\text{components}(a)|)$ sortiert werden, sodass die Funktion `maxLayers` lineare Laufzeitkomplexität aufweist:

```haskell
import ...
import Data.Discrimination (Sorting(..), sort)

maxLayers :: (Num a, Sorting a) => StaticPress a -> [Component a] -> Integer
maxLayers p =
    genericLength
  . takeWhile (<= p^.pressMaxHeight)
  . drop 1
  . scanl (+) 0
  . sort
  . fmap (view compHeight)
```

Listing 15: Funktion zur Berechnung der maximalen Schichten pro Presse in $\mathcal{O}(|\text{components}(a)|)$

Betrachten wir die durch diese Funktion ermittelten Schranken für Aufträge a aus Kapitel 6, bei denen ein optimaler Pressenplan $s \models_{opt} a$ mit fester Presse p bekannt ist:

Auftrags-Nr.	maxLayers(p, a)	$\left\lvert\left\{l \in L \mid \beta_{\mathbb{B}}^{s_{opt}}(\neg\text{isEmpty}(l))\right\}\right\rvert$
1	3	3
2	11	10
3	11	11
5	11	7
6	11	7
14	4	4
15	4	4
16	4	4
18	4	4
20	4	4

Tabelle 5.1.: Vergleich der ermittelten Maximalanzahl an Schichten mit der tatsächlichen Anzahl in einer optimalen Lösung

Die ermittelte Schranke ist für einige der Testfälle perfekt, bei anderen wie 5 und 6 allerdings weit entfernt. Existiert eine bessere korrekte Schranke? Also:

$$\begin{aligned}&\exists f : P \times A \to \mathbb{N},\ \forall p \in P,\ \forall a \in A_{sat},\ \exists s \in \text{Mod}(a) : \\ &\quad s \models_{opt} a \to (|\{l \in \text{layers}(p) \mid \neg\beta_{\mathbb{B}}^{s}(\text{isEmpty}(l))\}| \leq f(p,a) \leq \text{maxLayers}(p,a) \\ &\qquad \wedge \exists p\prime \in P,\ \exists a\prime \in A_{sat} : f(p\prime, a\prime) < \text{maxLayers}(p\prime, a\prime))\end{aligned}$$

Da in dieser Arbeit lediglich ein Prototyp für die optimale Lösung des Pressenplanungsproblems entwickelt werden soll, genügt die Schranke der kumulierten Summen.

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`. `Hasmtlib`'s 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:

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

Listing 16: Sorten-Polymorphie bei der Problemdeklaration

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

```haskell
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 17: 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 aus 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 können. 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 bei der Pressenlänge vorkommen und dort gilt: $\forall p \in P : \text{maxLength}(p) < 2^{15}$. 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 zahlreicher arithmetischer Operationen vermutlich zu ineffizienten Kodierungen (Constraints 2.9 und 2.10). 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:

```haskell
pressBounds :: Orderable (Expr t) => PressPlan t -> Expr BoolSort
pressBounds = all
  (\p -> p^.pressHeight >=? p^.press.pressMinHeight
      && p^.pressHeight <=? p^.press.pressMaxHeight
      && p^.pressLength >=? p^.press.pressMinLength
      && p^.pressLength <=? p^.press.pressMaxLength
  ) . toListOf (presses.folded)
```

Listing 18: 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 das Constraint 2.11:

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

Listing 19: Haskell-Kodierung des Constraints 2.11

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.

```haskell
nonNegativeWaste :: (Orderable (Expr t), Num (Expr t))
  => PressPlan t -> Expr BoolSort
nonNegativeWaste = all (>=? 0) . toListOf (layers.folded.waste)
```

Listing 20: Haskell-Kodierung des extra Constraints: $\forall l \in L : \text{waste}(l) \geq 0$

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 20 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 fast immer

durch einen Pressenplan mit genau null übrigen Komponenten erfüllbar. Optimierungsziel 2 ist die Minimierung der Anzahl geplanter Pressen. Dabei gilt $1 \leq |P| \leq k$ mit erfahrungsgemäß $k \leq 11$ für die größten Aufträge. Wird ein Auftrag also k mal mit jeweils unterschiedlicher Pressen-Anzahl kodiert, dann existiert für genau eine jener Kodierungen mindestens eine optimale Belegung:

$$\forall a \in A_{sat},\ \exists! i \in \left\{ n \in \mathbb{N}^+ \mid n \leq k \right\},\ \exists s \in \mathrm{Mod}(a_{|P|=i}) : s \models_{opt} a$$

Optimal ist jene Belegung in entsprechender Kodierung genau dann, wenn der Gesamtrest unter der Belegung $\beta_{\mathbb{N}_0}$ minimal ist (Optimierungsziel 3):

Sei i die optimale Pressenanzahl für einen erfüllbaren Auftrag a, dann gilt:
$\forall a \in A_{sat},\ \exists s \in \mathrm{Mod}(a_{|P|=i}),\ \forall s\prime \in \mathrm{Mod}(a_{|P|=i}) :$

$$s \models_{opt} a \leftrightarrow \sum_{l \in \mathrm{layers}(a)} \beta^{s}_{\mathbb{N}_0}(\mathrm{waste}(l)) \leq \sum_{l \in \mathrm{layers}(a)} \beta^{s\prime}_{\mathbb{N}_0}(\mathrm{waste}(l))$$

Das kleinste Optimum des Gesamtrests liegt bestenfalls bei null. Durch das Hinzufügen des Constraints $0 = \sum_{l \in L} \mathrm{waste}(l)$ zur Kodierung mit der minimalen Pressen-Anzahl kann dieses bestmögliche Optimum forciert werden. Ist das Problem nach Hinzufügen des Constraints noch immer erfüllbar, dann ist jede erfüllende Belegung des Problems optimal.

$$\forall a \in A_{sat},\ a\prime = a \wedge 0 = \sum_{l \in \mathrm{layers}(a)} \mathrm{waste}(l),\ \forall s \in \mathrm{Mod}(a\prime) : s \models_{opt} a\prime$$

Ein solches zusätzliches Constraint schränkt den Lösungsraum des Problems erheblich ein, sodass die Lösung des Problems durch den Solver wesentlich effizienter wird.

Allerdings kann nicht garantiert werden, dass es eine Lösung ohne Rest gibt. Daher wird ein Auftrag $2k$ Mal kodiert: Auf k verschiedene Pressen-Anzahlen in jeweils zwei Variationen: Mit dem extra Constraint, welches den Gesamtrest auf null fixiert und ohne dieses. Diese $2k$ verschiedenen Kodierungen eines Auftrages werden bestenfalls auf $2k$ verschiedene Kerne einer CPU (oder mehrerer) verteilt. Die Maschine des Kunden, welcher Germanedge mit dem Pressenplanungsproblem beauftragt hat, verfügt über eine CPU mit acht Kernen. Daher wird ein Auftrag mit vier verschiedenen Pressen-Anzahlen kodiert, sodass insgesamt acht Kodierungen des Auftrages entstehen. Ein Eingabeparameter $p \in \mathbb{N}^+$ des Programms beschreibt 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, falls $p\prime < p \oplus p\prime > p+3$ gilt. Ähnlich wie bei der Ermittlung der maximalen Schichtanzahl pro Presse in Abschnitt 5.2 könnte auch hier ein (bestenfalls 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 für den Eingabeparameter p ein niedriger Wert durch Germanedge's interne Heuristik ermittelt.

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 signifikante 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:

