Heinrich Heine
Universität
Düsseldorf

Institut für Informatik
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Ein Schach-Interpreter in Rosette

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Robin Wroblewski

Beginn der Arbeit: 10. Juli 2023
Abgabe der Arbeit: 30. September 2023

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Prof. Dr. Gunnar W. Klau

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30. September 2023

Robin Wroblewski

Zusammenfassung

Die Aufgabe dieser Bachelorarbeit war es einen Schach-Interpreter in Rosette als Fallstudie zu entwickeln. Dadurch sollten Erkenntnisse über die Umsetzung von solverunterstützten Werkzeugen für nichtdeterministische Probleme gewonnen werden. Der Autor erhofft sich dadurch Rückschlüsse auf ein übergeordnetes Ziel, nämlich die Nutzung von Rosettes solverunterstützten Werkzeugen für B-Maschinen, ziehen zu können.

Die geplante Software konnte mit Hilfe der vorgestellten Vorgehensweise erfolgreich umgesetzt werden. Weiter war es möglich vom Solver Lösungen für verschiedene Anfragen zu erhalten, die im Kontext Schach interessant sind. Es wurde jedoch auch festgestellt, dass Programme dieser Art zu Performanceproblemen neigen. Dafür werden einige Lösungsansätze und relevante Forschung besprochen.

Es konnte gezeigt werden, dass Probleme dieser Art mit Rosette gelöst werden können und sich weitere Projekte in diesem Bereich lohnen könnten. Die Ergebnisse dieser Arbeit eignen sich dafür als Grundlage und als Wegweiser, um die besprochenen Fallen zu umgehen.

Inhaltsverzeichnis

1 Einleitung

Rosette [Tor23b] ist eine *solver-aided programming language* (solverunterstützte Programmiersprache) auf der Basis von Racket [Rac23]. Bei solverunterstützten Sprachen und Werkzeugen geht es darum bestimmte Programmieraufgaben aus einer Vielzahl von Domains zu automatisieren. Im Wesentlichen besteht Rosette aus einer Teilmenge des Racket Kerns ergänzt um einen *symbolic compiler* (symbolischen Compiler) und Konstrukte, mit denen Queries abgesetzt werden. Dieser Compiler kann ein Programm in logische Constraints übersetzen, die der integrierte SMT-Solver [DMB08] versteht und lösen kann. Das Ergebnis wird in das Programm zurückgeführt, sodass es beispielsweise für weitere Modifikationen oder zur Formulierung eines Ausdrucks verwendet werden kann.

Die Sprache wurde als Host-Sprache für solverunterstützte DSLs entworfen [TB13]. Dies kennzeichnet sich bereits durch die Entscheidung Racket als Grundlage zu wählen. Dadurch erbt Rosette die Unterstützung von Metaprogrammierung und damit außerdem eine einfache Einbettung neuer Sprachen. Durch die Ergänzung des symbolischen Compilers können diese dann in solverunterstützte DSLs überführt werden. Entwickler, die ihre Sprache in Rosette implementieren, sind also davon befreit selbst einen symbolischen Compiler zu entwickeln. Dieser Prozess wäre sehr aufwendig, weshalb die Entwicklung von solverunterstützten Werkzeugen in der Vergangenheit viel Zeit in Anspruch genommen hat [TB14, BT18]. Durch Rosette erhalten Entwickler automatisch Zugang zu Konstrukten für Verifikation, Fehlerlokalisierung, Programmsynthese und Programmreparatur.

Die Motivation dieser Arbeit ist diese Konstrukte im Zusammenhang mit B-Maschinen [Abr96] nutzbar zu machen. Dies könnte beispielsweise in Form einer direkten Übersetzung von B-Maschinen in ein Rosette Programm oder der Entwicklung eines vollständigen B-Interpreters in Rosette realisiert werden. Ersteres würde Zugriff auf Werkzeuge für Verifizierung und Buglokalisierung liefern, während letzteres außerdem Programmsynthese ermöglicht. Der mit diesem Projekt verbundene Aufwand ist jedoch vergleichsweise hoch. Ziel dieser Arbeit ist es daher einen Zwischenschritt einzuführen, der gleichzeitig als Machbarkeitsprüfung dienen soll. Dafür soll ein Schach-Interpreter als Fallstudie entwickelt werden.

Rosettes solverunterstützte Konstrukte basieren auf vier grundlegenden Konzepten: *Assumptions* (Annahmen), *Assertions* (Behauptungen), *Symbolic Values* (symbolischen Werten) und *Queries* (Anfragen). Mit Assumptions und Assertions können Regeln ausgedrückt werden, denen jedes Programm folgen muss. Eine Annahme kann über die Eingabewerte des Programms getroffen werden. Erhält ein Programm zwei Werte vom Typ Integer als Argument, kann eine Assumption beispielsweise festlegen, dass diese in einem bestimmten Bereich liegen oder in einem bestimmten Verhältnis zueinander (größer, kleiner, gleich, ungleich) stehen. Mit Assertions lassen sich Eigenschaften angeben, die das Programm für alle durch die Annahmen zugelassenen Eingabewerte erfüllen muss. Wenn gegen eine solche Behauptung verstoßen wird, terminiert das Programm mit einer Fehlermeldung. Mit Hilfe des Solvers können Fragen über diese Eigeschaften beantwortet werden. Dabei kann es

beispielsweise darum gehen, ob eine Belegung der Eingabewerte existiert, die sowohl erlaubt ist als auch dazu führt, dass eine Behauptung verletzt wird. Diese Queries werden unter anderem durch symbolische Werte ausgedrückt. Ein symbolischer Wert steht, im Gegensatz zu einem konkreten Wert, für eine beliebige Belegung des entsprechenden Datentypen. Ein symbolischer Boolean kann also die konkreten Werte **true** und **false**, oder in Rosette ausgedrückt **#t** und **#f**, annehmen.

Grundsätzlich gibt es vier Queries, die den Großteil der Anwendungsfälle für Solver in der Programmierung abdecken [TB13]:

1. Verifizieren, dass ein Programm eine Spezifikation für einen gegebenen Eingaberaum erfüllt (**verify**);
2. Finden eines Satzes von Eingabewerten, die zur Erfüllung einer Spezifikation bei Programmausführung führen (**solve**);
3. Lokalisieren eines minimalen unerfüllbaren Kerns, der das Programm daran hindert ein gewünschtes Verhalten zu demonstrieren (**debug**);
4. Synthetisieren eines Programms mit gewünschtem Verhalten (**synthesize**).

Alle diese Queries werden von Rosette unterstützt. Sie können vereinzelt oder auch kombiniert im Entwicklungsprozess verwendet werden. Ein mögliches Szenario ist das Hinzufügen einer neuen Methode in eine bestehende solverunterstützte DSL. Eine **verify**-Query kann überprüfen, ob die Methode auf allen erlaubten Eingaben korrekte Ausgaben erzeugt. Ist dies nicht der Fall, können die problematischen Programmteile mit Hilfe einer **debug**-Query lokalisiert werden. Anschließend kann eine reparierte Variante dieser Teile mit der **synthesize**-Query gefunden und im Code eingesetzt werden.

2 Problemstellung

Die Aufgabe dieser Arbeit ist es einen Schach-Interpreter in Rosette zu entwickeln. Das bedeutet die Regeln des Spiels in Programmcode abzubilden. Insbesondere beinhaltet dies die Bewegungsmöglichkeiten von Figuren auf dem Spielbrett oder die verschiedenen spielentscheidenden Situationen (Schach, Schachmatt, Remis). Damit verleiht der Interpreter einem syntaktischen Programm in einer Schach DSL die Semantik. Angenommen ein Programm versuchte eine bestimmte Zugfolge auf dem Spielbrett durchzuführen. Nun könnte der Interpreter eine Aussage darüber treffen, ob diese Folge erlaubt ist. Kann ein solcher Interpreter erfolgreich implementiert werden, soll darauf basierend mit Queries für verschiedene Aufgaben experimentiert werden. Dabei könnte es beispielsweise darum gehen zu überprüfen, ob eine anfängliche Spielsituation mit einer legalen Zugfolge in eine gewünschte Spielsituation überführt werden kann.

Ein anderes Anwendungsgebiet sind sogenannte „Matt in drei Zügen“-Rätsel, bei denen es darum geht eine Zugfolge zu finden, die den gegnerischen Spieler innerhalb von drei Zügen in einen Schachmatt Zustand versetzt.

Der Fokus dieser Arbeit liegt dabei jedoch zu allererst auf der Erkundung der Programmiersprache Rosette und den dazugehörigen solverunterstützten Werkzeugen. Der Schach-Interpreter liefert dabei ein festgelegtes Ziel auf das hingearbeitet wird. Die Domain Schach wurde ausgewählt, da es sich dabei um ein nichtdeterministisches[1] Problem handelt. Diese Eigenschaft ist in Bezug auf die Motivation, die Werkzeuge im Zusammenhang mit B-Maschinen zu nutzen, interessant. Angenommen in einem Programm wird der Rückgabewert einer Methode als Eingabewert einer anderen Methode verwendet. Dadurch ergibt sich eine vergleichbare Situation wie beim Schach, da die Möglichkeiten und auch die verfolgten Pfade der Methoden jeweils voneinander abhängig sind. Diese Gemeinsamkeit erlaubt es gegebenenfalls Aussagen über die Machbarkeit des übergeordneten Ziels anhand der gewonnenen Erkenntnisse zu machen. Der erwartete Mehrwert dieser Arbeit wird das im Prozess erlangte Wissen sein. Das bedeutet, dass besonderer Fokus auf der Dokumentation des Entwicklungsprozesses liegt. Wichtige Punkte sind dabei die das Design betreffenden Entscheidungen, welche im Verlauf der Entwicklungsphase getroffen werden. Des Weiteren sollen potenziell auftretende Hindernisse und die unternommenen Lösungsversuche besprochen werden.

Der Umfang des Interpreters und die Aufgaben, die mit dem Solver automatisiert wurden, haben sich erst im Laufe der Entwicklungsphase herausgestellt. Die Ergebnisse werden im Folgenden erläutert. Dazu wird die Arbeit in Abschnitt 3 zunächst in den aktuellen Forschungsstand eingeordnet. In Abschnitt 4 wird der finale Zustand der Applikation vorgestellt. Dieser Teil ist in zwei Sektionen unterteilt. Die erste Sektion befasst sich mit dem Aufbau des Interpreters, während die zweite Sektion die darauf arbeitenden Queries erläutert. Abschnitt 5 behandelt einzelne Aspekte der Entwicklungsphase. Es wird also beschrieben, wie der finale Zustand der Applikation erreicht wurde. Dabei wird erläutert wie eine kleine Version der Applikation als Prototyp entwickelt wurde und welche Kenntnisse dadurch gewonnen wurden. Da ein Großteil dieses Prototypen bei der Entwicklung der vollständigen Software wiederverwendet wurde, befassen sich die anschließenden Sektionen mit anderen Aspekten. Die Vorgehensweise in dieser Phase war mit einem Interpreter in Racket zu beginnen, diesen nach Rosette zu konvertieren und anschließend schrittweise die solverunterstützte Funktionalität einzuführen. Die Beschreibung beginnt mit der Konvertierung des Racket Programms nach Rosette. Anschließend wird über den Umgang mit Performanceproblemen und die Ergänzung um weitere Queries, die im Prototypen noch nicht getestet wurden, berichtet. In Abschnitt 6 werden die Ergebnisse zusammengefasst und bewertet, ob Rosette als solverunterstützte Sprache für dieses Problem geeignet war. Außerdem wird auf die wichtigsten Erkenntnisse eingegangen und ein Ausblick auf mögliche zukünftige Forschung gegeben.