```
totalWaste :: Num (Expr t) => PressPlan t -> Expr t
totalWaste = sumOf (layers.folded.waste)

step :: (KnownSMTSort t, Num (Expr t)) => Expr t -> Expr t
step x = case sortSing @t of
  SIntSort  -> x - 1
  SRealSort -> x - 0.001
  _         -> x

main :: IO ()
main = do
  disperseOnPresses $ do
    ...
    (res, sol) <- solveMinimized (totalWaste pressPlan) (Just step) Nothing
    ...
```

Listing 21: 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 einigen Konfigurationsparametern. Konfiguriert werden können:

- Art der Kodierung: SMT mit inkrementellem Verfeinern oder OMT mit linearer Optimierung
- Sorte der Kodierung: `Int` mit `QF_LIA`, `Real` mit `QF_LRA` oder `BitVec 15 (Unsigned)` mit `QF_BV`
- Solver zum Lösen: `Z3`, `OpenSMT`, ...
- Timeout: Maximale Laufzeit des Programms
- 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

Wenige Kombinationen wie beispielsweise Solver `OpenSMT` mit Art der Kodierung OMT sind ungültig und führen zu unmittelbaren Laufzeitfehlern, da nicht alle Solver alle Funktionalitäten unterstützen. Obwohl diese Fehler mit etwas Aufwand im Quell-Code verhindert werden könnten, kann der Anwender in der Schnittstelle hier keinen Fehler machen, weil die Evaluation in Kapitel 6 zu einer vermeintlich besten allgemeinen Konfiguration führt, welche später Standard-Konfiguration wird.

6. Auswertung und Laufzeitmessung

In diesem Kapitel wird die Performance des Programms zur Lösung des Pressenplanungsproblems ausgewertet. Dazu werden zunächst folgende Konfigurationen des Programms miteinander verglichen:

- SMT mit Incremental Refinement, `QF_LRA`, `OpenSMT`
- SMT mit Incremental Refinement, `QF_LIA`, `OpenSMT`
- SMT mit Incremental Refinement, `QF_LIA`, `Z3`
- OMT mit linearer Optimierung, `QF_LIA`, `Z3`

Die Wahl der Logiken `QF_LIA` und `QF_LRA` wurde bereits in Abschnitt 5.3 diskutiert. Für beide Logiken wird der jeweilige Gewinner jener Kategorien der SMT-COMP 2024 `OpenSMT` (Version 2.7.0) verwendet [BBJ24]. Zusätzlich wird auch `Z3` (Version 4.13.0) mit Incremental Refinement und Logik `QF_LIA` getestet. Trotz schwachem Ergebnis in der SMT-COMP 2024 erwies sich diese Kombination in einigen Testfällen als hoch-performant. Aufgrund dieser Erkenntnis wurde auch die Performance weiterer Solver untersucht. Diese Untersuchung ergab allerdings keine nennenswerten Ergebnisse, sodass diese im Folgenden ausgelassen werden. Weiter wird auch eine Programmkonfiguration mit `Z3` und OMT in Logik `QF_LIA` evaluiert.

Danach wird die Performance der insgesamt besten Programmkonfiguration mit jener von Germanedge's Heuristik verglichen und ausgewertet. Am Ende werden mögliche Verbesserungen diskutiert.

6.1. Vorbetrachtung

Die Laufzeitmessungen der Konfigurationen des in dieser Arbeit entwickelten Programms erfolgen auf einer Maschine mit acht CPU-Kernen mit Takt von jeweils 2,4 GHz und 16 GB Arbeitsspeicher. Aufgrund einiger Firmenregularien kann die Laufzeitmessung der Heuristik nicht auf der gleichen Maschine wie jener des hier entwickelten Programms erfolgen. Die firmeninterne Maschine weist allerdings sehr ähnliche Hardware-Spezifikationen auf, sodass ein Vergleich dennoch aussagekräftig ist. Die Laufzeitmessungen erfolgen unter ähnlichen Bedingungen mit minimaler Auslastung durch andere Prozesse.

Insgesamt wird die Lösung von 26 verschiedenen Probleminstanzen verglichen. Darunter befinden sich sowohl reale als auch konstruierte Testfälle. Zu Zwecken der Anonymisierung werden diese allerdings nicht als solche gekennzeichnet.

Ausgewertet wird die Güte der Lösungen nach den Optimierungszielen in Abschnitt 2.3. Zusätzlich kommt als viertes lexikografisches Optimierungsziel die Laufzeit des Programms beziehungsweise der Heuristik hinzu. Sind alle vier Metriken bei einer Probleminstanz gleich, dann wird die Lösung als gleich gut bewertet.

Die Laufzeitmessungen werden in den folgenden Abschnitten mit verschiedenen oberen Laufzeitgrenzen (Timeouts) versehen. Gemessen wird die Zeit, welche Programm oder Heuristik für eine endgültige Lösung benötigen. Ermittelt das Programm also beispielsweise eine Lösung nach $30s$, rechnet aber bis zum Timeout weiter und verbessert diese Lösung nicht, dann wird die Messung der Laufzeit mit TO (kurz für Timeout) versehen. Die gemessene Laufzeit des Programms schließt zwar alle Schritte von Eingabeverarbeitung über Kodierung der Constraints bis hin zur Lösung durch den Solver ein, umfasst letztendlich aber fast ausschließlich die Laufzeit des Solvers.

Folgend die Erklärung aller Kriterien der Vergleiche zur Lösung einer Probleminstanz:

Kriterium	**Erklärung**
Nr	Identifizierende Nummer der Probleminstanz
Name	Name der Probleminstanz
$\|C\|$	Anzahl der Komponenten
$\|C_{Spec}\|$	Anzahl verschiedener Komponenten-Spezifikationen
Typ	SMT (inkrementelles Verfeinern) oder OMT (lineare Optimierung)
Logik	`QF_LIA` oder `QF_LRA`
Solver	Externer SMT-Solver
p	Minimale Pressen-Anzahl (erklärt in Abschnitt 5.5)
Größe	Anzahl symbolischer Operationen für kleinste erfüllbare $\|P\|$, sonst p
$\#\mathbb{B}$	Anzahl boolescher Unbekannter im Constraint-System
$\#\mathbb{Z}$	Anzahl numerischer Unbekannter im Constraint-System
Zeit in s	Laufzeit in Sekunden (aufgerundet), TO ist Timeout
$\|C_{Not}\|$	Anzahl unverplanter Komponenten
$\|P\|$	Anzahl geplanter Pressen
Rest in mm	Gesamtrest in Millimetern
Zert.	✓ falls Lösung zertifiziert optimal ist, ✗ sonst

Tabelle 6.1.: Kriterien für die Auswertung

6.2. Vergleich verschiedener Programm-Konfigurationen

In diesem Abschnitt wird die Performance verschiedener Konfigurationen des in dieser Arbeit entwickelten Programms verglichen. Der Timeout für alle Probleminstanzen beträgt zehn Minuten, die maximal tolerierte Laufzeit aus Sicht des Kunden. Die tabellarische Auswertung der Laufzeitmessungen ist in Anhang B, Tabelle 1 zu finden.

Diese Ergebnisse zeigen, dass alle Programmkonfigurationen korrekte und auch optimale Lösungen für bestimmte Eingaben finden. Von insgesamt 26 Testfällen werden 25 gelöst. Zehn Testfälle können innerhalb von zehn Minuten sogar optimal gelöst und zertifiziert werden. Eine optimale Lösung ist genau dann zertifiziert, wenn der Solver nachweisen kann, dass keine bessere Lösung existiert.

Die Kodierung des Constraint-Systems in Logik `QF_LIA` liefert im Vergleich zu Logik `QF_LRA` meist die besseren Lösungen. Das ist wenig verwunderlich, da die kleinste Einheit in der Problemdefinition Millimeter ist, sodass die arithmetischen Unbekannten ganzzahlig dargestellt werden können. Dies wiederum vereinfacht die

Lösung für den Solver, da die Sorte `Int` die Menge $\mathbb{Z}$ (abzählbar unendlich) und die Sorte `Real` die Menge $\mathbb{R}$ (überabzählbar unendlich) repräsentiert, der Lösungsraum für Sorte `Int` also kleiner ist.

Erstaunlich ist die dominante Performance von `Z3` mit Kodierung in `QF_LIA` und Optimierung durch inkrementelles Verfeinern. In fast allen Testfällen liefert diese Konfiguration die beste Lösung. Besonders bemerkenswert sind dabei die Testfälle 2, 6, 18 und 20. Die Konfiguration liefert dort als einzige die zertifiziert optimalen Lösungen. Diese Ergebnisse stehen in Konflikt zu jenen der SMT-COMP 2024, in welcher `Z3` klar hinter `OpenSMT` einzuordnen ist [BBJ24]. Möglicherweise liegt das am niedrigen Verhältnis arithmetischer Unbekannter zu booleschen Unbekannten in der Modellierung des Pressenplanungsproblems. In den meisten Benchmarks der SMT-COMP 2024 für `QF_LIA` sind die Unbekannten fast ausschließlich von der Sorte `Int`.

Die Konfiguration mit der Optimierung nach linearen Verfahren in OMT, `Z3` und `QF_LIA` liefert für einfache Testfälle wie 1,2,3 oder 5 ebenfalls optimale Ergebnisse, für größere Testfälle allerdings gar keine. Das ist durch interne Beschränkungen von `Z3`'s Optimierungsmethodik und das Fehlen der Ausgabe von Zwischenergebnissen zu erklären.

Betrachten wir nun die beste Konfiguration mit inkrementellem Verfeinern, Kodierung in `QF_LIA` und Solver `Z3` etwas genauer. Das Finden und Zertifizieren der optimalen Lösung scheint für Probleminstanzen, bei welchen eine Lösung ohne Rest existiert am einfachsten. Testfall 20 zeigt, dass das sogar für größere Eingaben innerhalb von zehn Minuten möglich ist. Hingegen deuten die Testfälle 4, 13 und 17 darauf hin, dass der Nachweis von Optimalität selbst für kleine Testfälle, bei denen keine Lösung mit Rest null existiert, sehr schwer ist. Das ist so, weil der Solver erst nachweisen muss, dass keine andere Lösung existiert, die besser ist. Für den Fall mit Rest null entsteht ein einfacher Widerspruch zu Constraint 20 in Abschnitt 5.4. Bei allen anderen Fällen mit optimalem Rest w und $0 < k \leq w$ muss für alle möglichen w besseren Lösungen mit Rest k Unerfüllbarkeit nachgewiesen werden.

Die Lösung der größten Probleminstanzen 11, 12, 25 und 26 ist mit Betrachtung des Rests vermutlich weit vom Optimum entfernt. Unter allen Testfällen bleibt ein Testfall (12) innerhalb von zehn Minuten sogar gänzlich ungelöst. Testfälle wie dieser mit über 250 Bauteilen und mehr als acht Pressen stellen die größtmöglichen und damit schwierigsten Eingaben aus der Realität dar. Auch, wenn sowohl Anzahl Unbekannter und Formeln, als auch Größe letzterer linear zur Eingabe steigen (Abschnitt 2.2), scheint die Schwere des Problems mit erhöhter Komplexität des Constraint-Systems weitaus stärker zu wachsen. Große Probleme vergrößern nicht nur den Lösungsraum, sondern erhöhen auch die Anforderungen des Solvers an die Hardware. Laufzeitmessungen auf einer Maschine mit 128 GB Arbeitsspeicher und Timeout nach zehn Stunden (Anhang B, Tabelle 2) zeigen für Probleme 11, 12, 25 und 26 Arbeitsspeichernutzung von bis zu 40 GB pro Problem. Je länger der Solver-Prozess läuft, desto mehr Speicher benötigt er, weil er Lemmas lernt (CDCL(T), Abschnitt 3.1).