[1] Mit Nichtdeterminismus ist an dieser Stelle gemeint, dass der gegnerische Spieler in den meisten Situationen zwischen einer Vielzahl möglicher Züge entscheiden kann und damit die Zugabfolge nicht vorherbestimmt ist. Im Vergleich zum Begriff aus der Spieltheorie, in der Determinismus die Abwesenheit von Zufallselementen beschreibt.

3 Verwandte Arbeiten

Entscheidend für dieses Projekt sind die Arbeiten und Publikationen von Emina Torlak, der Entwicklerin von Rosette. Torlak & Bodik [TB13] stellen Rosette als ein Framework für das Entwickeln von solverunterstützten Sprachen vor. Anhand der Prämisse einer Sprache für Boolesche Schaltkreise zeigen sie wie eine solverunterstützte DSL schrittweise entwickelt werden kann. Dafür wird zunächst von einem Racket Programm ausgegangen mit dem sich Schaltkreise definieren lassen. Dieses Programm wird in ein Rosette Programm umgewandelt. Anschließend wird gezeigt wie die Äquivalenz zweier Schaltkreise für alle möglichen Eingabewerte überprüft werden kann, indem symbolische Werte als Eingabe für die Schaltkreise verwendet werden. Außerdem werden weitere Queries und deren Nutzen im Kontext des Beispiels vorgestellt.

Diese Vorgehensweise wird für verschiedene Ebenen beschrieben. Die erste Ebene ist die beschriebene, bei der es um das Korrigieren von konkreten Schaltkreisen geht. Auf der zweiten Ebene wird eine Methode korrigiert, die einen gegebenen Schaltkreis in einen anderen, dazu äquivalenten Schaltkreis transformieren soll. Die dritte Ebene geht einen Schritt weiter und synthetisiert eine korrekte Implementierung einer allgemeinen Methode zur Transformation von Schaltkreisen in äquivalente Schaltkreise. Der Ansatz, der in diesem Beispiel beschrieben ist, nämlich vorerst einen Interpreter in Racket zu entwickeln und diesen dann schrittweise in einen solverunterstützten Interpreter zu überführen, wird in dieser Arbeit auch verfolgt. Rosette ist so in Racket integriert, dass sich Rosette Programme ohne symbolische Werte wie Racket Programme verhalten. Dies ermöglicht die beschriebene Vorgehensweise.

Außerdem Teil des Papers sind drei weitere Fallstudien von solverunterstützten Systemen, die in Rosette entwickelt wurden. Diese sind:

- WebSynth, ein Werkzeug für *web scraping*, also dem Extrahieren von Daten aus HTML Dateien. Diese Funktionalität wird erreichtm, indem *ZPath* Ausdrücke für Webseiten synthetisiert werden. Die ZPath Sprache basiert auf XPath [XPa17] und kann verwendet werden um anzugeben, wie die Daten aus der HTML Baumstruktur gewonnen werden.
- Ein Partitionierer für GA144 Programme. GA144 ist eine Architektur, die aus 144 Prozessoren mit jeweils sehr geringem Speicher besteht. Programme, die auf dieser Architektur laufen, müssen durch den geringen Speicher sehr fein aufgetrennt werden. So werden selbst Basisoperationen auf Datentypen in unterschiedlichen Kernen ausgeführt.
- Ein Superoptimierer für Bitvektor Programme. Dieser kann aus einem Satz von low-level Instruktionen ein zum übergebenen Programm äquivalentes Programm synthetisieren.

Pinckney et al. [PCG+23] entwarfen mit *PacSolve* ein Framework zur Generierung von Dependency Tools. Mit diesem waren sie in der Lage ein Ersatzwerkzeug für NPM [NPM23] zu erstellen, das verschiedene Schwachstellen von NPM aufgreift und verbessert. PacSolve wurde als solverunterstützte DSL in Rosette realisiert. Auf der Rosette Webseite [Tor23b] werden weitere Applikationen aus unterschiedlichen Domains vorgestellt. Dies umfasst beispielsweise die Themen Compiler [NVGTW20], Speichermodelle [BT17] und Datenbanken [CLW+17] oder auch fachfremde Bereiche, wie Medizin [PLT+16].

Butler et al. [BTP17] entwickelten ein System mit dem Strategien für das Lösen von Nonogrammen gefunden werden können. Nonogramme sind logische Puzzle, bei denen jeder Zeile und jeder Spalte eines Gitters eine oder mehrere Zahlen zugewiesen werden. Das Ziel ist es, die Kästchen dieser Zeilen und Spalten so einzufärben, dass die farbigen Kästchen in Anzahl und Aufbau den zugewiesenen Zahlen entsprechen. Das System wurde in Rosette entwickelt und verwendet die darin enthaltenen Programmsynthese Werkzeuge. Das besondere an den gefundenen Strategien ist, dass diese sich durch Regeln ausdrücken lassen, die eine Bedingung für eine Aktion an einer bestimmten Stelle überprüfen und damit über die Validität der Aktion entscheiden können. Damit verhalten sich die Strategien in etwa so wie ein Mensch versuchen würde Probleme dieser Art zu lösen. Erreicht wurde dies durch die Entwicklung einer DSL mit der sich die Regeln basierend auf Bedingungen und darauf folgenden Aktionen zusammenstellen lassen. Die Autoren konnten damit zeigen, dass es möglich ist mit solverunterstützen Werkzeugen domänenspezifisches Wissen zu generieren. Aus den Publikationen, die sich mit Rosette beschäftigen, ist diese am engsten mit dem in dieser Arbeit behandelten Thema verwandt. Schach unterhält jedoch zwei Spieler, die entgegengesetzte Ziele verfolgen. Dadurch wird mit dem Alternieren der Aktionen eine Komponente hinzugefügt, die keine äquivalente Eigenschaft bei Nonogrammen hat.

Philip Höfges [Hö19] entwickelte in seiner Master-Thesis ein Schach Modell in EventB [Abr10]. Der Bezug zu B im Kontext Schach ist für diese Arbeit äußerst Interessant. Gegebenenfalls können Teile der Struktur oder bestimmte Funktionen der B-Maschine abgeleitet und im hier zu entwickelnden Schach-Interpreter übernommen werden.

4 Implementierung

Im folgenden Kapitel wird die Struktur der fertigen Applikation vorgestellt. Nach Vorlage der studierten Publikationen [TB13, TB14] wurde zunächst das Grundgerüst mit dem Interpreter gelegt. Dabei handelt es sich um die Semantik verleihende Komponente der Applikation. Dies beinhaltet die Schachregeln, die beispielsweise Auskunft über die Validität einer Zugfolge geben. Daraufhin wurde der Interpreter in ein Rosette Programm überführt. Damit gehen in diesem Fall einige Änderungen bezogen auf die verwendeten Methoden und Datentypen einher. Abschließend wird auf die erfolgreich getesteten Queries eingegangen. Der vollständige Programmcode kann im folgenden Repository gefunden werden:

`https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/wroblewski-bachelor`

4.1 Aufbau des Interpreters

Der Spielzustand wird durch einen dafür eingeführten Datentypen **game** repräsentiert, der Attribute für das Spielbrett und den aktiven Spieler enthält. Weiter wird das Spielbrett durch eine Liste aus Paaren, welche die Position und den Bezeichner einer Spielfigur enthalten, dargestellt. Als Positionen können die im Schach gängigen Namen A1 bis H8 als Strings verwendet werden. Diese werden allerdings auf die Zahlen von 0 bis 63 gemappt, da für die Bewegungsvalidierung mit den Positionen gerechnet werden muss. Die externe Repräsentation existiert nur zu Lesbarkeitszwecken. Der Bezeichner gibt an, welchem Spieler die Figur gehört und um welche Art von Figur es sich handelt. So steht beispielsweise **w_rook** für einen Turm des weißen Spielers. Der aktive Spieler wird entsprechend durch die Zeichen **w** für weiß und **b** für schwarz dargestellt. Zum Start des Programmes werden alle so vorhandenen String Attribute in eine interne Repräsentation umgewandelt. Eine beispielhafte **game** Instanz vor der Umwandlung könnte wie folgt aussehen:

```
(game (list (list "A7" "w_rook")
            (list "B4" "w_knight")
            (list "B1" "w_king")
            (list "D6" "w_pawn")
            (list "C7" "b_knight")
            (list "D7" "b_king"))
      "b")
```

Um mit dem Programm zu interagieren, wird ein Ausgangszustand und ein durchzuführender Zug in die **move**-Methode (Quellcode 1) gegeben. Ein Zug wird dabei durch eine Start- und eine Zielposition ausgedrückt. Diese Methode stößt daraufhin die verschiedenen Abfragen des Interpreters an. Es wird bestimmt, ob sich eine Spielfigur auf der Startposition befindet und ob der gewünschte Zug innerhalb der für die Figur erlaubten Bewegungsmuster vorhanden ist. Dann wird überprüft, ob die Ausführung des Zuges den aktiven Spieler in eine unerlaubte Schach-Position bringen würde. Wenn alle Bedingungen erfüllt sind, wird der Zug vollzogen und das aktualisierte Spielbrett zurückgegeben. Ansonsten gibt die Methode **#f** zurück als Indikator, dass keine Aktion auf dem Spielbrett durchgeführt wurde.

Die Validierung des Bewegungsmusters ist an eine Reihe von **valid-move?**-Methoden gekoppelt, die jeweils für einen Figurentyp das Muster sowie mögliche andere Anforderungen festlegen. So kann sich ein Turm beliebig viele Felder in einer horizontalen oder vertikalen Linie bewegen, insofern keine Figur zwischen Start- und Zielposition liegt und sich auf der Zielposition entweder keine oder eine gegnerische Figur befindet. Anhand des Bezeichners der auf der Startposition lokalisierten Figur wird bestimmt, welche der **valid-move?**-Methoden aufgerufen wird.

Handelt es sich um eine valide Bewegung, wird der gewünschte Zug testweise lokal durchgeführt und daraufhin überprüft, ob es sich um eine valide Schach-Position handelt. Dafür

Quellcode 1: Implementierung einer Bewegungsaktion

```
(define (move game from to)
  (let* ([board (game-board game)]
         [num-from (pos-id->num from)]
         [num-to (pos-id->num to)]
         [active-player (game-active-player game)]
         [piece (assoc num-from board)])
    (if (and piece
             (valid-move? board piece num-to active-player))
             (legal-move? board piece num-to active-player))
        (execute-movement board piece num-to active-player)
        #f)))
```

Quellcode 2: Implementierung der has-legal-move? Methode

```
(define (has-legal-move? board active-player)
  (define pieces (get-players-pieces board active-player))
  (ormap (lambda (piece-to)
           (and
             (valid-move?
               board
               (first piece-to)
               (second piece-to)
               active-player)
             (legal-move?
               board
               (first piece-to)
               (second piece-to)
               active-player)))
         (cartesian-product pieces (my-range 0 64 1))))
```

wird über die Figuren des inaktiven Spielers iteriert und nach einer Figur gesucht, die einen validen Zug, der auf dem den König des aktiven Spielers endet, hat. Ist das der Fall, ist der gewünschte Zug illegal und kann nicht durchgeführt werden.

Ist die Aktion jedoch sowohl eine valide Bewegung als auch ein legaler Zug, wird die Figur entsprechend bewegt und dabei gegebenenfalls eine gegnerische Figur geschlagen. Es wechselt der aktive Spieler und falls dieser sich nun im Schach befindet, wird zunächst überprüft ob es einen legalen Zug gibt. Dafür wird jede mögliche Zielposition für jede Spielfigur des Spielers getestet (Quellcode 2). Gibt es keinen validen Zug, ist der Spieler in Schachmatt und das Spiel ist beendet. Ein Schachmatt-Flag wird mit dem finalen Zustand des Spielbretts zurückgegeben. Wenn es jedoch mindestens einen legalen Zug gibt, wird lediglich der aktualisierte Zustand des Spielbretts zurückgegeben. Dieser kann dann als Argument für den nächsten Zug verwendet werden.

Quellcode 3: Implementierung der Schachposition Überprüfung

```
(define (white-in-check? board)
  (let* ([white-king (get-piece-by-id board "w_king")]
         [black-pieces (get-players-pieces board 1)]
         [piece-attacking-king
          (if (not white-king)
              #f
              (findf (lambda (black-piece)
                       (valid-move? board
                                    black-piece
                                    (first white-king)
                                    1))
                     black-pieces))])
    (if piece-attacking-king
        #t
        #f)))
```

Die **move**-Methode (Quellcode 1) impliziert außerdem die nichtdeterministische Natur von Schach. Ausgehend von einem Spielzustand ermöglicht die Auswahl der **from** und **to** Parameter eine Vielzahl von Folgezuständen. Je nach ausgewählter Startposition wird eine entsprechend andere Figur betrachtet. Da diese zu sehr unterschiedlichen Bewegungen in der Lage sind, entsteht dadurch schon eine hohe Anzahl von Möglichkeiten. Die Zielposition erweitert diese Möglichkeiten durch Ereignisse, die potentiell eintreten können. Beispielsweise könnte durch die Aktion eine gegnerische Figur geschlagen oder der gegnerische Spieler in Schach gesetzt werden. Daher können sich die Folgezustände je nach Parameter stark unterscheiden. Führt man als Spieler einen Zug durch, ist also nicht festgelegt mit welchem Zug der gegnerische Spieler antworten wird. Die einzige Situation, in der Schach deterministisch ist, ist wenn ein Spieler nur noch einen legalen Zug durchführen kann. Dies wird vor dem Zug des Spielers überprüft (Quellcode 2). Dafür wird das kartesische Produkt aus allen übrigen Figuren des Spielers und allen Positionen des Spielbretts gebildet und für jede Kombination überprüft, ob es sowohl ein valides Bewegungsmuster als auch ein legaler Zug ist. Gibt es nur einen solchen Zug, ist die nächste Aktion damit determiniert.

Ähnlich der Suche nach einem legalen Zug funktioniert die Überprüfung auf ein Schachgebot (Quellcode 3). Statt alle Positionen als Zielpositionen auszuprobieren wird hier jedoch geschaut, ob eine der gegnerischen Figuren einen validen Zug auf den eigenen König ausführen kann. Die daraus resultierende Position muss dabei nicht wieder auf Validität geprüft werden.

Quellcode 4: Spezifikation zur Korrektheitsüberprüfung

```
1: (define (check-moves impl initial-state target-state)
2:   (assert (equal? (impl initial-state) target-state)))
```

Quellcode 5: Sketch für die benötigten Spielzüge

```
1: (define (required-moves initial-state)
2:   (possible-moves initial-state #:depth 3))
```

4.2 Aufbau der Queries

Der Großteil der unternommenen Versuche Queries abzusetzen und auch die ersten erfolgreichen Anfragen basierten auf der **synthesize**-Query. Eine der Grundbedingungen dieser und ebenso der **verify**-Query ist eine Spezifikation, die die Korrektheit einer Implementierung formalisiert. In dieser Spezifikation werden Annahmen über die Eingabewerte (**assume**) festgelegt und das Ergebnis mit Hilfe von festgelegten Behauptungen überprüft (**assert**). Für die in dieser Arbeit eingeführten Queries wird die Implementierung auf die übergebene Ausgangssituation angewendet und das Ergebnis mit dem gewünschten Zustand verglichen (Quellcode 4). Mit **assume** könnten an dieser Stelle beispielsweise Voraussetzungen für das übergebene Spielbrett festgelegt werden. Da dieses Programm jedoch nur intern zu Forschungszwecken entwickelt wurde, kann davon ausgegangen werden, dass es nur mit korrekten Eingaben verwendet wird.

Mit Hilfe der **synthesize**-Query kann dann eine Implementierung gefunden werden, sodass die Spezifikation erfüllt ist. Dafür sucht die Query innerhalb eines Raumes aus möglichen Implementierungen, die durch ein syntaktisches Programm mit auszufüllenden Lücken (*Sketch*) definiert sind. Dabei werden die Möglichkeiten angegeben, die in die Lücke eingesetzt werden können. Dies kann in simplen Fällen eine beliebige Konstante vom Typ Integer sein, ausgedrückt durch einen symbolischen Integer. Es ist aber auch möglich eine Grammatik zu formulieren, mit der ganze Ausdrücke produziert werden können. Angenommen es gebe eine Grammatik **possible-moves**, die alle möglichen Züge auf einem Schachbrett abbildet. Mit dieser Grammatik kann infolge ein Sketch für eine beliebige Zugfolge definiert werden (Quellcode 5). Dieser Sketch beschreibt den Raum aller Ausdrücke der possible-moves Grammatik, die einen Syntaxbaum mit maximaler Tiefe von drei haben, was in diesem Fall drei aufeinander folgende Züge bedeutet.

Mit einer **synthesize**-Query (Quellcode 6) wird der Solver nun nach einer Vervollständigung des Sketches befragt, die die Spezifikation **check-moves** erfüllt. Eine solche Query hat einen relativ simplen Aufbau. Mit **#:forall** können symbolische Konstanten angegeben werden, die einem Sketch als Eingabewerte übergeben werden. Mit **#guarantee** wird eine zu erfüllende Spezifikation, der Sketch und die Eingabewerte definiert. In diesem Fall werden keine symbolischen Konstanten, sondern die vom Nutzer bereitgestellten

Quellcode 6: Synthese-Query

```
1: (define sol
2:   (synthesize
3:    #:forall '()
4:    #:guarantee (check-moves
5:                 required-moves
6:                 initial-state
7:                 target-state)))
```

konkreten Repräsentationen der Spielzustände **initial-state** und **target-state**, in den Sketch übergeben. Das kommt daher, dass die Query nur eine Lösung für ein bestimmtes Paar von Start- und Zielzuständen statt für einen ganzen Raum finden soll. Dies folgt aus der Natur der Fragestellung, da beim Schach in der Regel nur einzelne Spielsituationen betrachtet werden.

Der Solver sucht beim Absetzen der Query dann nach einer Belegung der Lücken im Sketch, die die Spezifikation erfüllt. Ist das Programm terminiert, lässt sich mit einem Aufruf von **sol** der Inhalt des Solvers anzeigen. Existiert eine passende Belegung wird diese in Form eines Models ausgegeben, falls nicht erscheint lediglich ein **(unsat)**. Mit **(print-forms sol)** lässt sich eine lesbare Repräsentation des ausgefüllten Sketches ausgeben. Diese könnte beispielsweise wie folgt aussehen:

```
(define (required-moves game)
  (move
    (move
      (move initial-state "E2" "E3")
      "F7" "F5")
    "D2" "D4"))
```

Die **possible-moves**-Grammatik (Quellcode 7) ermöglicht eine beliebige Verkettung der **move**-Methode (Quellcode 1) und enthält dazu Lücken für die Start- und Zielposition der Bewegung. Um ein vollständiges Wort der Grammatik zu erhalten, wird letztendlich der in der Query übergebene **initial-state** als Eingabeparameter eingesetzt. Durch diese Struktur wird automatisch der Rückgabewert der innersten **move**-Methode zum Argument der darum liegenden. Dieser Vorgang wird bis zum äußersten Methodenaufruf weitergeführt.

Durch Anpassungen der Spezifikation können mit dieser grundlegenden Struktur auch andere Fragen beantwortet werden. Wird mit der Assertion nicht die Gleichheit eines Start- und Zielzustandes nach Ausführung der Implementierung sichergestellt, sondern beispielsweise das Vorkommen einer bestimmten Spielfigur oder ein bestimmter Zustand, wie Schachmatt, können mit einem sonst unveränderten Programm gänzlich andere Ziele erreicht werden. Im Rahmen dieser Arbeit wurde jedoch hauptsächlich mit der vorgestellen Spezifikation experimentiert.

Quellcode 7: Grammatik possible-moves mit Positionen und allgemeiner move-Methode

```
1: (define-grammar (possible-moves initial-state)
2:   [expr
3:    (choose initial-state (move (expr) (pos) (pos)))]
4:   [pos
5:    (choose 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 62 63)])
```

Eine Limitierung von Queries dieser Art ist, dass maximal eine Lösung ausgegeben wird, auch wenn mehrere existieren können. Weiter kann auch keine Aussage über die vom Solver untersuchten Ansätze getroffen werden und wie diese die Lösung gegebenenfalls beeinflussen könnten. So könnte mit einer Query beispielsweise eine Zugfolge gefunden werden, die den gegnerischen Spieler Schachmatt setzen würde. Es wäre dann interessant zu wissen, ob ein bestimmter Zug des gegnerischen Spielers dieses Ergebnis hätte verhindern können. Ist dies der Fall müsste dieser Zug in den untersuchten Ansätzen, die nicht zu einer Belegung geführt haben, die die Spezifikation erfüllt, vorhanden sein. Es gibt jedoch keine Möglichkeit auf diese Ansätze zuzugreifen und demnach Lösungen auszuschließen. Insofern wird die durch den Nichtdeterminismus bedingt hohe Anzahl von Möglichkeiten in Betracht gezogen, jedoch nur eine ganz spezielle Kombination von Zügen als Lösung ausgegeben. Diese Lösungen sind zwar valide, verhalten sich aber in etwa so als hätten beide Spieler Interesse daran auf diese Lösung hinzuarbeiten.

5 Details

Im folgenden Kapitel wird auf ausgewählte Aspekte der Entwicklungsphase eingegangen. Dabei werden aufgetretene Hindernisse und mögliche Lösungswege besprochen. Außerdem wird die gewählte Vorgehensweise vorgestellt und erläutert inwiefern diese geeignet war um die geplante Software umzusetzen.

5.1 Entwicklung eines Prototypen

Im Rahmen dieser Arbeit schien es sinnvoll einen Prototypen zu entwickeln, bei dem der Interpreter über einen geringeren Funktionsumfang verfügt. Es war im Vorfeld nicht bekannt, ob das gestellte Problem mit Rosette lösbar ist. Der Prototyp sollte zumindest ein erstes Indiz auf die Machbarkeit liefern und richtungsweisend für die Entwicklung der vollständigen Software sein. In dieser Phase ging es darum sich mit Racket und Rosette vertraut zu machen und auszutesten wie der Interpreter aufgebaut sein muss, um damit erfolgreich Queries zu lösen und wie die Queries implementiert werden. Dabei ist aufgefallen, dass es Wechselwirkungen zwischen Interpreter und Queries gibt, die den Ansatz einen Prototypen zu entwickeln für umfangreichere Projekte interessant macht.