Während auch größere Probleme wie Testfall 24 ohne Rest innerhalb von zehn Stunden optimal gelöst werden können, kann selbst der kleine Testfall 4 nach zehn Stunden noch immer nicht optimal vom Solver gelöst werden. Testfälle 11 und 12 demonstrieren, dass die Solver mit mehr Laufzeit auch für die größten Probleme signifikant bessere Lösungen finden. Auch dort bestätigt sich die Programmkonfigu-

ration mit `Z3`, `QF_LIA` und inkrementellem Verfeinern als performanteste.

6.3. Vergleich mit Germanedge's Heuristik

Der Vergleich der Laufzeitmessungen für Germanedge's Heuristik und das hier entwickelte Programm ist in Anhang B Tabelle 3 zu finden. Das Programm findet innerhalb von zehn Minuten in 16 von 26 Testfällen die bessere Lösung (grün). In sieben von 26 Probleminstanzen siegt die Heuristik (rot). Bei den verbleibenden drei Testfällen finden Programm und Heuristik gleich gute Lösungen in gleicher Zeit.

Die Lösungen der Heuristik enthalten fast immer deutlich weniger Rest als die des Programms. Dennoch gewinnt das Programm knapp zwei Drittel der Testfälle, da die Minimierung des Rests lediglich Optimierungsziel 3 ist. Aufgrund der lexikografischen Bewertung der Güte der Lösungen und der Wahl der Heuristik, gegebenenfalls wenige Bauteile ungeplant zu lassen oder mehr Pressen zu nutzen, um einen geringeren Rest zu erzielen, kann das Programm in diesem Vergleich sehr gut überzeugen. Während das Programm beispielsweise trotz 110 Metern Rest nach der lexikografischen Bewertung in Testfall 11 besser abschneidet, würde der Kunde höchstwahrscheinlich die Lösung der Heuristik mit einem übrigen Bauteil und lediglich zwei Metern Rest bevorzugen. Das deutet darauf hin, dass der Kunde anstelle der lexikografischen Bewertung der Lösung die Wahl einer Kostenfunktion in Betracht ziehen sollte.

An der Implementierung des Programms ändert das nichts. Oberste Priorität stellt dann noch immer das Verplanen aller Komponenten dar. Da das Programm bereits Kodierungen für p, $p+1$, $p+2$ und $p+3$ erstellt, enthält höchstwahrscheinlich eine dieser die optimale Lösung nach der gewünschten Kostenfunktion.

Die Heuristik kann besonders bei großen Eingaben ($|C| > 130$) im Vergleich zum Programm überzeugen. Sie findet dort aufgrund einiger interner Vereinfachungen in geringer Zeit Lösungen nahe am Optimum. Optimale Lösungen können von der Heuristik zwar möglicherweise gefunden, aber nicht als optimal zertifiziert werden. Die Testfälle zeigen, dass das Programm das hingegen kann. Für kleine und mittlere Testfälle ($|C| \leq 130$) sogar innerhalb von zehn Minuten, wenn Lösungen ohne Rest existieren.

Daher sollte das hier entwickelte Programm für den Einsatz in der Industrie bei kleineren bis mittleren Testfällen in Betracht gezogen werden. Wenn die Restriktion an die Laufzeit liberaler gestaltet werden kann, also ein Timeout von zehn bis 24 Stunden möglich ist, dann kann das Programm selbst für größere Probleme Anwendung in der Industrie finden.

6.4. Diskussion

Das in dieser Arbeit entwickelte Programm berücksichtigt die Kernelemente des Pressenplanungsproblems. Die Heuristik implementiert bereits einige Sonderfälle wie Doppellagen in Pressen oder Pressen mit treppenartiger Abstufung. Diese wären auch für das Programm wünschenswert. Inwiefern die Eingliederung dieser in das Constraint-System sinnvoll ist, hängt davon ab, wie die Modellierung dieser Sonderfälle die Komplexität und damit Schwere des Constraint-Systems beeinflussen.

Die Testfälle in Anhang B Tabelle 1 zeigen, dass das Constraint-System überschaubar viele Unbekannte kodiert. Allerdings scheinen die Constraints groß und komplex. Wenige zusätzliche, bestenfalls boolesche Unbekannte und dafür weniger komplexe Constraints könnten die Solver-Laufzeit positiv beeinflussen. Außerdem verspricht eine mögliche Verbesserung der in Abschnitt 5.2 betrachteten Funktion zur Ermittlung der Höchstanzahl an Schichten pro Presse großes Potenzial.

7. Zusammenfassung

Pressenplanungsprobleme sind Ressourcenzuordnungsprobleme und diese sind NP-schwer. Germanedge nutzt zur Lösung dieser bereits eine Heuristik, welche durch einige Vereinfachungen die Komplexität des Problems reduziert und somit Lösungen nah am Optimum findet. Diese Vereinfachungen ermöglichen eine schnelle (< 10min), aber meist nicht optimale Lösung der Probleme. Daher wurde in dieser Arbeit ein neuer Ansatz untersucht.

Das in dieser Arbeit entwickelte prototypische Programm implementiert einen vollständigen Lösungsansatz mit Methoden der Constraint-Programmierung. Das Programm transformiert die Eingabe des Problems mithilfe der Haskell-Bibliothek `Hasmtlib` in ein in SMT-kodiertes Constraint-System, welches anschließend durch einen externen SMT-Solver gelöst wird. Dieser implementiert ein vollständiges Lösungsverfahren und garantiert somit die Optimalität der Lösung.

Ähnlich wie bei der Heuristik werden auch bei der Kodierung des Constraint-Systems Vereinfachungen vorgenommen. Im Gegensatz zu ersterer behalten diese allerdings optimale Lösungen bei. So werden beispielsweise Kodierungen mit verschiedenen aber festen Pressen-Anzahlen auf verschiedene CPU-Kerne aufgeteilt und parallel gelöst. Unter diesen wird anschließend die beste Lösung gewählt.