Quellcode 8: Grammatik possible-moves für eine Figur

```
1: (define-grammar (possible-moves initial-state)
2:   [expr
3:    (choose initial-state ((move) (expr)))]
4:   [move
5:    (choose pawn-move knight-move)])
```

Eine der wichtigen Aufgaben des Schach-Interpreters ist es, die Bewegungsmuster der verschiedenen Spielfiguren zu kodieren. Ein erster Ansatz dafür war, dass es eine Vielzahl von **move**-Methoden geben soll, die je ein Muster abbilden. Im Falle der Bauern würde es sich also um vier Methoden handeln, eine für die einfache Vorwärtsbewegung, eine für die zweifache Vorwärtsbewegung, eine für den Angriff nach vorne links und eine für den Angriff nach vorne rechts. Diese werden aufgerufen und die entsprechende Bewegung auf dem Spielbrett durchgeführt. Der Gedankengang dabei war, dass der Solver eine Verkettung dieser Methoden zusammenstellen könnte, die einen vorgegebenen Spielzustand in einen gewünschten Zustand überträgt. Die folgenden Unterkapitel zeigen, wie sich diese Grundidee über mehrere Iterationen weiterentwickelt hat, welche Probleme bei den jeweiligen Iterationen aufgetreten sind und wie dies letztendlich zu dem Ansatz geführt hat, der in der vollständigen Applikation verwendet wurde.

5.1.1 Mehrfachbewegung mit einer Figur

Zum Zeitpunkt als das erste Mal erfolgreich eine Query gelöst werden konnte, umfasste der Prototyp eine **pawn-move**-Methode für die einfache Vorwärtsbewegung des Bauern, eine **knight-move**-Methode für eines aus den vielen möglichen Bewegungsmustern des Springers, die Spezifikation (Quellcode 4), den Sketch (Quellcode 5) und die **synthesize**-Query (Quellcode 6). Die Grammatik **possible-moves** (Quellcode 8) wird in diesem Fall also eine beliebige Verkettung dieser beiden Bewegungen bilden können, um alle möglichen Züge abzudecken. Der innerste Aufruf einer der move-Methoden bekommt mit **initial-state** den Ausgangszustand des Spielbretts übergeben. Der Rückgabewert ist das aktualisierte Spielbrett und ist gleichzeitig das Argument für die umschließende move-Methode. In dieser Variante des Prototypen gibt es noch kein richtiges Spielbrett mit mehreren Figuren. Der Parameter **initial-state** ist lediglich die Position, auf der sich die eine Figur befindet. Die **move**-Methoden bewegen diese Figur dann um einen festgelegten Wert. Dies sollte zeigen, dass es möglich ist in einer Grammatik Methoden zu verketten.

Quellcode 9: Grammatik possible-moves für mehrere Figuren

```
1: (define-grammar (possible-moves initial-state)
2:   [expr
3:    (choose initial-state ((move) (expr) (piece)))]
4:   [move
5:    (choose knight-move pawn-move)]
6:   [piece
7:    (choose (first initial-state) (second initial-state))])
```

5.1.2 Einfachbewegung mit mehreren Figuren

Um mehrere Figuren auf dem Spielbrett abbilden und bewegen zu können, braucht es eine geeignete Repräsentation des Spielzustandes. In der nächsten Iteration des Prototypen wird dafür eine Liste aus Paaren von Positionen und Bezeichnern der Figuren verwendet. Die Bewegungsmethoden bekommen nicht mehr nur das Spielbrett, sondern auch die Figur, mit der die Bewegung durchgeführt werden soll, übergeben. Die Grammatik wird also dementsprechend angepasst und eine neue Lücke für die Figur eingeführt (Quellcode 9).

Hier werden die ersten zwei Figuren des Spielbretts als Belegung für die Lücke in Betracht gezogen. In der ausgewählten **move**-Methode wird überprüft, ob die übergebene Figur die zu der Methode passende ist und dementsprechend die Bewegung ausgeführt oder nicht. Es gibt jedoch zwei Probleme bei dieser Variante. Da die Auswahl der Figur in der Grammatik auf das initiale Spielbrett zugreift, werden Änderungen auf dem Brett für darauf folgende Bewegungen nicht beachtet. Wird beispielsweise versucht zwei Mal **pawn-move** mit der ersten Figur durchzuführen, befindet sich die beim zweiten Aufruf übergebene Figur schon nicht mehr auf dem Spielbrett und die Bewegung kann nicht durchgeführt werden. Das führt dazu, dass mit jeder Figur nur eine Bewegung korrekt ausgeführt werden kann, da es in der Grammatik keine Möglichkeit gibt auf den neuen Zustand zuzugreifen. Das zweite Problem bezieht sich auf die Ausgabe. Da bei einer gefundenen Lösung eine Repräsentation des ausgefüllten Sketches ausgegeben wird und in dieser Form der Grammatik ausschließlich Methodenaufrufe und Variablen vorkommen, spiegelt sich dies in der Ausgabe wider. So ist

```
(pawn-move
  (pawn-move
    (initial-state
    (second initial-state))
  (first initial-state)))
```

eine korrekte Lösung für eine getestete Query, jedoch ist schwer zu erkennen welche Figuren von welcher Position auf welche Zielposition bewegt wurden.

Quellcode 10: Grammatik possible-moves mit globalem Spielbrett

```
1: (define-grammar (possible-moves)
2:   [expr
3:    (choose (move) (and (expr) (expr)))]
4:   [move
5:    (choose (knight-move (knight-piece)) (pawn-move (pawn-piece)))]
6:   [pawn-piece
7:    (choose (first-pawn) (second-pawn) (third-pawn) ...)]
8:   [knight-piece
9:    (choose (first-knight) (second-knight))])
```

5.1.3 Mehrfachbewegung mit mehreren Figuren

Um das Problem bezüglich der Aktualisierung des Spielbretts zu lösen, wurde für die nächste Iteration des Prototypen ein globaler Spielzustand eingeführt. Dafür wird mit **(struct board (pieces) #:mutable #:transparent)** ein Datentyp für das Spielbrett erstellt, der ein Attribut für die Figuren enthält und sich modifizieren lässt. Damit kann ein Spielbrett definiert werden, auf dem alle ausgeführten **move**-Methoden operieren. Außerdem werden Methoden bereitgestellt, mit denen bestimmte Figuren aus diesem Spielbrett abgerufen werden können. In der Grammatik (Quellcode 10) ist somit kein initialer Zustand mehr als Parameter notwendig und die den **move**-Methoden übergebenen Figuren können bereits auf die passenden beschränkt werden. Da das Spielbrett aber nicht mehr als Parameter verwendet wird, funktioniert die Verkettung der **move**-Methoden hier mit einem **and** anstatt den Rückgabewert der inneren als Argument in die äußere Methode hineinzugeben. In dieser Variante können auch mehrere Bewegungen mit einer Figur ausgeführt werden, aber die Ausgabe ist durch das **and** noch schlechter lesbar. Da das Verändern des Spielbretts mit dem selbsterstellten Datentyp etwas Verwaltungsaufwand bedeutet, wurde eine allgemeine **move**-Methode eingeführt, welche die Form der Bewegung von den spezifischen **move**-Methoden übergeben bekommt und diesen Aufwand abarbeitet.

5.1.4 Mehrfachbewegung mit mehreren Figuren und verbesserte Ausgabe

Als alternative Lösung wurde damit experimentiert den direkten Zugriff auf die Figuren aus der Grammatik zu entfernen (Quellcode 11). Stattdessen wird aus allen möglichen Positionen ausgewählt und in der **move**-Methode vor der Durchführung überprüft, ob die entsprechende Figur auf dieser Position steht. Dieser Ansatz löst nicht nur die Problematik bezüglich der Aktualisierung des Spielbretts, sondern es wird auch eine Ausgabe produziert, die die Startpositionen enthält.

Quellcode 11: Grammatik possible-moves mit Positionen

```
1: (define-grammar (possible-moves initial-state)
2:   [expr
3:    (choose initial-state ((move) (expr) (pos)))]
4:   [move
5:    (choose knight-move pawn-move)]
6:   [pos
7:    (choose 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 62 63)])
```

5.1.5 Anwendung des Gelernten

Durch die Verknüpfung dieser Idee und der einer allgemeinen **move**-Methode ergibt sich die finale Version des Prototypen. Der letzte Schritt ist die gewünschte Endposition der Bewegung ebenfalls als Parameter anzulegen und von der Grammatik (Quellcode 7) auswählen zu lassen. Statt der spezifischen **move**-Methoden wird hier direkt die allgemeine aufgerufen. Diese überprüft ob und welche Figur sich auf der übergebenen Startposition befindet und gibt die Positionen entsprechend weiter, um das Bewegungsmuster validieren zu lassen. Gibt die spezifische Methode zurück, dass die Figur diese Bewegung ausführen kann, wird der Zug vollzogen. Die Ausgabe des Solvers ist beispielsweise mit

```
(move (move (move initial-state 9 17) 1 18) 2 10)
```

sehr gut lesbar, da die Reihenfolge und auch die Form der Bewegungen unmittelbar erkannt wird. Bei der Entwicklung des richtigen Interpreters wurden die Zahlen der Positionen in der Grammatik noch durch die Strings A1 - H8 ersetzt.

Es wurde gezeigt, dass das Design des Interpreters bereits einen Einfluß auf die Queries hat. Durch neue Ansätze in der Grammatik wurden Teile des Interpreters häufiger umgeschrieben oder angepasst. Da es für unerfahrene Entwickler bei größeren Projekten schwierig ist die Wechselwirkungen in der Planungsphase ausreichend einzuschätzen, bietet sich die Entwicklung eines Prototypen an. Diese Vorgehensweise weicht etwas von der vorgeschlagenen Strategie ab, bei der zuerst der Interpreter fertiggestellt und dann erst die Queries implementiert werden sollen. Es wird jedoch Wissen darüber gewonnen wie der Interpreter aussehen muss, damit die gewünschten Ziele erreicht werden können. Dadurch können gegebenenfalls umfangreiche Anpassungen im Nachhinein eingespart werden.

5.2 Die Konvertierung nach Rosette

Nach den Experimenten mit dem Prototypen wurde mit der Entwicklung der vollständigen Software begonnen. Dazu wurde zunächst das Grundgerüst des Interpreters aus dem Prototypen entnommen und um die fehlenden Regeln ergänzt. Dies umfasste die

Bewegungsvalidierung für die übrigen Spielfiguren und die Überprüfung von Schach- und Schachmatt-Zuständen. Dabei wurde auch die Unterstützung von Positionenangaben als Strings eingeführt. Durch diese Ergänzungen verlief die Konvertierung nach Rosette, die bei dem Prototypen unproblematisch war, nicht auf Anhieb fehlerfrei.

Damit die solverunterstützten Werkzeuge verwendet werden können, muss der Interpreter zunächst in ein Rosette Programm überführt werden. Der erste Schritt dafür ist das Ersetzen von **#lang racket** durch **#lang rosette**. Damit wird die Applikation bereits als Rosette Programm ausgeführt. Potenziell vorhandene Unit-Tests sollten ergebnisgleich durchlaufen wie vor der Änderung, da sich ein Rosette Programm ohne symbolische Werte genauso verhält wie die Racket Version. An dieser Stelle wäre es möglich erste Queries einzusetzen und sich die Ergebnisse anzuschauen. Diese Vorgehensweise wurde in dieser Arbeit gewählt. Es hat sich jedoch herausgestellt, dass sich noch einige mögliche Fehlerquellen im konvertierten Code befinden.

Das Rosette System beinhaltet zwei auswählbare Dialekte der Sprache Rosette, ein *safe dialect* (sicherer Dialekt) und ein *unsafe dialect* (unsicherer Dialekt). Der sichere Dialekt besteht aus einer Teilmenge von Racket, welche die Kernfunktionen abdeckt und garantiert erwartungsgemäß vom symbolischen Compiler aufgelöst werden kann. Dieser Teil von Racket wird auch als *lifted* bezeichnet und kann im Zusammenhang mit symbolischen Werten und anderen solverunterstützten Konstrukten verwendet werden. Der unsichere Dialekt hingegen beinhaltet die Gesamtheit von Racket. Werden in einem Rosette Programm *unlifted* Konstrukte aufgerufen, greift die Ausführung auf den Racket Interpreter zurück. Das kann zu fehlerhaftem oder unerwartetem Verhalten führen, zum Beispiel wenn der Racket Interpreter versucht symbolische Werte zu lesen und damit zu interagieren.

Durch das Einsetzen von **#lang rosette** wird der unsichere Dialekt verwendet. Somit wird das Programm trotz Vorkommen ungelifteter Methoden ohne Warnungen ausgeführt. Unerfahrene Rosette Benutzer geraten so möglicherweise in folgende Situation: Es wird eine Query, zum Beispiel mit Programmsynthese, eingeführt und dem Solver vorgelegt; dieser gibt nun **(unsat)** zurück, es gibt also kein Modell für die das Programm repräsentierende Formel; das kann entweder die korrekte Lösung sein, oder aber es liegt ein Fehler im Code oder der Query vor. Im Falle des Schach-Interpreters führten die ersten Versuche mit einer Query jedes mal zu keiner Lösung, obwohl das gestellte Problem offensichtlich eine Lösung hatte. Der Solver sollte eine Zugkombination finden, die einen Spielzustand in einen anderen überführt, in dem lediglich eine Figur ein Feld nach vorne bewegt wurde. Mit der Rückgabe **(unsat)** suggeriert der Solver, dass es diesen Zug nicht gibt. Das Problem war jedoch die Verwendung von Strings beziehungsweise String-Operationen auf symbolischen Werten, da Strings nicht zu den lifted Datentypen gehören. Die Ausgabe **(unsat)** ist technisch korrekt, aber an dieser Stelle nicht hilfreich zur Lokalisierung des Fehlers.

Als Lösung für dieses Problem bietet sich eine andere Vorgehensweise an. Wird anstelle von **#lang rosette** **#lang rosette/safe** angegeben, verwendet man den sicheren Dialekt von Rosette. Versucht man das beschriebene Programm damit auszuführen, erhält man sofort Fehlermeldungen für alle nicht definierten Methoden, in diesem Fall also die verwendeten

Quellcode 12: Synthesize-Query mit Input Mapping

```
1:  (define sol
2:    (synthesize
3:     #:forall '()
4:     #:guarantee (check-moves
5:                  required-moves
6:                  (game-strings->nums
7:                   (game (list (list "D2" "w_pawn")
8:                               (list "E7" "b_pawn")
9:                               (list "F2" "b_pawn")) "b"))
10:                 (game-strings->nums
11:                  (game (list (list "D3" "w_pawn")
12:                              (list "E6" "b_pawn")
13:                              (list "F2" "b_pawn")) "b")))))
```

unlifted Methoden. Diese kann man durch vorhandene lifted Methoden oder durch selbst geschriebene Methoden, die nur aus lifted Konstrukten bestehen, ersetzen. Beispiele für unlifted Konstrukte, die ich zuerst verwendet habe, sind: die Datentypen String und Hashmap und darauf agierende Methoden, wie substring, string-prefix? und string-append, die Methoden index-of, range, cartesian-product und group-by. Es ist prinzipiell möglich die Datentypen String und Hashmap unter der Bedingung, dass diese nur in konkreten Zuständen vorkommen, zu verwenden. Es hat sich jedoch als effektiv herausgestellt diese Fehlerquellen zunächst auszuschließen und eine funktionierende Query zu implementieren, auf der man aufbauen kann. Von dieser Position aus können gezielt wieder unlifted Konstrukte hinzugefügt werden. Dann wird außerdem sofort erkannt, wenn dadurch Fehler eingeführt werden.

Für die Schach-Applikation ist der Einsatz von Strings für die Positionen und die Figuren relevant, da die Nutzung mit den tatsächlichen Positionen A1 - H8 und Kürzeln, wie **w_pawn** und **w_king**, intuitiver ist. Ein Weg damit umzugehen ist die Argumente der Nutzereingabe in einer lesbaren Form zu akzeptieren und diese dann in eine Form umzuwandeln, mit der der Interpreter gut rechnen kann (Quellcode 12). Im Beispiel wird eine **synthesize**-Query definiert, die als Eingabe zwei **games** in der leserlichen Variante erhält. Die **game-strings->nums**-Methode durchläuft den Inhalt und ersetzt bei jeder Figur Position und Bezeichner und auch den aktiven Spieler durch die definierte Nummer. Die einzelnen Umwandlungen werden letztendlich in Methoden mit einem **case**-Konstrukt (Quellcode 13) durchgeführt. Dieses Konstrukt verwendet standardmäßig **equal?**, eine Methode, die auch in ihrer Spezifikation für Strings eine lifted Methode ist. Damit ist auch **case** lifted und kann in Interaktion mit symbolischen Werten problemlos verwendet werden.

Neben den Datentypen gibt es wie bereits erwähnt auch vereinzelte Methoden, die nicht Teil von **rosette/safe** sind, zum Beispiel **range**, **index-of** und **cartesian-product**. Diese wurden zuerst genutzt um über die möglichen Positionen zu iterieren, alle Kombinationen

Quellcode 13: Konvertierung der Positionen

```
1: (define (pos-id->num pos)
2:   (case pos
3:     [("A1") 0] [("B1") 1] [("C1") 2] [("D1") 3] [("E1") 4]
4:     [("F1") 5] [("G1") 6] [("H1") 7] [("A2") 8] [("B2") 9]
5:     [("C2") 10] [("D2") 11] [("E2") 12] [("F2") 13] ...
6:     [else pos]))
```

von Figuren und Zielpositionen zu bilden oder die Position einer Figur in der Spielbrett-Liste zu bekommen. Man kann solche Methoden durch selbstgeschriebene Varianten, die die gleiche Funktionalität haben, ersetzen oder eine bestehende lifted Methode verwenden und den umgebenden Code anpassen, um die gleiche Funktionalität zu erreichen. Statt für die Zugausführung eine Figur mit **index-of** aus der Liste zu erhalten und die Position zu modifizieren, lässt sich beispielweise mit **remove** auch die entsprechende Figur entfernen und mit **append** wieder mit aktualisierter Position einfügen.

5.3 Performance

Durch das Experimentieren mit komplizierteren Queries und dem vollen Funktionsumfang des Interpreters stellte sich heraus, dass die Anfragen teilweise sehr unterschiedliche Laufzeiten haben können. Die schnellsten Queries sind nach wenigen Sekunden gelöst, während andere erst nach mehreren Minuten oder sogar gar nicht vom Solver beantwortet werden. Um verschiedene Szenarien zu entwickeln und anschließend auszutesten, kann für eine Query mit **(time *<query>*)** die Laufzeit ausgegeben werden. Durch diese Tests und das Einfügen von Logs an verschiedenen Punkten des Programms, konnten einige Gründe für die variierende Performance gefunden werden. Dadurch lässt sich auf bestimmte Verhaltensweisen des symbolischen Compilers schließen.

Zuerst ist aufgefallen, dass eine Anfrage sehr viel langsamer wird sobald Figuren verschiedener Art auf dem Spielbrett stehen. Die Logs zeigen in diesen Fällen, dass das Programm durch jede der **valid-move?**-Methoden läuft. Das bedeutet, dass die Figur auf der von der Grammatik ausgewählten Startposition scheinbar jedem möglichen Figurentyp entspricht. Da dieses Ereignis jedoch auch bei Versuchen auftritt, bei denen nur zwei verschiedene Figurentypen auf dem initialen Spielbrett stehen, handelt es sich oft um Pfade, die nicht existieren können. Ist stattdessen nur ein Figurentyp auf dem Spielbrett, wird nur der richtige Pfad verfolgt und die Logs zeigen an, dass nur eine **valid-move?**-Methode aufgerufen wird.

Um Hinweise darauf zu erhalten warum dieser Ablauf auftritt, wurden weitere Logs zu Beginn der **move**-Methode eingeführt, die den aktuellen Zustand der Eingabeparameter und der gefundenen Figur ausgeben. Man erkennt dabei, dass das Spielbrett im ersten Durchlauf einen konkreten Wert hat, nämlich den der Eingabe. Alle anderen Werte sind zu

diesem Zeitpunkt symbolisch und müssen also noch vom Solver aufgelöst werden. Dies zeigt sich beispielsweise an Ausgaben wie **(ite* (...))** für die Positionswerte aus der Grammatik oder **(union #size: 2 #:hash ...)** für die gefundene Figur auf der Startposition. Das **ite*** bezieht sich auf eine Verzweigung ausgehend vom **choose**, also eine eingeführte Lücke der Grammatik.

Symbolic Unions (symbolische Vereinigungen) sind symbolische Werte eines Datentyps, der nicht lifted ist, wie in diesem Fall einer Liste. Rosette verpackt diese Datentypen mit sogenannten *Guards*, symbolische Terme vom Typ Boolean, die jeweils für zwei mögliche konkrete Zustände des Wertes stehen. Wird der Term aufgelöst und ist **#t**, findet einer der konkreten Werte Verwendung, bei **#f** der entsprechend andere. Mit **(union-contents)** erhält man den Inhalt einer symbolischen Vereinigung und sieht im Falle der Figur, dass die Guards Terme enthalten, die sich je nach Auswahl der Startposition auflösen.

Dies ist bis dahin auch das erwartete Verhalten, was folgt ist allerdings, dass das Programm bis zu Ende durchläuft und bei Verzweigungen, die aufgrund von symbolischen Werten entschieden werden, für alle Möglichkeiten Pfade geöffnet werden. Da auch die Figur symbolisch ist und diese den gesamten Ablauf entlang weitergereicht wird, müssen scheinbar alle vorhandenen Pfade vom Solver abgegangen werden, auch wenn diese von den Eingabewerten ausgeschlossen werden. Wird jedoch ein Spielzustand übergeben, in dem nur eine Art von Figur vorhanden ist, wird auch nur die dazugehörige **valid-move?**-Methode aufgerufen. Der Solver erkennt also, dass nach der Auflösung nur eine Figur mit einem bestimmten Bezeichner übrig bleiben kann und handelt dementsprechend. Diese Vermutung bestätigte sich auch bei einem Versuch, in dem das Wechseln des aktiven Spielers entfernt wurde. Da der aktive Spieler nur in wenig rechenintensiven Methoden gebraucht wird, ist die Laufzeitdifferenz allerdings nicht so groß. Es ist dennoch eindeutig, dass das Fehlen der Entscheidung die Anzahl der abgelaufenen Pfade verringert und dadurch die Zeit verkürzt wird. Dies ist gerade in Fällen wie diesem eine andere Vorgehensweise als erwartet, da durch die alternierende Natur der Spielerreihenfolge für den Menschen eindeutig ist welcher konkrete Wert an welcher Stelle vorliegt.