Haskell und Bibliothek `Hasmtlib` eignen sich dabei aufgrund derer Ausdrucksstärke hervorragend für die Transformation der Eingabe zum Constraint-System. Das Constraint-System wächst linear zur Größe der Eingabe. Besonders Probleme kleiner und mittlerer Größe können schnell und optimal auf durchschnittlichen Maschinen gelöst werden. Für das optimale Lösen größerer Probleme sind sowohl mehr Laufzeit als auch mehr Arbeitsspeicher notwendig. Die Lösung mit der Constraint-Programmierung schlägt die Heuristik in knapp zwei Dritteln aller Testfälle dieser Arbeit. Damit empfiehlt sich der neue Ansatz für die Anwendung in der Industrie bei kleinen und mittleren Pressenplanungsproblemen. Bei den größten Probleminstanzen und der harten Grenze von zehn Minuten bleibt die Heuristik die bessere Wahl.

Das hier entwickelte Programm ist lediglich ein Prototyp, sodass einige Sonderfälle und spezielle Anforderungen des Pressenplanungsproblems zukünftig noch abgedeckt werden sollten. Sowohl Modellierung als auch Implementierung des Problems im entwickelten Programm lassen dabei noch Spielraum für Verbesserungen übrig. Insgesamt zeigt diese Arbeit, dass die Constraint-Programmierung eine vielversprechende und vollständige Alternative zur Lösung von Ressourcenzuordnungsproblemen durch Heuristiken bietet.

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 2.7.0. Abgerufen am: 06.10.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.7.0/docs/src/Language.Hasmtlib.Type.Expr.html#Expr.

[Bru24c] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.7.0. Abgerufen am: 06.10.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.7.0/docs/src/Language.Hasmtlib.Type.SMTSort.html#SSMTSort.

[Bru24d] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.7.0. Abgerufen am: 06.10.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.7.0/docs/src/Language.Hasmtlib.Type.SMTSort.html#KnownSMTSort.

[Bru24e] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.7.1. Abgerufen am: 06.10.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.7.1/docs/src/Language.Hasmtlib.Counting.html.

[Bru24f] Julian Bruder. *hasmtlib: A monad for interfacing with external SMT solvers*. Version 2.7.1. Abgerufen am: 06.10.2024. 2024. URL: https://hackage.haskell.org/package/hasmtlib-2.7.1/docs/src/Language.Hasmtlib.Type.Solver.html#solveOptimized.

[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.

[Hen08] Fritz Henglein. „Generic discrimination: sorting and paritioning unshared data in linear time“. In: *SIGPLAN Not.* 43.9 (Sep. 2008), S. 91–102. ISSN: 0362-1340. DOI: 10.1145/1411203.1411220. URL: https://doi.org/10.1145/1411203.1411220.

[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.

[Kme22] Eward A. Kmett. *discrimination: Fast generic linear-time sorting, joins and container construction*. Version 0.5. Abgerufen am: 27.09.2024. 2022. URL: https://hackage.haskell.org/package/discrimination.

[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.

[Mar+10] Simon Marlow u. a. „Haskell 2010 language report“. In: (2010). URL: https://www.haskell.org/definition/haskell2010.pdf.

[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.

Glossar

ADT Algebraischer Datentyp: Zusammengesetzter Datentyp. 16

CDCL(T) Conflict-driven clause learning: Vollständiges Lösungsverfahren für SMT-Probleme bestimmter Logiken. 12

Constraint Allgemein eine Anforderung, hier im Rahmen SMT eine prädikatenlogische Formel. 1

Constraint-System Konjunktion von Constraints. 1

eDSL Embedded domain-specific language: Eingebettete domänenspezifische Sprache. 17

GADT Generalisierter Algebraischer Datentyp: Algebraischer Datentyp mit möglichem Type-Refinement beim Pattern-Match. 17

Incremental Refinement Schrittweises Anpassen einer Schranke mit der Suche nach einem Optimum. 20

Incremental-Stack Stapel, auf welchem in einem SMT-Solver ebenenweise Variablen angelegt oder Formeln assertiert werden können. Die oberste Ebene des Stapels kann jederzeit entfernt werden.. 20

ITE If-Then-Else Ausdruck: Pfadwahl nach Bedingung. 13

Kind Englisch für *Art*: Typ eines Typen. 16

OMT Optimization Modulo Theories: SMT mit zusätzlichen symbolischen Optimierungszielen. 15

SAT Satisfiability: Aussagenlogisches Erfüllbarkeitsproblem. 12

SAT-Solver Ausführbares Programm zum Lösen von SAT-Problemen. 12

Singleton Laufzeitzeuge für Übersetzungszeit-Typen. 18

SMT Satisfiability Module Theories: Erfüllbarkeitsproblem mit Arithmetik. i

SMT-Logik Kombination von Theorien. 12

SMT-COMP SMT-Competition: Wettbewerb für SMT-Solver in verschiedenen SMT-Logiken. 12

SMT-Solver Ausführbares Programm zum Lösen von SMT-Problemen. 12

SMTLib Sprachstandard zur Definition von SMT-Problemen. 1

Type-Refinement Verfeinern eines Typen: Ein allgemeinerer Typ kann durch einen spezifischeren repräsentiert werden. 17

Appendix

A. Erweiterungen von Hasmtlib