Als eine mögliche Ursache für Performanceprobleme nennt die Rosette Dokumentation [Tor23a] fehlende Konkretisierung. In einer Beispielsituation hat man bereits durch eine Bedingung etabliert, dass eine symbolische Konstante nur drei bestimmte Werte annehmen kann. Übergibt man diese Konstante daraufhin einer Methode als Argument, betrachtet diese die Konstante wieder so als könnte sie jeden Wert annehmen. Dadurch werden Pfade eröffnet, die durch den vorherigen Programmablauf schon ausgeschlossen wurden. Dies ist ein ähnliches Verhalten wie die Ausführung der verschiedenen **valid-move?**-Methoden, die anhand der Eingaben nicht vorkommen dürften. Die Startposition wird an keiner Stelle im Programm konkretisiert. Also liegen diese und auch die Figur, die mit der Startposition aus dem Spielbrett entnommen wird, als symbolischer Wert vor. Dadurch kann in der **valid-move?**-Methode nicht bestimmt werden, welche Figur in diesem Moment behandelt wird und welcher Pfad damit der richtige ist. Die Folge ist hier ebenfalls, dass Zweige erstellt werden, die sich als nicht ausführbar herausstellen und damit überflüssige Rechenzeit in Anspruch nehmen.

Um die Positionswerte in einen konkreten Zustand zu bringen, wurde ein Wrapper für die **move**-Methode, also für den Einstiegspunkt des Programms, geschrieben. Dieser Wrapper identifiziert die Position und ruft den Rest des Programms anschließend mit dem expliziten Wert dieser Position auf. Die Logs zeigen bei dieser Variante, dass sowohl die Positionen als auch die Spielfigur in der **move**-Methode zunächst als konkrete Werte vorliegen. Dadurch wird auch bei der Bewegungsvalidierung nur der Pfad mit der tatsächlich ausgewählten Figur genommen.

Das Problem bei dieser Vorgehensweise ist, dass das Programm nun tatsächlich für jede mögliche Kombination von Positionswerten ausgeführt wird anstatt den Solver die symbolischen Werte auflösen zu lassen. Dies führt zu einem quadratischen Wachstum von Durchläufen und ist damit schon bei einer **depth** von zwei nicht mehr schneller als die Variante ohne Wrapper. Ein weiter ausschlaggebender Punkt dabei ist, dass das Spielbrett ab Tiefe zwei bereits symbolisch ist. Das kommt daher, dass der Zustand des Spielbretts im zweiten Zug davon abhängig ist, welche Figur im ersten Zug bewegt wurde. Damit ist auch die Spielfigur im zweiten Zug symbolisch. Man gewinnt den Vorteil dieser Methode also nur für den ersten Durchlauf des Programms.

Die Natur des Problems, also der Nichtdeterminismus und die Abhängigkeit der Eingaben von der Ausgabe des vorherigen Zuges, machen diesen Lösungsansatz für eine Performancesteigerung nicht anwendbar. Versuche mit unterschiedlicher **depth** haben bestätigt, dass der Solver dem Programmablauf zunächst bis zum Ende folgt, bevor die symbolischen Werte aufgelöst werden. Das ist bereits daran erkennbar, dass gleiche Anfragen, bei denen nur die Tiefe verändert wurde, unterschiedliche Laufzeiten haben. Es wird die gleiche Lösung gefunden, aber es gibt keinen Weg für den Solver früher abzubrechen.

Ein anderer Lösungsansatz ergibt sich aus der Verwendung des *Symbolic Profilers* [BT18], einem Werkzeug zur Diagnostizierung von Performance einschränkenden Schwachstellen. Der Profiler wird mit **raco symprofile** ***<program-to-execute>*** ausgeführt und erzeugt eine browserbasierte Ausgabe. In dieser wird jede aufgerufene Methode aufgeführt und anhand von verschiedenen Metriken eine Bewertung zugewiesen. Eine hohe Bewertung steht für eine erhöhte Wahrscheinlichkeit, dass diese Methode ein Grund für Performanceprobleme ist. Die angegebenen Metriken sind die Anzahl der Aufrufe einer Methode, die Gesamtlaufzeit aller dieser Aufrufe, die Anzahl der von dieser Methode erstellen symbolischen Terme sowie der ungenutzten Terme, die Größe aller von dieser Methode erstellten symbolischen Vereinigungen und die Anzahl der *Merges* (Verschmelzungen).

In einer beispielhaften Ausgabe des Profilers (Abbildung 1) ist erkennbar, dass besonders die Methoden **assoc** und **filter** zur erhöhten Laufzeit beitragen. Diese haben mit 3.1 und 2.4 einen hohen Score, welcher sich durch die große Anzahl an erstellten Terms (321.501 und 553.228), erstellten Zweigen in den symbolischen Vereinigungen (53.306 und 17.152) und in den Verschmelzungen verwendeten Zweigen (155.721 und 375.680) ergibt. Dabei hat **filter** vier Aufrufe, ausgehend von der **my-range**-Methode im Rahmen der Bewegungsvalidierung für einen Turm. Das zeigt, dass die Menge der Aufrufe hier nicht an symbolische Werte geknüpft ist, die Auflösung der symbolischen Terme erfordert

aber eine lange Laufzeit. Eine Vermutung für die Ursache ist, dass es sich bei **filter** um eine Listenoperation handelt und das Auflösen dieser Operationen und den für Listen erforderlichen symbolischen Vereinigungen aufwendig ist. Im Fall von **assoc** sind es hingegen 4.096 Aufrufe, ebenfalls ausgehend von der Turmbewegung, jedoch weiter bedingt durch das von **my-range** produzierte Intervall. Die Abhängigkeit von symbolischen Werten erklärt die hohe Anzahl der Aufrufe und vermutlich ebenso die hohe Anzahl der ungenutzten Terme (251.878), da viele der eröffneten Pfade im Auflösungsprozess nicht relevant sind. Da es sich bei **assoc** gleichfalls um eine Listenoperation handelt, wird die hohe Anzahl der Aufrufe zur erhöhten Laufzeit des Programms beitragen.

Die Ergebnisse des Profilers weisen also in die Richtung einzelne rechenintesive Methoden durch andere zu ersetzen. Da letztendlich aber Listenoperationen notwendig sind, um mit der ausgewählten Datenstruktur für das Spielbrett zu interagieren, konnte auch eine neue Implementierung von **assoc** nicht zu einer Performancesteigerung führen (Abbildung 2). Daraus ergibt sich als Lösungsansatz die Datenstruktur für den Spielzustand auszutauschen und dabei ausschließlich den Datentyp Integer oder sogar Bitvektor zu verwenden [Tor23a]. Dies könnte möglich sein, indem für jede Art von Spielfigur ein Bitvektor mit 64 Stellen angelegt wird, bei dem jede Stelle für eine Position auf dem Spielbrett steht und jede 1 für das Vorkommen der entsprechenden Figur.

Im Rahmen dieser Arbeit wird jedoch auf weiteres Experimentieren verzichtet, da es hier primär um die Machbarkeit und das gewonnene Wissen geht. Für weitere Forschung ist dieser Vorschlag aber ein vielversprechender erster Ansatz, da die identifizierten Problemstellen in dieser Implementierung wegfallen und der Aufwand vergleichbar niedrig ist. Das Öffnen vieler Zweige liegt durch die hohe Anzahl möglicher Positionskombinationen in der Natur des Problems und durch den Nichtdeterminismus ist das Spielbrett ab dem zweiten Zug immer symbolisch. Lösungsansätze, die sich auf diese Bereiche konzentrieren, sind daher vermutlich schwerer umzusetzen.

5.4 Weitere Queries

Da durch das Ausprobieren verschiedener Szenarien und den Profiler die problematischen Programmteile identifiziert werden konnten, wurden diese für nachfolgende Versuche entfernt, indem Bewegungvalidierungen für Turm, Läufer oder Königin direkt ein **#f** zurückgeben. Es zeigt sich wie erwartet eine deutlich verbesserte Performance (Abbildung 3) und damit ist diese Version praktischer für Tests, auch wenn dafür die Funktionalität eingeschränkt ist.

Eine weitere von Rosette bereitgestellte Query ist *Angelic Execution*. Es sei ein Programm gegeben, das symbolische Werte beinhaltet. Dann kann mit Angelic Execution eine Belegung dieser Werte gefunden werden, die das Programm ohne Fehler terminieren lässt. Mit Hilfe von Assumptions und Assertions lassen sich also die Rahmenbedingungen festlegen, die die Auswahl der Belegung beschränken. Diese Query war zuerst für das Schachprogramm als Lösungsweg angedacht, wurde aber aufgrund von Missinterpretation und mangelnden

Quellcode 14: Angelic Execution

```
(define-symbolic from-1 to-1 from-2 to-2 from-3 to-3 integer?)
(solve
   (begin
     (assume (<= 0 from-1 63))
     (assume (<= 0 to-1 63))
     (assume (<= 0 from-2 63))
     (assume (<= 0 to-2 63))
     (assume (<= 0 from-3 63))
     (assume (<= 0 to-3 63))
     (assert (equal? (move
                      (move
                       (move
                        (game-strings->nums
                         (game (list
                          (list "D3" "b_pawn")
                          (list "E7" "b_pawn")
                          (list "F2" "w_pawn")) "b"))
                        from-1 to-1)
                       from-2 to-2)
                      from-3 to-3)
                     (game-strings->nums
                      (game (list
                       (list "D2" "b_pawn")
                        (list "F3" "w_pawn")
                         (list "E6" "b_pawn")) "w")))))))
```

Kenntnissen verworfen. Mit der Weiterentwicklung des Interpreters und dem letztendlich vorliegendem Stand wurde dieser Weg jedoch erneut erkundet.

Da die Grammatik der **synthesize**-Query lediglich die Eingabewerte der Start- und Zielpositionen auswählt, lässt sich die gleiche Funktionalität mit Angelic Execution erreichen (Quellcode 14). Anstatt der Grammatik werden hier direkt symbolische Werte für die Positionen definiert und der **move**-Methode als Argumente übergeben. Ein dadurch entstehender Unterschied zur **synthesize**-Query ist, dass genau angegeben werden muss wie viele Züge durchgeführt werden sollen. Die **depth** bei einem Sketch bestimmt hingegen die maximale Tiefe und bewirkt außerdem, dass alle Wege bis zu dieser Tiefe evaluiert werden. Es kann jedoch auch eine Lösung oberhalb dieser Tiefe erkannt und ausgegeben werden. Mit Angelic Execution ist dies nicht möglich.

Ein Durchlauf mit dem Profiler (Abbildung 4) zeigt, dass auch die Anzahl der Funktionsaufrufe, Terme, symbolischen Vereinigungen und Merges sehr ähnlich aussieht wie bei der Synthese. Die Zeit, die der Solver zum Auflösen der Anfrage benötigt (hellblauer Bereich, hier abgeschnitten), ist jedoch in Versuchen mit Angelic Execution um einiges kürzer. Dies ist ein Hinweis darauf, dass sich diese Art von Query für das gestellte Problem besser eignet.

Quellcode 15: Kurzform der Schachmatt-Query

```
(solve
   (...
     (assert (equal?
               (game-c-check-mate
                (fast-move
                 (game-strings->nums (game ...)
                 from-1 to-1))
               #t)))))
```

Um eine Query zu implementieren, die eine Lösung für „Matt-in-drei-Zügen“-Rätsel findet, wird die Datenstruktur **game** um ein Flag für den Schachmatt-Zustand des aktiven Spielers ergänzt. Im Programmablauf des Interpreters wird außerdem ein Aufruf der **check-mate?**-Methode vor die Rückgabe des aktualisierten Spielzustandes gesetzt. Diese Methode überprüft erst ob der gewechselte aktive Spieler sich auf dem neuen Spielbrett im Schach befindet. Ist dies der Fall wird weiter geschaut, ob der Spieler einen legalen Zug (Quellcode 2) ausführen kann. Falls nicht, befindet sich der Spieler im Schachmatt und das Flag wird entsprechend gesetzt.

An der eigentlichen Query müssen nicht viele Anpassungen vorgenommen werden. Es wird immer noch nach einer Zugfolge gesucht, die den übergebenen Startzustand in einen gewünschten Zielzustand überführt. Der Zielzustand wird in diesem Fall jedoch nicht über eine bestimmte Belegung des Spielbretts, sondern über den Zustand der Schachmatt-Flags definiert (Quellcode 15). Bei dem vorliegenden Quellcode handelt es sich um ein Beispiel einer sehr verkürzten Anfrage, da nur ein einziger Zug betrachtet wird und im Interpreter außerdem jede Verzweigung aufgrund des aktiven Spielers entfernt wurde. In den Versuchen konnten nur auf diese Weise Ergebnisse erzielt werden, da die Auflösung beim Hinzufügen der Schachmatt Überprüfung nicht abgeschlossen werden konnte, bevor der Arbeitsspeicher des Testgerätes gefüllt war. Dies lässt sich unter anderem auf die bereits beschriebenen Performanceschwachstellen zurückführen, die sich durch eine optimierte Implementierung gegebenenfalls beheben lassen. Dennoch konnte gezeigt werden, dass generell eine Lösung für Anfragen dieser Art vom Solver gefunden werden kann.

6 Folgerungen

Zusammenfassend lässt sich sagen, dass die ursprünlich gesetzten Ziele zu unterschiedlichen Graden erreicht werden konnten. Im Zeitraum der Bearbeitungszeit konnte ein nahezu vollumfänglicher Schach-Interpreter in Racket implementiert werden. Die Funktionalität wurde dazu mit Unit-Tests kontrolliert und kann alleinstehend als fehlerfrei bezeichnet werden. Die zu einem vollständigen Schachspiel fehlenden Komponenten sind Regeln zu Rochade, En Passant, Promotion und Remis. Diese wurden in Hinsicht auf die weiteren Ziele als vernachlässigbar eingestuft.

Weiter wurde das Programm erfolgreich in Rosette überführt. Genauer wurde es in den sicheren Dialekt, gekennzeichnet durch **rosette/safe**, konvertiert. Dazu gehört es das Programm von Konstrukten zu befreien, die unlifted sind. In diesem Fall bedeutete es jegliche Operationen auf Strings auszutauschen und die Eingaben vor der Ausführung des Codes in eine interne Repräsentation ohne Strings zu übertragen.

Im Bezug auf die entwickelten Queries sind die Ergebnisse nicht so eindeutig. Prinzipiell konnten die gewünschten Anfragen mit den Werkzeugen Programmsynthese und Angelic Execution umgesetzt werden. Es wurden erfolgreich Problemstellungen aus zwei Kategorien vom Solver gelöst. Die erste war eine Zugfolge zu finden, die eine Belegung des Spielbretts in eine andere überführt. Die zweite sollte eine Zugfolge finden, die eine Belegung des Spielbretts in einen Mattzustand überführt. Es konnte also festgestellt werden, dass Probleme nichtdeterministischer Natur, gegeben durch die abwechselnde Spielerreihenfolge, mit Rosette gelöst werden können.

Schon bei der ersten, aber besonders bei der zweiten Kategorie zeigten sich jedoch einige Limitierungen bezogen auf die Performance. Die Ausgaben des Profilers (Anhang A) bestätigten die bereits getroffene Vermutung, dass bei der Auflösung der Queries eine hohe Anzahl geöffneter Pfade bearbeitet werden müssen. Hinzu kommt, dass die Darstellung des Spielbretts als Liste im Hinblick auf die Solveranfragen nicht optimal ist.

Es wurden Versuche zur Verbesserung der Performance unternommen, beispielsweise die Konkretisierung der Eingaben oder die Verwendung eigener Methoden. Letztendlich konnten die Problemstellen identifiziert werden, aber im Rahmen der Arbeit wurde keine deutlich performantere Version erreicht. Es wurden jedoch weitere Lösungsansätze entwickelt, die in nachfolgender Forschung verfolgt werden könnten. Ein vielversprechender Ansatz ist die Entwicklung des Interpreters mit einer Repräsentation des Spielbretts durch Bitvektoren. Operationen auf diesen sollten selbst in hoher Anzahl vergleichbar schnell ausgeführt werden. Andererseits könnte mit dem erlangten Wissen über die Verhaltensweisen des Solvers in Richtung eines Programmablaufes geforscht werden, mit dem sich die Anzahl der geöffneten Pfade reduzieren lassen. Dennoch bleibt die Frage, wie gut sich Rosette für die Lösung des gestellten Problems eignet, vorerst offen. Die generelle Machbarkeit kann jedoch bereits bestätigt werden.

Die aufgetretenen Probleme wurden dokumentiert und sowohl Ursachen als auch gefundene Lösungen oder Lösungsansätze besprochen. Es sollte möglich sein auf diesen Ergebnissen aufzubauen und informierter mit einem neuen Projekt zu beginnen, ohne auf die gleichen Hindernisse zu stoßen. Diese Arbeit liefert außerdem Kenntnisse, die schon in der Konzeptionsphase eines Projekts dieser Größenordnung relevant sind. So haben sich die Entwicklung eines Prototypen, der einen Interpreter mit verringertem Funktionsumfang enthält, und das darauf basierende Experimentieren mit Queries als besonders nützlich herausgestellt. Der Grund dafür sind die vielen möglichen Fehlerquellen, die bei der Ausführung von Anfragen zu falschen Ausgaben führen können. Diese sind leichter in einer verkürzten Version des Programms zu beheben. Außerdem können auf diese Weise früh die Wechselwirkungen zwischen Interpreter und Queries beobachtet werden, wodurch die weitere Konzeption vereinfacht wird. Weiter finden sich Ideen für eine verbesserte Version der entwickelten

Software, die auf Ergebnissen der durchgeführten Versuche basieren. Insgesamt lässt sich der Erkundungsteil der Problemstellung als erfolgreich bewerten.

Die Entwicklung des Schach-Interpreters sollte als Fallstudie dienen, um Informationen und Kenntnisse für ein übergeordnetes Ziel zu erhalten. Dieses Ziel ist die Nutzung der Queries im Zusammenhang mit B-Maschinen. Dadurch, dass unklar ist wie ähnlich ein B-Interpreter dem entwickelten Schach-Interpreter sein kann, ist es schwierig allgemeine Aussagen anhand der Ergebnisse zu treffen.

Sollten sich B-Maschinen in Rosette Code überführen lassen, könnten jedoch einige Vermutungen über die gewonnenen Werkzeuge aufgestellt werden. Beispielsweise wurde gezeigt, dass mit Angelic Execution eine Belegung von Eingabewerten gefunden werden kann, die zu festgelegten Zustandsveränderungen führt. Dabei sind auch die Ausführungen mehrerer und auch verschachtelter Methoden, wie die **move**-Methode des Schach-Interpreters, möglich. Gibt es entsprechende Flags, könnte damit beispielsweise überprüft werden ob es eine Eingabekombination gibt, die einen Fehlerzustand herbeiführt. Wird statt Angelic Execution eine **synthesize**-Query mit Grammatik verwendet, könnte sogar eine Abfolge von Methodenaufrufen gefunden werden, die einen solchen Zustand produziert.

Ein solches Projekt unterliegt jedoch ebenso den besprochenen Limitierungen. Somit lässt sich nicht vorhersagen, ob eine Übertragung von B-Maschinen vollumfänglich möglich ist. Es könnte sich auch herausstellen, dass sich nur eine Teilmenge abbilden und darauf mit praktikabler Performance Queries absetzen lässt. Das hängt beispielsweise davon ab wie sich Objekte aus B in Rosette Datentypen übertragen lassen.

Um dieses Thema weiter zu erschließen könnte zukünftige Forschung die in dieser Arbeit entwickelte Grundlage optimieren. Dabei könnten bereits diskutierte Ansätze verfolgt werden. Andere Anhaltspunkte könnten sich durch das tiefere Auseinandersetzen mit der Forschung zu Performance und Optimierung von solverunterstützten Tools [SLTB+06, BT18, WLH+17] ergeben. Dieser Bereich beschäftigt sich mit der Frage wie symbolische Compiler ein Programm zu Constraints reduzieren können, die sich effektiv von Solvern lösen lassen. Weiter wird darauf eingegangen wie letztendlich Programme für tatsächliche Probleme konzipiert werden, die die Verhaltensweisen des Compilers bestmöglich nutzen, um solche Constraints zu produzieren. Dies ist keine leichte Aufgabe, unter anderem weil ein symbolischer Compiler in der Regel alle Pfade des Programms ausführt [BT18]. Eine weitere Möglichkeit ist eine neue Fallstudie, in der ein anderer Aspekt als Nichtdeterminismus erkundet wird.