```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)

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 && atLeast @t 1 xs
exactly k xs = count xs === k

atLeast  :: forall t f. (Functor f, Foldable f, KnownSMTSort t
  , Num (HaskellType t), Ord (HaskellType t))
  => Expr t -> f (Expr BoolSort) -> Expr BoolSort
atLeast 0 = true
atLeast 1 = or
atLeast k = (>=? k) . count

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 = productEncoding
atMost k = (<=? k) . count

productEncoding :: (Foldable f, Boolean b) => f b -> b
productEncoding 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 productEncoding vs && productEncoding 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 22: Anzahl-Constraints in `Hasmtlib` [Bru24e]

```haskell
solveMinimized = solveOptimized (<?)
solveMaximized = solveOptimized (>?)

solveOptimized :: (MonadIncrSMT Pipe m, MonadIO m, KnownSMTSort t)
  => (Expr t -> Expr t -> Expr BoolSort) -- optimization comparison
  -> Expr t                              -- term to optimize
  -> Maybe (Expr t -> Expr t)            -- step-size-adjustment
  -> Maybe (Solution -> IO ())           -- access to intermediate results
  -> m (Result, Solution)
solveOptimized op goal mStep mDebug = refine Unsat mempty goal 0
  where
    refine oldRes oldSol target n_pushes = do
      res <- checkSat
      case res of
        Sat   -> do
          sol <- getModel
          case decode sol target of
            Nothing        -> return (Sat, mempty)
            Just targetSol -> do
              case mDebug of
                Nothing    -> pure ()
                Just debug -> liftIO $ debug sol
              push
              let step = fromMaybe id mStep
              assert $ target `op` step (encode targetSol)
              refine res sol target (n_pushes + 1)
        r -> do
          if n_pushes < 1
          then return (r, mempty)
          else case mStep of
            Nothing -> do
              replicateM_ n_pushes pop
              return (oldRes, oldSol)
            Just _ -> do
              pop
              -- make sure the very last step did not skip the optimal result
              opt <- solveOptimized op goal Nothing mDebug
              replicateM_ (n_pushes - 1) pop
              return opt
```

Listing 23: Incremental Refinement in `Hasmtlib` [Bru24f]

B. Laufzeitmessungen

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
1	15	1	SMT	QF_LRA	OpenSMT	1	552	48	11	1	0	1	0	✓
			SMT	QF_LIA	OpenSMT					1		1	0	✓
			SMT	QF_LIA	Z3					1		1	0	✓
			OMT	QF_LIA	Z3					1		1	0	✓
2	20	2	SMT	QF_LRA	OpenSMT	1	5089	231	35	18	0	1	0	✓
			SMT	QF_LIA	OpenSMT					10		1	0	✓
			SMT	QF_LIA	Z3					1		1	0	✓
			OMT	QF_LIA	Z3					1		1	0	✓
3	40	2	SMT	QF_LRA	OpenSMT	1	9969	451	35	49	0	1	0	✓
			SMT	QF_LIA	OpenSMT					50		1	0	✓
			SMT	QF_LIA	Z3					3		1	0	✓
			OMT	QF_LIA	Z3					8		1	0	✓
4	27	6	SMT	QF_LRA	OpenSMT	1	7536	336	42	TO	0	3	298	✗
			SMT	QF_LIA	OpenSMT					TO		3	298	✗
			SMT	QF_LIA	Z3					TO		3	298	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
5	41	6	SMT	QF_LRA	OpenSMT	1	10213	462	35	2	0	1	0	✓
			SMT	QF_LIA	OpenSMT					1		1	0	✓
			SMT	QF_LIA	Z3					1		1	0	✓
			OMT	QF_LIA	Z3					1		1	0	✓
6	82	6	SMT	QF_LRA	OpenSMT	1	50939	1826	70	TO	0	2	58034	✗
			SMT	QF_LIA	OpenSMT					TO		2	6000	✗
			SMT	QF_LIA	Z3					351		2	0	✓
			OMT	QF_LIA	Z3					TO		-	-	✗
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
7	123	6	SMT	QF_LRA	OpenSMT	2	130902	4092	105	TO	0	3	87150	✗
			SMT	QF_LIA	OpenSMT					TO		3	87150	✗
			SMT	QF_LIA	Z3					TO		3	60006	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
8	39	8	SMT	QF_LRA	OpenSMT	2	15619	640	56	TO	0	4	16114	✗
			SMT	QF_LIA	OpenSMT					TO		4	5359	✗
			SMT	QF_LIA	Z3					TO		4	5359	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
9	82	6	SMT	QF_LRA	OpenSMT	2	13097	624	28	TO	0	2	13600	✗
			SMT	QF_LIA	OpenSMT					TO		2	14996	✗
			SMT	QF_LIA	Z3					TO		2	11807	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
10	103	10	SMT	QF_LRA	OpenSMT	7	106143	3328	112	TO	0	8	83362	✗
			SMT	QF_LIA	OpenSMT					TO		8	83363	✗
			SMT	QF_LIA	Z3					TO		8	17228	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
11	193	2	SMT	QF_LRA	OpenSMT	5	400967	10476	174	TO	0	7	215950	✗
			SMT	QF_LIA	OpenSMT					TO		7	215950	✗
			SMT	QF_LIA	Z3					TO		6	110703	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
12	264	17	SMT	QF_LRA	OpenSMT	8	741951	18240	208	TO	-	-	-	✗
			SMT	QF_LIA	OpenSMT					TO		-	-	✗
			SMT	QF_LIA	Z3					TO		-	-	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
13	25	25	SMT	QF_LRA	OpenSMT	1	1376	104	14	TO	0	1	500	✗
			SMT	QF_LIA	OpenSMT					TO		1	500	✗
			SMT	QF_LIA	Z3					TO		1	500	✗
			OMT	QF_LIA	Z3					TO		2	0	✗
14	26	26	SMT	QF_LRA	OpenSMT	1	1428	108	14	1	0	1	0	✓
			SMT	QF_LIA	OpenSMT					1		1	0	✓
			SMT	QF_LIA	Z3					1		1	0	✓
			OMT	QF_LIA	Z3					1		1	0	✓
15	50	25	SMT	QF_LRA	OpenSMT	1	8561	408	28	9	0	2	0	✓
			SMT	QF_LIA	OpenSMT					10		2	0	✓
			SMT	QF_LIA	Z3					31		2	0	✓
			OMT	QF_LIA	Z3					5		2	0	✓
16	52	26	SMT	QF_LRA	OpenSMT	1	8897	424	28	8	0	2	0	✓
			SMT	QF_LIA	OpenSMT					8		2	0	✓
			SMT	QF_LIA	Z3					4		2	0	✓
			OMT	QF_LIA	Z3					2		2	0	✓
17	75	25	SMT	QF_LRA	OpenSMT	2	20496	912	42	TO	0	3	3300	✗
			SMT	QF_LIA	OpenSMT					TO		3	3300	✗
			SMT	QF_LIA	Z3					TO		3	500	✗
			OMT	QF_LIA	Z3					50		4	0	✗
18	78	26	SMT	QF_LRA	OpenSMT	2	21306	948	42	TO	0	4	51300	✗
			SMT	QF_LIA	OpenSMT					TO		4	51300	✗
			SMT	QF_LIA	Z3					47		3	0	✓
			OMT	QF_LIA	Z3					TO		4	0	✗
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
19	100	25	SMT	QF_LRA	OpenSMT	3	39531	1616	56	TO	0	5	23800	✗
			SMT	QF_LIA	OpenSMT					TO		5	28200	✗
			SMT	QF_LIA	Z3					TO		4	3208	✗
			OMT	QF_LIA	Z3					TO		5	0	✗
20	104	26	SMT	QF_LRA	OpenSMT	3	41099	1680	56	TO	0	5	32950	✗
			SMT	QF_LIA	OpenSMT					TO		5	51400	✗
			SMT	QF_LIA	Z3					575		4	0	✓
			OMT	QF_LIA	Z3					TO		5	0	✗
21	125	25	SMT	QF_LRA	OpenSMT	4	66916	2520	70	TO	0	5	4900	✗
			SMT	QF_LIA	OpenSMT					TO		5	4900	✗
			SMT	QF_LIA	Z3					TO		5	2516	✗
			OMT	QF_LIA	Z3					TO		6	0	✗
22	130	26	SMT	QF_LRA	OpenSMT	4	88901	3144	84	TO	0	6	40800	✗
			SMT	QF_LIA	OpenSMT					TO		6	40800	✗
			SMT	QF_LIA	Z3					TO		6	23458	✗
			OMT	QF_LIA	Z3					TO		6	0	✗
23	175	25	SMT	QF_LRA	OpenSMT	6	148636	4928	98	TO	0	8	56000	✗
			SMT	QF_LIA	OpenSMT					TO		8	56000	✗
			SMT	QF_LIA	Z3					TO		7	7700	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
24	182	26	SMT	QF_LRA	OpenSMT	6	187039	5856	112	TO	0	8	48950	✗
			SMT	QF_LIA	OpenSMT					TO		8	48950	✗
			SMT	QF_LIA	Z3					TO		8	47300	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
25	250	25	SMT	QF_LRA	OpenSMT	9	396926	11044	154	TO	0	-	-	✗
			SMT	QF_LIA	OpenSMT					TO		-	-	✗
			SMT	QF_LIA	Z3					TO		11	52858	✗
			OMT	QF_LIA	Z3					TO		-	-	✗
26	260	26	SMT	QF_LRA	OpenSMT	9	412766	11484	154	TO	0	11	55800	✗
			SMT	QF_LIA	OpenSMT					TO		11	55800	✗
			SMT	QF_LIA	Z3					TO		11	75211	✗
			OMT	QF_LIA	Z3					TO		-	-	✗