Anhang

A Symprofiler Ausgaben

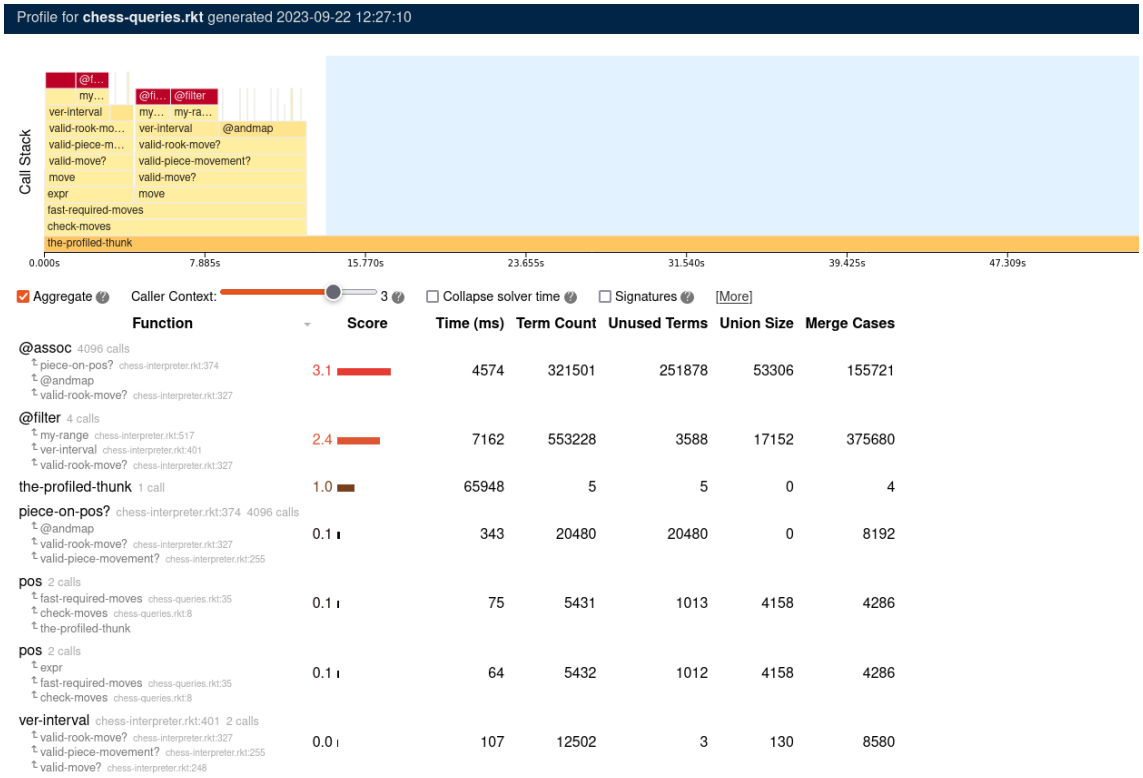

Abbildung 1: Ausgabe einer getesteten Query

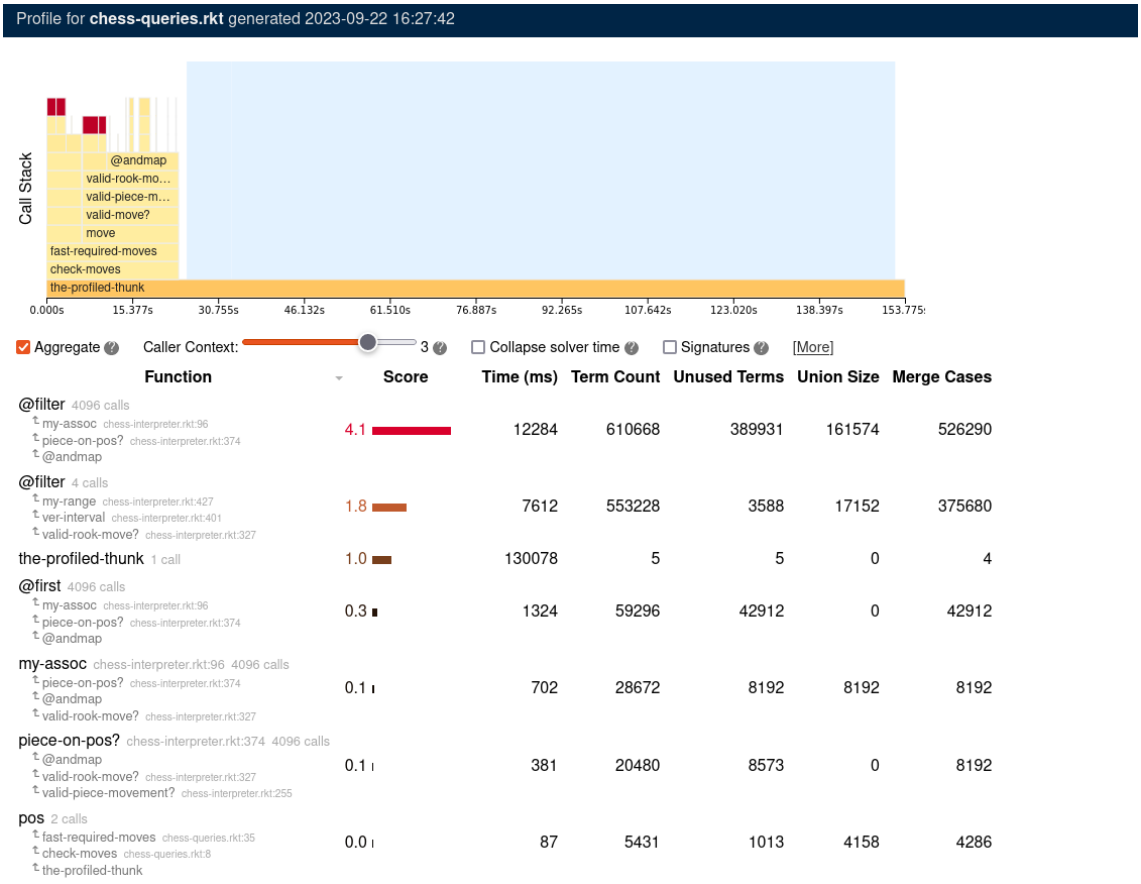

Abbildung 2: Ausgabe bei Verwendung von eigener assoc-Methode

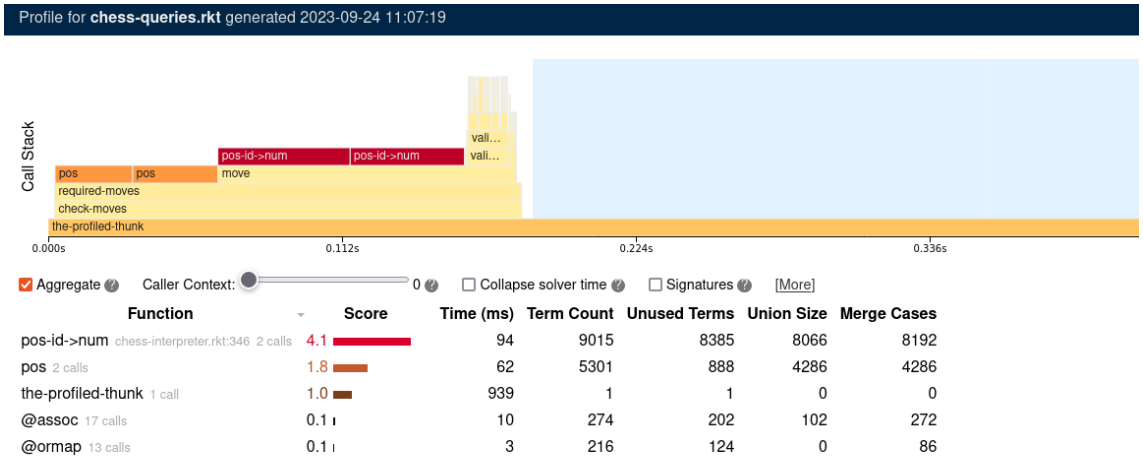

Abbildung 3: Ausgabe bei Verwendung des reduzierten Interpreters

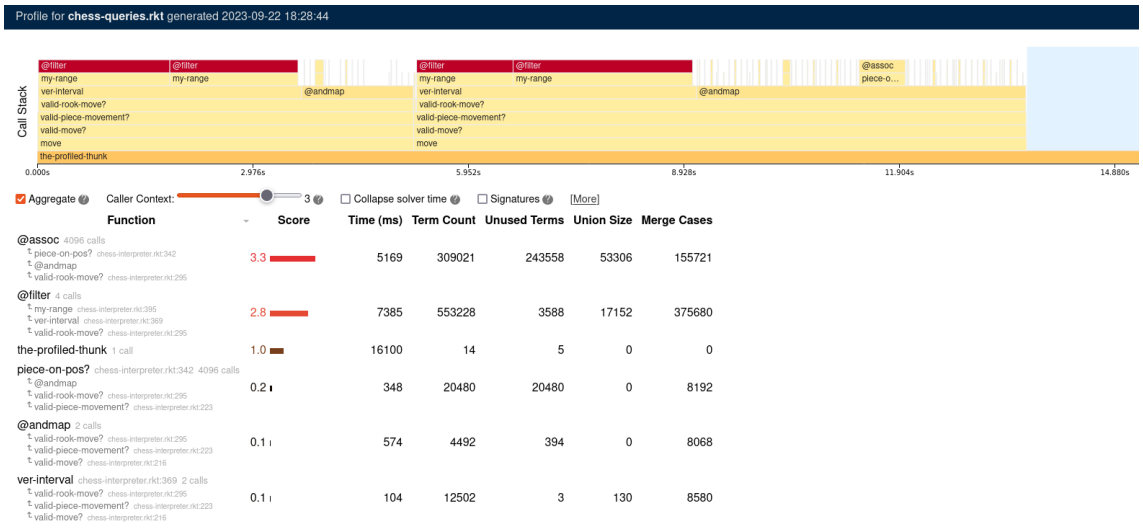

Abbildung 4: Ausgabe bei Verwendung von Angelic Execution

Abbildungsverzeichnis

Quellcodeverzeichnis

Literatur

[Abr96] Abrial, Jean-Raymond: *The B-Book: Assigning Programs to Meanings*. New York, NY, USA : Cambridge University Press, 1996

[Abr10] Abrial, Jean-Raymond: *Modeling in Event-B: System and Software Engineering*. New York, NY, USA : Cambridge University Press, 2010

[BT17] Bornholt, James ; Torlak, Emina: Synthesizing Memory Models from Framework Sketches and Litmus Tests. In: *SIGPLAN Not.* 52 (2017), Juni, Nr. 6, S. 467–481

[BT18] Bornholt, James ; Torlak, Emina: Finding code that explodes under symbolic evaluation. In: *Proceedings of the ACM on Programming Languages* 2 (2018), Oktober, S. 1–26

[BTP17] Butler, Eric ; Torlak, Emina ; Popović, Zoran: Synthesizing Interpretable Strategies for Solving Puzzle Games. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*. New York, NY, USA : Association for Computing Machinery, August 2017 (FDG '17), S. 1–10

[CLW+17] Chu, Shumo ; Li, Daniel ; Wang, Chenglong ; Cheung, Alvin ; Suciu, Dan: Demonstration of the Cosette Automated SQL Prover. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. New York, NY, USA : Association for Computing Machinery, Mai 2017 (SIGMOD '17), S. 1591–1594

[DMB08] De Moura, Leonardo ; Bjørner, Nikolaj: 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*. Berlin, Heidelberg : Springer-Verlag, März 2008 (TACAS'08/ETAPS'08), S. 337–340

[Hö19] Höfges, Philip: *Chess in EventB*. https://github.com/pkoerner/b-chess-example/tree/master, 2019. – Accessed: 2023-09-25

[NPM23] NPM: *NPM documentation*. https://docs.npmjs.com/, 2023. – Accessed: 2023-09-25

[NVGTW20] Nelson, Luke ; Van Geffen, Jacob ; Torlak, Emina ; Wang, Xi: Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USA : USENIX Association, November 2020 (OSDI'20), S. 41–61

[PCG+23] Pinckney, Donald ; Cassano, Federico ; Guha, Arjun ; Bell, Jon ; Culpo, Massimiliano ; Gamblin, Todd: Flexible and Optimal Dependency Management via Max-SMT. In: *Proceedings of the 45th International Conference on Software Engineering*. New York, NY, USA : IEEE Press, Juli 2023 (ICSE '23), S. 1418–1429

[PLT+16] Pernsteiner, Stuart ; Loncaric, Calvin ; Torlak, Emina ; Tatlock, Zachary ; Wang, Xi ; Ernst, Michael D. ; Jacky, Jonathan: Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In: *Computer Aided Verification* Bd. 9780. Cham : Springer International Publishing, Juli 2016 (LNTSC), S. 23–41

[Rac23] Racket: *Racket Programming Language*. https://racket-lang.org/, 2023. – Accessed: 2023-09-25

[SLTB+06] Solar-Lezama, Armando ; Tancau, Liviu ; Bodik, Ratislav ; Seshia, Sanjit ; Saraswat, Vijay: Combinatorial Sketching for Finite Programs. In: *SIGOPS Oper. Syst. Rev.* 40 (2006), Oktober, Nr. 5, S. 404–415

[TB13] Torlak, Emina ; Bodik, Ratislav: Growing solver-aided languages with rosette. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. New York, NY, USA : Association for Computing Machinery, Oktober 2013 (Onward! 2013), S. 135–152

[TB14] Torlak, Emina ; Bodik, Rastislav: A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In: *SIGPLAN Not.* 49 (2014), Juni, Nr. 6, S. 530–541

[Tor23a] Torlak, Emina: *The Rosette Guide*. https://docs.racket-lang.org/rosette-guide/index.html, 2023. – Accessed: 2023-09-25

[Tor23b] Torlak, Emina: *The Rosette Language*. https://emina.github.io/rosette/index.html, 2023. – Accessed: 2023-07-10

[WLH+17] Weitz, Konstantin ; Lyumbomirsky, Steven ; Heule, Stefan ; Torlak, Emina ; Ernst, Michael D. ; Tatlock, Zachary: SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. In: *Proc. ACM Program. Lang.* 1 (2017), August, Nr. ICFP, S. 1–28

[XPa17] XPath: *XML Path Language*. https://www.w3.org/TR/xpath/, 2017. – Accessed: 2023-07-10