Tabelle 1.: Vergleich der Programm-Konfigurationen mit Timeout nach zehn Minuten

Eingabe			Problem							Lösung				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Typ**	**Logik**	**Solver**	p	**Größe**	$\#\mathbb{B}$	$\#\mathbb{Z}$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
4	27	6	SMT	QF_LIA	Z3	1	7536	336	42	TO	0	3	298	✗
			OMT							TO		-	-	✗
7	123	6	SMT	QF_LIA	Z3	2	130902	4092	105	20571	0	3	0	✓
			OMT							14994		3	0	✓
11	193	2	SMT	QF_LIA	Z3	6	400967	10476	174	TO	0	6	63883	✗
			OMT							TO		-	-	✗
12	264	17	SMT	QF_LIA	Z3	8	876160	20520	234	TO	0	9	205437	✗
			OMT							TO		-	-	✗
22	130	26	SMT	QF_LIA	Z3	4	69576	2620	70	632	0	5	0	✓
			OMT							9788		5	0	✓
24	182	26	SMT	QF_LIA	Z3	6	154558	5124	98	35806	0	7	0	✓
			OMT							30730		7	0	✓

Tabelle 2.: Vergleich der Programm-Konfigurationen mit Timeout nach zehn Stunden

Eingabe			Heuristik				Constraint-System				
Nr.	$\|C\|$	$\|C_{Spec}\|$	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zeit in s**	$\|C_{Not}\|$	$\|P\|$	**Rest in mm**	**Zert.**
1	15	1	1	0	1	0	1	0	1	0	✓
2	20	2	1	0	1	0	1	0	1	0	✓
3	40	2	1	0	1	0	3	0	1	0	✓
4	27	6	23	1	3	378	TO	0	3	298	✗
5	41	6	1	0	1	0	1	0	1	0	✓
6	82	6	143	1	3	0	351	0	2	0	✓
7	123	6	233	0	5	0	TO	0	3	60006	✗
8	39	8	28	3	4	1016	TO	0	4	5359	✗
9	82	6	TO	2	2	0	TO	0	2	11807	✗
10	103	10	TO	1	9	2434	TO	0	8	17228	✗
11	193	2	TO	1	7	2000	TO	0	6	110703	✗
12	264	17	7	0	9	11331	TO	-	-	-	✗
13	25	25	45	0	1	500	TO	0	1	500	✗
14	26	26	12	0	1	0	1	0	1	0	✓
15	50	25	78	0	3	0	31	0	2	0	✓
16	52	26	214	0	3	750	4	0	2	0	✓
17	75	25	45	0	3	500	TO	0	3	500	✗
18	78	26	36	0	4	0	47	0	3	0	✓
19	100	25	55	0	5	500	TO	0	4	3208	✗
20	104	26	108	0	5	0	575	0	4	0	✓
21	125	25	98	0	5	500	TO	0	5	2516	✗
22	130	26	48	0	6	0	TO	0	6	23458	✗
23	175	25	TO	1	7	0	TO	0	7	7700	✗
24	182	26	140	0	8	750	TO	0	8	47300	✗
25	250	25	296	0	12	750	TO	0	11	52858	✗
26	260	26	TO	1	11	0	TO	0	11	75211	✗

Tabelle 3.: Vergleich der Performance von Programm-Konfiguration SMT, Z3, `QF_LIA` und Germanedge's Heuristik mit TO 10 Minuten

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 6. Oktober 2024